Praise for *Javascript® Bible*

"*JavaScript® Bible* is the definitive resource in JavaScript programming. I am never more than three feet from my copy."

— Steve Reich, CEO, PageCoders

"This book is a must-have for any web developer or programmer."

— Thoma Lile, President, Kanis Technologies, Inc.

"Outstanding book. I would recommend this book to anyone interested in learning to develop advanced Web sites. Mr. Goodman did an excellent job of organizing this book and writing it so that even a beginning programmer can understand it."

— Jason Hensley, Director of Internet Services, NetVoice, Inc.

"Goodman is always great at delivering clear and concise technical books!"

— Dwayne King, Chief Technology Officer, White Horse

"*JavaScript® Bible* is well worth the money spent!"

— Yen C.Y. Leong, IT Director, Moo Mooltimedia, a member of SmartTransact Group

"A must-have book for any internet developer."

— Uri Fremder, Senior Consultant, TopTier Software

"I love this book! I use it all the time, and it always delivers. It's the only JavaScript book I use!"

— Jason Badger, Web Developer

"Whether you are a professional or a beginner, this is a great book to get."

— Brant Mutch, Web Application Developer, Wells Fargo Card Services, Inc.

"I never thought I'd ever teach programming before reading your book [*JavaScript® Bible*]. It's so simple to use — the Programming Fundamentals section brought it all back! Thank you for such a wonderful book, and for breaking through my programming block!"

— Susan Sann Mahon, Certified Lotus Instructor, TechNet Training

"Danny Goodman is very good at leading the reader into the subject. *JavaScript® Bible* has everything we could possibly need."

— Philip Gurdon

"An excellent book that builds solidly from whatever level the reader is at. A book that is both witty and educational."

— Dave Vane

"I continue to use the book on a daily basis and would be lost without it."

— Mike Warner, Founder, Oak Place Productions

"*JavaScript® Bible* is by *far* the best JavaScript resource I've ever seen (and I've seen quite a few)."

— Robert J. Mirro, Independent Consultant, RJM Consulting

JavaScript® Bible

Seventh Edition

JavaScript® Bible

Seventh Edition

Danny Goodman
Michael Morrison
Paul Novitski
Tia Gustaff Rayl

Wiley Publishing, Inc.

JavaScript® Bible, Seventh Edition

Published by
Wiley Publishing, Inc.
10475 Crosspoint Boulevard
Indianapolis, IN 46256
`www.wiley.com`

Copyright © 2010 by Wiley Publishing, Inc., Indianapolis, Indiana

Published simultaneously in Canada

ISBN: 978-0-470-52691-0

Manufactured in the United States of America

10 9 8 7 6 5 4 3 2 1

No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, except as permitted under Sections 107 or 108 of the 1976 United States Copyright Act, without either the prior written permission of the Publisher, or authorization through payment of the appropriate per-copy fee to the Copyright Clearance Center, 222 Rosewood Drive, Danvers, MA 01923, (978) 750-8400, fax (978) 646-8600. Requests to the Publisher for permission should be addressed to the Permissions Department, John Wiley & Sons, Inc., 111 River Street, Hoboken, NJ 07030, (201) 748-6011, fax (201) 748-6008, or online at `http://www.wiley.com/go/permissions`.

Limit of Liability/Disclaimer of Warranty: The publisher and the author make no representations or warranties with respect to the accuracy or completeness of the contents of this work and specifically disclaim all warranties, including without limitation warranties of fitness for a particular purpose. No warranty may be created or extended by sales or promotional materials. The advice and strategies contained herein may not be suitable for every situation. This work is sold with the understanding that the publisher is not engaged in rendering legal, accounting, or other professional services. If professional assistance is required, the services of a competent professional person should be sought. Neither the publisher nor the author shall be liable for damages arising herefrom. The fact that an organization or web site is referred to in this work as a citation and/or a potential source of further information does not mean that the author or the publisher endorses the information the organization or web site may provide or recommendations it may make. Further, readers should be aware that Internet web sites listed in this work may have changed or disappeared between when this work was written and when it is read.

For general information on our other products and services please contact our Customer Care Department within the United States at (877) 762-2974, outside the United States at (317) 572-3993 or fax (317) 572-4002.

Wiley also publishes its books in a variety of electronic formats. Some content that appears in print may not be available in electronic books.

Library of Congress Control Number: 2010923547

Trademarks: Wiley and the Wiley logo are trademarks or registered trademarks of John Wiley & Sons, Inc. and/or its affiliates, in the United States and other countries, and may not be used without written permission. JavaScript is a registered trademark of Oracle America, Inc. All other trademarks are the property of their respective owners. Wiley Publishing, Inc. is not associated with any product or vendor mentioned in this book.

To Tanya, with whom I share this loving orbit with Briar and Callum in our own cozy Klemperer rosette.
— Paul Novitski

To my husband, Edward, whose love, support, and encouragement kept me going through this and all my adventures, and to the memory of my parents who inspired me to step out in faith.
— Tia Gustaff Rayl

About the Authors

Danny Goodman is the author of numerous critically acclaimed and best-selling books, including *The Complete HyperCard Handbook*, *Danny Goodman's AppleScript Handbook*, *Dynamic HTML: The Definitive Reference*, and *JavaScript & DHTML Cookbook*. He is a renowned authority on and expert teacher of computer scripting languages. His writing style and pedagogy continue to earn praise from readers and teachers around the world.

Michael Morrison is a writer, developer, toy inventor, and author of a variety of books covering topics such as Java, C++, Web scripting, XML, game development, and mobile devices. Some of Michael's notable writing projects include *Faster Smarter HTML and XML*, *Teach Yourself HTML & CSS in 24 Hours*, and *Beginning Game Programming*. Michael is also the founder of Stalefish Labs (www.stalefishlabs.com), an entertainment company specializing in unusual games, toys, and interactive products.

Paul Novitski has been writing software as a freelance programmer since 1981. He once taught himself BASIC in order to write a machine language disassembler so that he could lovingly hack Wang's OIS microcode. He has focused on internet programming since the late '90s. His company, Juniper Webcraft, produces HTML-strict websites featuring accessible, semantic markup, separation of development layers, and intuitive user interfaces. He knows the righteousness of elegant code, the poignancy of living on the bleeding edge of wilderness, the sweet melancholy of mbira music, and the scorching joy of raising twin boys.

Tia Gustaff Rayl is a consultant who does development and training in database and Web technologies. Most recently she has published courseware for XHTML, CSS, JavaScript, and SQL. It comes as no surprise to those who know her that she began her software career with degrees in English and Education from the University of Florida. As is usual for most newcomers to the field, her introduction to computing was maintaining software. She went on to a long-standing career in the software industry in full life cycle system, application, and database development; project management; and training for PC and mainframe environments. In the mid-nineties she worked on early Web-enabled database applications, adding JavaScript to her repertoire. She continues to take on development projects to maintain her code-slinging skills. If she had any spare time (and money) she would go on an around-the-world cruise with her husband and two dogs.

About the Technical Editor

Benjamin Schupak holds a master's degree in computer science and has more than 11 years of professional programming experience for large corporations and U.S. federal departments. He lives in the New York metro area and enjoys traveling.

Credits

Executive Editor
Carol Long

Project Editor
John Sleeva

Technical Editor
Benjamin Schupack

Production Editor
Rebecca Anderson

Editorial Director
Robyn B. Siesky

Editorial Manager
Mary Beth Wakefield

Marketing Manager
Ashley Zurcher

Production Manager
Tim Tate

Vice President and Executive Group Publisher
Richard Swadley

Vice President and Executive Publisher
Barry Pruett

Associate Publisher
Jim Minatel

Project Coordinator, Cover
Lynsey Stanford

Proofreaders
Maraya Cornell, Word One New York
Sheilah Ledwidge, Word One New York

Indexer
Robert Swanson

Cover Designer
Michael E. Trent

Cover Image
© Joyce Haughey

Acknowledgments

My gratitude for being given the opportunity and the latitude to work on this immense tome extends to Carol Long of Wiley, editor John Sleeva, Carole Jelen at Waterside Productions, Julian Hall, my partner in crime at Juniper Webcraft, and, above all, my sweet, loving, supportive, and nearly homicidal spouse Tanya Wright, all of them long-suffering and all to be commended for not actually throttling me in the course of this overlong birthing. The solid foundation of research and explication established by Danny Goodman and his past collaborators in previous editions is awesome and I am humbled to have been able to help move this great body of knowledge forward to the next step. Tia Gustaff Rayl, who with wings spread and sparks flying rode in to the rescue, has been a kick in the pants to work with; she's smart and funny, but more than that she is gifted with the precise serendipity and good taste to share a birthday with me.

—Paul Novitski

I have been blessed with the support of many people who have reviewed my work and encouraged me along the way. I could never have done any of this without the devoted support of my husband, Edward. I love him without cease and thank God for bringing him into my life. I have been blessed by the remarkable patience of an incredibly tolerant immediate and extended family, as well as amused friends, who put up with my "Hi, I love you. Don't talk to me right now." way of answering the phone and greeting visitors at the door. My husband, family and friends are the ones who sacrificed the most for this book. Thank you all. I want to thank Paul for being great to work with and for his wry sense of humor which left me on the floor laughing just when I most needed a lift. I want to thank my editor, John Sleeva, who with great patience and humor guided me through the maze of the publishing world. I also want to thank Rebecca Anderson and Maraya Cornell for forcing me to be a better writer. Finally, I want to thank Miss Bigelow, my 11th and 12th grade English teacher, who instilled within me a great respect and love for the written word.

—Tia Gustaff Rayl

Contents at a Glance

Introduction xxv

Part I: Getting Started with JavaScript 1

Chapter 1: JavaScript's Role in the World Wide Web and Beyond 3
Chapter 2: Developing a Scripting Strategy 15
Chapter 3: Selecting and Using Your Tools 27
Chapter 4: JavaScript Essentials 37

Part II: JavaScript Tutorial 59

Chapter 5: Your First JavaScript Script 61
Chapter 6: Browser and Document Objects 77
Chapter 7: Scripts and HTML Documents 95
Chapter 8: Programming Fundamentals, Part I 109
Chapter 9: Programming Fundamentals, Part II 121
Chapter 10: Window and Document Objects 135
Chapter 11: Forms and Form Elements 153
Chapter 12: Strings, Math, and Dates 179
Chapter 13: Scripting Frames and Multiple Windows 191
Chapter 14: Images and Dynamic HTML 207

Part III: JavaScript Core Language Reference 223

Chapter 15: The String Object 225
Chapter 16: The Math, Number, and Boolean Objects 269
Chapter 17: The Date Object 285
Chapter 18: The Array Object 311
Chapter 19: JSON — Native JavaScript Object Notation 357
Chapter 20: E4X — Native XML Processing 363
Chapter 21: Control Structures and Exception Handling 373
Chapter 22: JavaScript Operators 411
Chapter 23: Function Objects and Custom Objects 437
Chapter 24: Global Functions and Statements 481

Part IV: Document Objects Reference 501

Chapter 25: Document Object Model Essentials 503
Chapter 26: Generic HTML Element Objects 537
Chapter 27: Window and Frame Objects 739
Chapter 28: Location and History Objects 881
Chapter 29: Document and Body Objects 907
Chapter 30: Link and Anchor Objects 995
Chapter 31: Image, Area, Map, and Canvas Objects 1003
Chapter 32: Event Objects 1043

Part V: Appendixes 1123
Chapter A: JavaScript and Browser Objects Quick Reference 1125
Chapter B: What's on the CD-ROM 1133

Index 1137

Bonus Chapters on the CD-ROM

Part VI: Document Objects Reference (continued) BC1
Chapter 33: Body Text Objects BC2
Chapter 34: The Form and Related Objects BC103
Chapter 35: Button Objects BC128
Chapter 36: Text-Related Form Objects BC153
Chapter 37: Select, Option, and FileUpload Objects BC177
Chapter 38: Style Sheet and Style Objects BC207
Chapter 39: Ajax, E4X, and XML BC272
Chapter 40: HTML Directive Objects BC289
Chapter 41: Table and List Objects BC303
Chapter 42: The Navigator and Other Environment Objects BC360
Chapter 43: Positioned Objects BC411
Chapter 44: Embedded Objects BC448
Chapter 45: The Regular Expression and RegExp Objects BC465

Part VII: More JavaScript Programming BC491
Chapter 46: Data-Entry Validation BC492
Chapter 47: Scripting Java Applets and Plug-Ins BC524
Chapter 48: Debugging Scripts BC564
Chapter 49: Security and Netscape Signed Scripts BC590
Chapter 50: Cross-Browser Dynamic HTML Issues BC608
Chapter 51: Internet Explorer Behaviors BC623

Part VIII: Applications BC636
Chapter 52: Application: Tables and Calendars BC637
Chapter 53: Application: A Lookup Table BC652
Chapter 54: Application: A Poor Man's Order Form BC665
Chapter 55: Application: Outline-Style Table of Contents BC674
Chapter 56: Application: Calculations and Graphics BC695
Chapter 57: Application: Intelligent "Updated" Flags BC705
Chapter 58: Application: Decision Helper BC715
Chapter 59: Application: Cross-Browser DHTML Map Puzzle BC747
Chapter 60: Application: Transforming XML Data BC764
Chapter 61: Application: Creating Custom Google Maps BC782

Part IX: Appendixes (continued) BC799
Appendix C: JavaScript Reserved Words BC800
Appendix D: Answers to Tutorial Exercises BC801
Appendix E: JavaScript and DOM Internet Resources BC818

Contents

Introduction xxv

Part I: Getting Started with JavaScript 1

Chapter 1: JavaScript's Role in the World Wide Web and Beyond 3

Competing for Web Traffic 4
Other Web Technologies 4
JavaScript: A Language for All 10
JavaScript: The Right Tool for the Right Job 12

Chapter 2: Developing a Scripting Strategy 15

Browser Leapfrog 15
Duck and Cover 16
Compatibility Issues Today 17
Developing a Scripting Strategy 22

Chapter 3: Selecting and Using Your Tools 27

The Software Tools 27
Setting Up Your Authoring Environment 28
Validate, Validate, Validate 31
Creating Your First Script 31

Chapter 4: JavaScript Essentials 37

Combining JavaScript with HTML 37
Designing for Compatibility 51
Language Essentials for Experienced Programmers 54

Part II: JavaScript Tutorial 59

Chapter 5: Your First JavaScript Script 61

What Your First Script Will Do 61
Entering Your First Script 62
Have Some Fun 74
Exercises 75

Chapter 6: Browser and Document Objects 77

Scripts Run the Show 77
When to Use JavaScript 78
The Document Object Model 79
When a Document Loads 82

Object References 85
Node Terminology 87
What Defines an Object? 88
Exercises 93

Chapter 7: Scripts and HTML Documents 95

Connecting Scripts to Documents 95
JavaScript Statements 99
When Script Statements Execute 100
Viewing Script Errors 104
Scripting versus Programming 105
Exercises 106

Chapter 8: Programming Fundamentals, Part I 109

What Language Is This? 109
Working with Information 109
Variables 110
Expressions and Evaluation 112
Data Type Conversions 115
Operators 116
Exercises 118

Chapter 9: Programming Fundamentals, Part II 121

Decisions and Loops 121
Control Structures 122
Repeat Loops 124
Functions 124
Curly Braces 128
Arrays 129
Exercises 133

Chapter 10: Window and Document Objects 135

Top-Level Objects 135
The window Object 135
window Properties and Methods 139
The location Object 142
The navigator Object 143
The document Object 143
Exercises 152

Chapter 11: Forms and Form Elements 153

The Form object 153
Form Controls as Objects 158
Passing Elements to Functions with this 170
Submitting and Prevalidating Forms 173
Exercises 177

Chapter 12: Strings, Math, and Dates 179

Core Language Objects 179
String Objects 180
The Math Object 183
The Date Object 184

Date Calculations 186
Exercises 189

Chapter 13: Scripting Frames and Multiple Windows 191

Frames: Parents and Children 191
References Among Family Members 194
Frame-Scripting Tips 195
About iframe Elements 196
Highlighting Footnotes: A Frameset Scripting Example 196
References for Multiple Windows 202
Exercises 206

Chapter 14: Images and Dynamic HTML 207

The Image Object 207
Rollovers Without Scripts 216
The javascript: Pseudo-URL 219
Popular Dynamic HTML Techniques 220
Exercises 222

Part III: JavaScript Core Language Reference 223

Chapter 15: The String Object 225

String and Number Data Types 225
String Object 228
String Utility Functions 261
URL String Encoding and Decoding 267

Chapter 16: The Math, Number, and Boolean Objects 269

Numbers in JavaScript 269
Math Object 276
Number Object 280
Boolean Object 284

Chapter 17: The Date Object 285

Time Zones and GMT 285
The Date Object 287
Validating Date Entries in Forms 304

Chapter 18: The Array Object 311

Structured Data 311
Creating an Empty Array 312
Populating an Array 313
JavaScript Array Creation Enhancements 314
Deleting Array Entries 315
Parallel Arrays 315
Multidimensional Arrays 320
Simulating a Hash Table 321
Array Object 323
Array Comprehensions 353

Destructuring Assignment 354
Compatibility with Older Browsers 355

Chapter 19: JSON — Native JavaScript Object Notation 357

How JSON Works 357
Sending and Receiving JSON Data 359
JSON Object 360
Security Concerns 361

Chapter 20: E4X — Native XML Processing 363

XML 363
ECMAScript for XML (E4X) 364

Chapter 21: Control Structures and Exception Handling 373

If and If. . .Else Decisions 373
Conditional Expressions 379
The switch Statement 380
Repeat (for) Loops 384
The while Loop 388
The do-while Loop 390
Looping through Properties (for-in) 390
The with Statement 392
Labeled Statements 393
Exception Handling 397
Using try-catch-finally Constructions 398
Throwing Exceptions 402
Error Object 407

Chapter 22: JavaScript Operators 411

Operator Categories 411
Comparison Operators 412
Equality of Disparate Data Types 413
Connubial Operators 415
Assignment Operators 418
Boolean Operators 420
Bitwise Operators 424
Object Operators 425
Miscellaneous Operators 430
Operator Precedence 433

Chapter 23: Function Objects and Custom Objects 437

Function Object 437
Function Application Notes 447
Creating Your Own Objects with Object-Oriented JavaScript 458
Object-Oriented Concepts 470
Object Object 474

Chapter 24: Global Functions and Statements 481

Functions 482
Statements 492
WinIE Objects 496

Part IV: Document Objects Reference 501

Chapter 25: Document Object Model Essentials 503

The Object Model Hierarchy 503
How Document Objects Are Born 505
Object Properties 506
Object Methods 507
Object Event Handlers 508
Object Model Smorgasbord 509
Basic Object Model 510
Basic Object Model Plus Images 511
Navigator 4–Only Extensions 511
Internet Explorer 4+ Extensions 512
Internet Explorer 5+ Extensions 515
The W3C DOM 516
Scripting Trends 532
Standards Compatibility Modes (DOCTYPE Switching) 534
Where to Go from Here 535

Chapter 26: Generic HTML Element Objects 537

Generic Objects 537

Chapter 27: Window and Frame Objects 739

Window Terminology 739
Frames 740
window Object 746
frame Element Object 854
frameset Element Object 862
iframe Element Object 868
popup Object 875

Chapter 28: Location and History Objects 881

location Object 881
history Object 900

Chapter 29: Document and Body Objects 907

document Object 908
body Element Object 981
TreeWalker Object 990

Chapter 30: Link and Anchor Objects 995

Anchor, Link, and a Element Objects 995

Chapter 31: Image, Area, Map, and Canvas Objects 1003

Image and img Element Objects 1003
area Element Object 1024
map Element Object 1028
canvas Element Object 1032

Chapter 32: Event Objects 1043

Why "Events"? 1044
Event Propagation 1045
Referencing the event Object 1059
Binding Events 1059
event Object Compatibility 1064
Dueling Event Models 1066
Event Types 1070
NN6+/Moz event Object 1097

Part V: Appendixes 1123

Appendix A: JavaScript and Browser Objects Quick Reference 1125

Appendix B: What's on the CD-ROM 1133

Index 1137

Bonus Chapters on the CD-ROM

Part VI: Document Objects Reference (continued) BC1

Chapter 33: Body Text Objects BC2

Chapter 34: The Form and Related Objects BC103

Chapter 35: Button Objects BC128

Chapter 36: Text-Related Form Objects BC153

Chapter 37: Select, Option, and FileUpload Objects BC177

Chapter 38: Style Sheet and Style Objects BC207

Chapter 39: Ajax, E4X, and XML BC272

Chapter 40: HTML Directive Objects BC289

Chapter 41: Table and List Objects BC303

Chapter 42: The Navigator and Other Environment Objects BC360

Chapter 43: Positioned Objects BC411

Chapter 44: Embedded Objects BC448
Chapter 45: The Regular Expression and RegExp Objects BC465

Part VII: More JavaScript Programming BC491

Chapter 46: Data-Entry Validation BC492
Chapter 47: Scripting Java Applets and Plug-Ins BC524
Chapter 48: Debugging Scripts BC564
Chapter 49: Security and Netscape Signed Scripts BC590
Chapter 50: Cross-Browser Dynamic HTML Issues BC608
Chapter 51: Internet Explorer Behaviors BC623

Part VIII: Applications BC636

Chapter 52: Application: Tables and Calendars BC637
Chapter 53: Application: A Lookup Table BC652
Chapter 54: Application: A Poor Man's Order Form BC665
Chapter 55: Application: Outline-Style Table of Contents BC674
Chapter 56: Application: Calculations and Graphics BC695
Chapter 57: Application: Intelligent "Updated" Flags BC705
Chapter 58: Application: Decision Helper BC715
Chapter 59: Application: Cross-Browser DHTML Map Puzzle BC747
Chapter 60: Application: Transforming XML Data BC764
Chapter 61: Application: Creating Custom Google Maps BC782

Part IX: Appendixes (continued) BC799

Appendix C: JavaScript Reserved Words BC800
Appendix D: Answers to Tutorial Exercises BC801
Appendix E: JavaScript and DOM Internet Resources BC818

Introduction

This seventh edition of the *JavaScript Bible* represents knowledge and experience accumulated over fifteen years of daily work in JavaScript and a constant monitoring of newsgroups for questions, problems, and challenges facing scripters at all levels. Our goal is to help you avoid the same frustration and head-scratching we and others have experienced through multiple generations of scriptable browsers.

While the earliest editions of this book focused on the then-predominant Netscape Navigator browser, the browser market share landscape has changed through the years. For many years, Microsoft took a strong lead with its Internet Explorer, but more recently, other browsers that support industry standards are finding homes on users' computers. The situation still leaves an age-old dilemma for content developers: designing scripted content that functions equally well in both standards-compliant and proprietary environments. The job of a book claiming to be the "bible" is not only to present both the standard and proprietary details when they diverge, but also to show you how to write scripts that blend the two so that they work on the wide array of browsers visiting your sites or web applications. Empowering you to design and write good scripts is our passion, regardless of browser. It's true that our bias is toward industry standards, but not to the exclusion of proprietary features that may be necessary to get your content and scripting ideas flowing equally well on today's and tomorrow's browsers.

Organization and Features of This Edition

Like the previous three editions of the *JavaScript Bible*, this seventh edition contains far more information than can be printed and bound into a single volume. The complete contents can be found in the electronic version of this book (in PDF form) on the CD-ROM that accompanies the book. This edition is structured in such a way as to supply the most commonly needed information in its entirety in the printed portion of the book. Content that you use to learn the fundamentals of JavaScript and reference frequently are at your fingertips in the printed version, while chapters with more advanced content are in the searchable electronic version on the CD-ROM. Here are some details about the book's structure.

Part I: Getting Started with JavaScript

Part I of the book begins with a chapter that shows how JavaScript compares with Java and discusses its role within the rest of the World Wide Web. The web browser and scripting world have undergone significant changes since JavaScript first arrived on the scene. That's why Chapter 2 is devoted to addressing challenges facing scripters who must develop applications for both single- and cross-platform browser audiences amid rapidly changing standards efforts. Chapter 3 introduces some tools you can use to compose your pages and scripts, while Chapter 4 delves into the nitty-gritty of how to use JavaScript to run in a wide variety of browsers.

Part II: JavaScript Tutorial

All of Part II is handed over to a tutorial for newcomers to JavaScript. Ten lessons provide you with a gradual path through browser internals, basic programming skills, and genuine browser scripting, with an emphasis on industry standards as supported by most of the scriptable browsers in use today. Exercises follow at the end of each lesson to help reinforce what you just learned and challenge you to use your new knowledge (you'll find answers to the exercises in Appendix D, on the CD-ROM). The goal of the tutorial is to equip you with sufficient experience to start scripting simple pages right away while making it easier for you to understand the in-depth discussions and examples in the rest of the book.

Part III: JavaScript Core Language Reference

Reference information for the core JavaScript language fills Part III. In all reference chapters, a compatibility chart indicates the browser version that supports each object and object feature. Guide words near the tops of pages help you find a particular term quickly.

Part IV: Document Objects Reference

Part IV, the largest section of the book, provides in-depth coverage of the document object models as implemented in today's browsers, including the object used for modern Ajax applications. As with the core JavaScript reference chapters of Part III, these DOM chapters display browser compatibility charts for every object and object feature. One chapter in particular, Chapter 26, contains reference material that is shared by most of the remaining chapters of Part IV. To help you refer back to Chapter 26 from other chapters, a shaded tab along the outside edge of the page shows you at a glance where the chapter is located. Additional navigation aids include guide words near the tops of most pages to indicate which object and object feature is covered on the page. Note that the Objects Reference begun in Part IV of the book continues in Part VI on the CD, with an additional 13 chapters of reference material.

Part V: Appendixes

Appendix A offers a JavaScript and Browser Objects Quick Reference. Appendix B provides information about using the CD-ROM that comes with this book, which includes numerous bonus chapters and examples.

Part VI: Document Objects Reference (continued)

Beginning the portion of the book that resides only the accompanying CD, Part VI continues the document objects reference discussions that begin in Part IV by providing an additional 13 chapters of reference material.

Part VII: More JavaScript Programming

Chapters 46–51 discuss advanced JavaScript programming techniques, including data-entry validation, debugging, and security issues.

Part VIII: Applications

The final ten chapters of the book, available only on the CD-ROM, feature sample applications that cover the gamut from calendars to puzzles.

Part IX: Appendixes (continued)

The final three appendixes provide helpful reference information. These resources include a list of JavaScript reserved words in Appendix C, answers to Part II's tutorial exercises in Appendix D, and Internet resources in Appendix E.

CD-ROM

The CD-ROM is a gold mine of information. It begins with a PDF version of the entire contents of this seventh edition of the *JavaScript Bible*. This version includes bonus chapters covering:

- Dynamic HTML, data validation, plug-ins, and security
- Techniques for developing and debugging professional web-based applications
- Ten full-fledged JavaScript real-world applications

Another treasure trove on the CD-ROM is the `Listings` folder where you'll find over 300 ready-to-run HTML documents that serve as examples of most of the document object model and JavaScript vocabulary words in Parts III and IV. All of the bonus chapter example listings are also included. You can run these examples with your JavaScript-enabled browser, but be sure to use the `index.html` page in the Listings folder as a gateway to running the listings. We could have provided you with humorous little sample code fragments out of context, but we think that seeing full-fledged HTML documents (simple though they may be) for employing these concepts is important. We encourage you to manually type the script listings from the tutorial (Part II) of this book. We believe you learn a lot, even by aping listings from the book, as you get used to the rhythms of typing scripts in documents.

Be sure to check out the Chapter 4 listing file called `evaluator.html`. Many segments of Parts III and IV invite you to try out an object model or language feature with the help of an interactive workbench, called The Evaluator — a *JavaScript Bible* exclusive! You see instant results and will quickly learn how the feature works.

The Quick Reference from Appendix A is in PDF format on the CD-ROM for you to print out and assemble as a handy reference, if desired. Adobe Reader is also included on the CD-ROM, in case you don't already have it.

Prerequisites to Learning JavaScript

Although this book doesn't demand that you have a great deal of programming experience behind you, the more web pages you've created with HTML, the easier you will find it to understand how JavaScript interacts with the familiar elements you normally place in your pages. Occasionally, you will need to modify HTML tags to take advantage of scripting. If you are familiar with those tags already, the JavaScript enhancements will be simple to digest.

To learn JavaScript, you won't need to know server scripting or how to pass information from a form to a server. The focus here is on client-side scripting, which operates independently of the server after the JavaScript-enhanced HTML page is fully loaded into the browser. However, we strongly believe that a public web page should be operational in the absence of JavaScript, so any dynamic functionality that looks up results or modifies the content of a page should interact with a server-side script fundamentally. After that foundation is laid, we add JavaScript to make a page faster, easier, or more

fun. So although you don't need to know server-side scripting in order to learn JavaScript, for serious web work you should either learn a server-side scripting language such as PHP in addition to JavaScript or look for server-side programmers to complement your client-side scripting abilities.

The basic vocabulary of the current HTML standard should be part of your working knowledge. You should also be familiar with some of the latest document markup standards, such as XHTML and Cascading Style Sheets (CSS). Web searches for these terms will uncover numerous tutorials on the subjects.

If you've never programmed before

Don't be put off by the size of this book. JavaScript may not be the easiest language in the world to learn, but believe us, it's a far cry from having to learn a full programming language such as Java or C. Unlike developing a full-fledged monolithic application (such as the productivity programs you buy in stores), JavaScript lets you experiment by writing small snippets of program code to accomplish big things. The JavaScript interpreter built into every scriptable browser does a great deal of the technical work for you.

Programming, at its most basic level, consists of nothing more than writing a series of instructions for the computer to follow. We humans follow instructions all the time, even if we don't realize it. Traveling to a friend's house is a sequence of small instructions: Go three blocks that way; turn left here; turn right there. Amid these instructions are some decisions that we have to make: If the stoplight is red, then stop; if the light is green, then go; if the light is yellow, then floor it! (Just kidding.) Occasionally, we must repeat some operations several times (kind of like having to go around the block until a parking space opens up). A computer program not only contains the main sequence of steps, but it also anticipates what decisions or repetitions may be needed to accomplish the program's goal (such as how to handle the various states of a stoplight or what to do if someone just stole the parking spot you were aiming for).

The initial hurdle of learning to program is becoming comfortable with the way a programming language wants its words and numbers organized in these instructions. Such rules are called syntax, the same as in a living language. Computers aren't very forgiving if you don't communicate with them in the specific language they understand. When speaking to another human, you can flub a sentence's syntax and still have a good chance that the other person will understand you. Not so with computer programming languages. If the syntax isn't perfect (or at least within the language's range of knowledge), the computer has the brazenness to tell you that you have made a syntax error.

The best thing you can do is just accept the syntax errors you receive as learning experiences. Even experienced programmers make mistakes. Every syntax error you get — and every resolution of that error made by rewriting the wayward statement — adds to your knowledge of the language.

If you've done a little programming before

Programming experience in a procedural language, such as BASIC, may actually be a hindrance rather than a help to learning JavaScript. Although you may have an appreciation for precision in syntax, the overall concept of how a program fits into the world is probably radically different from JavaScript's role. Part of this has to do with the typical tasks a script performs (carrying out a very specific task in response to user action within a web page), but a large part also has to do with the nature of object-oriented programming.

If you've programmed in C before

In a typical procedural program, the programmer is responsible for everything that appears on the screen and everything that happens under the hood. When the program first runs, a great deal of code is dedicated to setting up the visual environment. Perhaps the screen contains several text entry fields or clickable buttons. To determine which button a user clicks, the program examines the coordinates of the click and compares those coordinates against a list of all button coordinates on the screen. Program execution then branches out to perform the instructions reserved for clicking in that space.

Object-oriented programming is almost the inverse of that process. A button is considered an object — something tangible. An object has properties, such as its label, size, alignment, and so on. An object may also contain a script. At the same time, the system software and browser, working together, can send a message to an object — depending on what the user does — to trigger the script. For example, if a user clicks in a text entry field, the system/browser tells the field that somebody has clicked there (that is, has set the focus to that field), giving the field the task of deciding what to do about it. That's where the script comes in. The script is connected to the field, and it contains the instructions that the field carries out after the user activates it. Another set of instructions may control what happens when the user types an entry and tabs or clicks out of the field, thereby changing the content of the field.

Some of the scripts you write may seem to be procedural in construction: They contain a simple list of instructions that are carried out in order. But when dealing with data from form elements, these instructions work with the object-based nature of JavaScript. The form is an object; each radio button or text field is an object as well. The script then acts on the properties of those objects to get some work done.

Making the transition from procedural to object-oriented programming may be the most difficult challenge for you. But when the concept clicks — a long, pensive walk might help — so many light bulbs will go on inside your head that you'll think you might glow in the dark. From then on, object orientation will seem to be the only sensible way to program.

By borrowing syntax from Java (which, in turn, is derived from C and C++), JavaScript shares many syntactical characteristics with C. Programmers familiar with C will feel right at home. Operator symbols, conditional structures, and repeat loops follow very much in the C tradition. You will be less concerned about data types in JavaScript than you are in C. In JavaScript, a variable is not restricted to any particular data type.

With so much of JavaScript's syntax familiar to you, you will be able to concentrate on document object model concepts, which may be entirely new to you. You will still need a good grounding in HTML to put your expertise to work in JavaScript.

If you've programmed in Java before

Despite the similarity in their names, the two languages share only surface aspects: loop and conditional constructions, C-like "dot" object references, curly braces for grouping statements, several keywords, and a few other attributes. Variable declarations, however, are quite different, because JavaScript is a loosely typed language. A variable can contain an integer value in one statement and a string in the next (though we're not saying that this is good style). What Java refers to as methods, JavaScript calls methods (when associated with a predefined object) or functions (for scripter-defined actions). JavaScript methods and functions may return values of any type without having to state the data type ahead of time.

Perhaps the most important Java concepts to suppress when writing JavaScript are the object-oriented notions of classes, inheritance, instantiation, and message passing. These aspects are simply non-issues when scripting. At the same time, however, JavaScript's designers knew that you'd have some hard-to-break habits. For example, although JavaScript does not require a semicolon at the end of each statement line, if you type one in your JavaScript source code, the JavaScript interpreter won't balk.

If you've written scripts (or macros) before

Experience with writing scripts in other authoring tools or macros in productivity programs is helpful for grasping a number of JavaScript's concepts. Perhaps the most important concept is the idea of combining a handful of statements to perform a specific task on some data. For example, you can write a macro in Microsoft Excel that performs a data transformation on daily figures that come in from a corporate financial report on another computer. The macro is built into the Macro menu, and you run it by choosing that menu item whenever a new set of figures arrives.

Some modern programming environments, such as Visual Basic, resemble scripting environments in some ways. They present the programmer with an interface builder, which does most of the work of displaying screen objects with which the user will interact. A big part of the programmer's job is to write little bits of code that are executed when a user interacts with those objects. A great deal of the scripting you will do with JavaScript matches that pattern exactly. In fact, those environments resemble the scriptable browser environment in another way: They provide a finite set of predefined objects that have fixed sets of properties and behaviors. This predictability makes learning the entire environment and planning an application easier to accomplish.

Formatting and Naming Conventions

The script listings and words in this book are presented in a `monospace font` to set them apart from the rest of the text. Because of restrictions in page width, lines of script listings may, from time to time, break unnaturally. In such cases, the remainder of the script appears in the following line, flush with the left margin of the listing, just as it would appear in a text editor with word wrapping turned on. If these line breaks cause you problems when you type a script listing into a document yourself, we encourage you to access the corresponding listing on the CD-ROM to see how it should look when you type it.

As soon as you reach Part III of this book, you won't likely go for more than a page before reading about an object model or language feature that requires a specific minimum version of one browser or another. To make it easier to spot in the text when a particular browser and browser version is required, most browser references consist of an abbreviation and a version number. For example, WinIE5 means Internet Explorer 5 for Windows; NN4 means Netscape Navigator 4 for any operating system; Moz stands for the modern Mozilla browser (from which Firefox, Netscape 6 or later, and Camino are derived); and Safari is Apple's own browser for Mac OS X. If a feature is introduced with a particular version of browser and is supported in subsequent versions, a plus symbol (+) follows the number. For example, a feature marked WinIE5.5+ indicates that Internet Explorer 5.5 for Windows is required at a minimum, but the feature is also available in WinIE8 and probably future WinIE versions. If a feature was implemented in the first release of a modern browser, a plus symbol immediately follows the browser family name, such as Moz+ for all Mozilla-based browsers. Occasionally, a feature or some highlighted behavior applies to only one browser. For example, a feature marked

NN4 means that it works only in Netscape Navigator 4.x. A minus sign (e.g., WinIE-) means that the browser does not support the item being discussed.

The format of HTML markup in this edition follows HTML5 coding conventions, but also adheres to many XHTML standards such as all-lowercase tag and attribute names.

Note **Tip**

Caution **Cross Reference**

Note, Tip, Caution, and Cross-Reference icons occasionally appear in the book to flag important points or suggest where to find more information.

JavaScript® Bible

Seventh Edition

Part I

Getting Started with JavaScript

IN THIS PART

Chapter 1
JavaScript's Role in the World Wide Web and Beyond

Chapter 2
Developing a Scripting Strategy

Chapter 3
Selecting and Using Your Tools

Chapter 4
JavaScript Essentials

CHAPTER 1

JavaScript's Role in the World Wide Web and Beyond

IN THIS CHAPTER

How JavaScript blends with other web-authoring technologies

The history of JavaScript

What kinds of jobs you should and should not entrust to JavaScript

Many of the technologies that make the World Wide Web possible have far exceeded their original goals. Envisioned at the outset as a medium for publishing static text and image content across a network, the Web is forever being probed, pushed, and pulled by content authors. By taking for granted so much of the "dirty work" of conveying the bits between server and client computers, content developers and programmers dream of exploiting that connection to generate new user experiences and practical applications. It's not uncommon for a developer community to take ownership of a technology and mold it to do new and exciting things. But with so many web technologies — especially browser programming with JavaScript — within reach of everyday folks, we have witnessed an unprecedented explosion in turning the World Wide Web from a bland publishing medium into a highly interactive, operating system–agnostic authoring platform.

The JavaScript language, working in tandem with related browser features, is a web-enhancing technology. When employed on the client computer, the language can help turn a static page of content into an engaging, interactive, and intelligent experience. Applications can be as subtle as welcoming a site's visitor with the greeting "Good morning!" when it is morning in the client computer's time zone — even though it is dinnertime where the server is located. Or, applications can be much more obvious, such as delivering the content of a slide show in a one-page download while JavaScript controls the sequence of hiding, showing, and "flying slide" transitions as we navigate through the presentation.

Of course, JavaScript is not the only technology that can give life to drab web content. Therefore, it is important to understand where JavaScript fits within the array of standards, tools, and other technologies at your disposal. The alternative technologies described in this chapter are HTML, Cascading Style Sheets (CSS), server programs, and plug-ins. In most cases, JavaScript can work side by side with these other technologies, even though the hype can make them sound like one-stop shopping places for all your interactive needs. (That's rarely the case.) Finally, you learn about the origins of JavaScript and what role it plays in today's advanced web browsers.

Competing for Web Traffic

Web page publishers revel in logging as many visits to their sites as possible. Regardless of the questionable accuracy of web page hit counts, a site consistently logging 10,000 dubious hits per week is clearly far more popular than one with 1,000 dubious hits per week. Even if the precise number is unknown, relative popularity is a valuable measure. Another useful number is how many links from outside pages lead to a site. A popular site will have many other sites pointing to it — a key to earning high visibility in web searches.

Encouraging people to visit a site frequently is the Holy Grail of web publishing. Competition for viewers is enormous. Not only is the Web like a 50 million–channel television, but also, the Web competes for viewers' attention with all kinds of computer-generated information. That includes anything that appears onscreen as interactive multimedia.

Users of entertainment programs, multimedia encyclopedias, and other colorful, engaging, and mouse-finger-numbing actions are accustomed to high-quality presentations. Frequently, these programs sport first-rate graphics, animation, live-action video, and synchronized sound. By contrast, the lowest-common-denominator web page has little in the way of razzle-dazzle. Even with the help of Dynamic HTML and style sheets, the layout of pictures and text is highly constrained compared with the kinds of desktop publishing documents you see all the time. Regardless of the quality of its content, an unscripted, vanilla HTML document is flat. At best, interaction is limited to whatever navigation the author offers in the way of hypertext links or forms whose filled-in content magically disappears into the web site's server.

Other Web Technologies

With so many ways to spice up web sites and pages, you can count on competitors for your site's visitors to do their darnedest to make their sites more engaging than yours. Unless you are the sole purveyor of information that is in high demand, you continually must devise ways to keep your visitors coming back and entice new ones. If you design for an intranet, your competition is the drive for improved productivity by colleagues who use the internal web sites for getting their jobs done.

These are all excellent reasons why you should care about using one or more web technologies to raise your pages above the noise. Let's look at the major technologies you should know about.

Figure 1-1 illustrates the components that make up a typical dynamic web site. The core is a dialog between the server (the web host) and the client (the browser); the client requests data and the server responds. The simplest model would consist of just the server, a document, and a browser, but in practice web sites use the other components shown here, and more.

The process begins when the browser requests a page from the server. The server delivers static HTML pages directly from its hard drive; dynamic pages are generated on-the-fly by scripts executed in a language interpreter, such as PHP, but likewise delivered by the server to the client, usually in the form of HTML markup. For example, a server-side database can store the items of a catalog, and a PHP script can look up those data records and mark them up as HTML when requested by the browser.

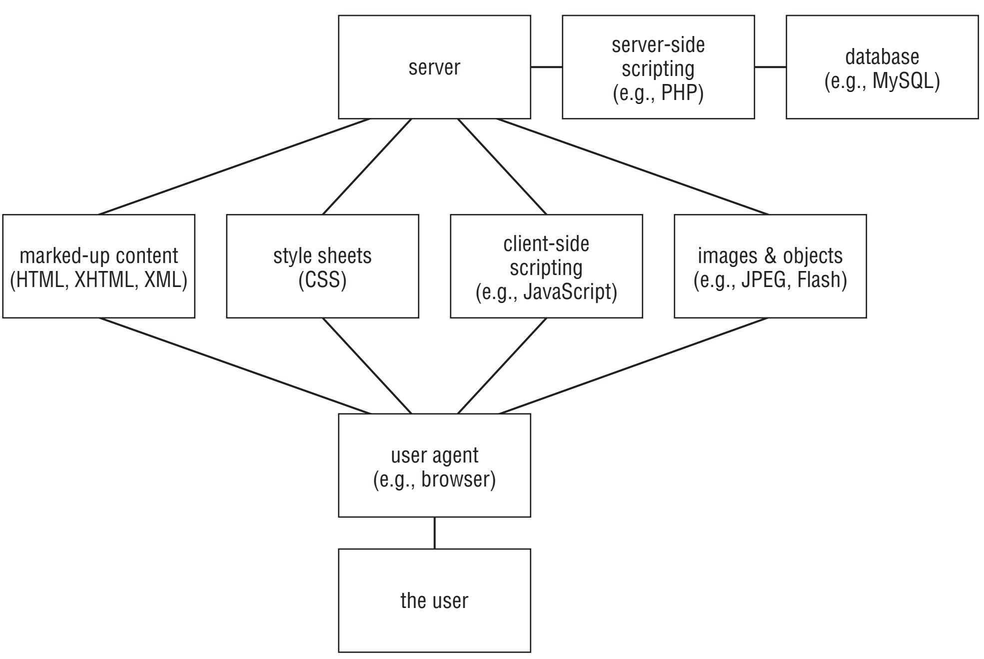
The downloaded page may contain the addresses of other components: style sheets, JavaScript files, images, and other assets. The browser requests each of these, in turn, from the server, combining them into the final rendering of the page.

A JavaScript program is inert — just another hunk of downloaded bits — until it's received by the browser, which validates its syntax and compiles it, ready for execution when called upon by the

HTML page or the human user. It's then part of the web page and confined to the client side of the sever-client dialog. JavaScript can make requests of servers, as we'll see later, but it cannot directly access anything outside of the page it's part of and the browser that's running it.

FIGURE 1-1

The components of a typical dynamic web site.



Hypertext Markup Language (HTML and XHTML)

As an outgrowth of SGML (Standard Generalized Markup Language), HTML brings structure to the content of a page. This structure gives us handles on the content in several important ways:

- Markup transforms a sea of undifferentiated text into discrete parts, such as headlines, paragraphs, lists, data tables, images, and input controls. Structure augments the meaning of the content by establishing relationships between different parts: between headline and subhead; between items in a list; between divisions of the page such as header, footer, and content columns. The semantics of a page are important to every reader: visual browsers for the sighted, vocal and Braille browsers for the visually impaired, search engines, and other software that parses the page, looking for a comprehensible structure.
- The browser transforms some marked-up structures into objects with particular behaviors. An image spills out rows of pixels in visually meaningful patterns. A form control accepts input from the user, while a button submits those form control values to the server. A hyperlink can load a new page into the browser window. These different types of objects would not be possible without some form of markup.

- The way a page is presented on screen, in Braille, or in speech, is determined by a style sheet that points to elements on the page and assigns appearance, emphasis, and other attributes. Every browser comes with a default style sheet that makes headlines, body text, hyperlinks, and form controls look a particular way. As web developers, we create our own style sheets to override the browser defaults and make our pages look the way we want. A style sheet tells the browser how to render the HTML page by referring to the document's markup elements and their attributes.
- Most importantly for the topic at hand, HTML markup gives JavaScript ways to locate and operate on portions of a page. A script can collect all the images in a gallery, or jump to a particular paragraph, because of the way the document is marked up.

Clearly, HTML markup and the way we handle it are critical to a document's structure and meaning, its presentation, and its successful interaction with JavaScript. While all the details of how best to use HTML are outside the scope of this book, it's clear that you'll want to hone your markup skills while you're learning JavaScript. Even the best script in the world can fail if launched on faulty, sloppy, or unplanned markup.

In the early years of web development, back before the turn of the century, in that medieval era now known as "the 90s," web designers didn't have much consciousness about markup semantics and accessibility. Pages were marked up in any way that would produce a desired visual presentation, relying on the default styling of the browser and without regard for the sensibility of the markup. A developer might choose an `h4` tag simply in order to produce a certain size and style of font, regardless of the document's outline structure, or use data table markup for non-tabular content, simply to force the page layout to align. Required spaces and faux spacer images were mashed into the real content merely to tweak the appearance of the page: how things *looked* took complete precedence over what things *meant*. Fortunately, today that perspective seems quaintly old-fashioned, but unfortunately, there are still thousands of people producing pages that way and millions of pages left over from that dark era, their finger bones clutching at our sleeves and urging us to join them in their morbid pastime. It is one goal of this book to encourage you to code like a modern, living mammal, and not like a fossil.

Relegating HTML to the category of a tagging language does disservice not only to the effort that goes into fashioning a first-rate web page, but also to the way users interact with the pages. To our way of thinking, any collection of commands and other syntax that directs the way users interact with digital information is *programming*. With HTML, a web-page author controls the user experience with the content just as the engineers who program Microsoft Excel craft the way users interact with spreadsheet content and functions.

Version 4.0 and later of the published HTML standards endeavor to define the purpose of HTML as assigning context to content, leaving the appearance to a separate standard for style sheets. In other words, it's not HTML's role to signify that some text is italic but rather to signify *why* we might choose to italicize it. For example, you would tag a chunk of text that conveys emphasis (via the `<em>` tag) or to mark its purpose (`<span class="title">`) separately from the decision of how to format it with the style sheet. This separation between HTML markup and CSS presentation is an extremely powerful concept that makes the tweaking and redesign of web sites much faster than in the old days of inline styling and `<font>` tags.

XHTML is a more recent adaptation of HTML that adheres to stylistic conventions established by the XML (eXtensible Markup Language) standard. No new tags come with XHTML, but it reinforces the notion of tagging to denote a document's structure and content. While for several years XHTML was seen as the next stage of evolution of HTML toward the holy grail of a universal XML markup for documents, that promise has failed to deliver, in part because Microsoft, with its enormous share of the browser market, has consistently declined to accommodate XHTML served correctly as `application/xhtml+xml`. Nearly all XHTML documents

today are served as `text/html`, which means that browsers treat them as just more HTML "tag soup." Perhaps the only advantage gained by using XHTML markup is the stricter set of rules used by the W3C HTML Validator (`http://validator.w3.org`), which checks to make sure that all tags are closed in XHTML documents, but isn't so fussy with HTML with its looser definitions.

For convincing arguments against the use of XHTML served as `text/html`, read "Sending XHTML as text/html Considered Harmful," by Ian Hickson (`http://hixie.ch/advocacy/xhtml`) and "Beware of XHTML," by David Hammond (`http://www.webdevout.net/articles/beware-of-xhtml`).

More recently, HTML5 has been growing in the Petri dishes of the World Wide Web Consortium (W3C). While modern browsers have only just begun to implement some of its features, HTML5 is considered by many to offer a more promising future than the seemingly abandoned XHTML. HTML5 offers an enriched vocabulary of markup tags compared to version 4, the better to align the markup language with the uses to which HTML is put in the real world.

Because HTML 4.01 is the prevailing markup language for the web today, the HTML samples in this book are compatible with HTML 4.01, except for those that illustrate newer HTML5 elements such as `canvas`. With the proper `DOCTYPE`, the samples should validate as HTML5. They are easily convertible to XHTML1.0 with a few simple transforms: change the `DOCTYPE`, add `lang` attributes to the `HTML` element, and close empty elements such as `input` and `link` with `/>`. It's our hope that this will make the samples useful to folks heading up either HTML or XHTML paths, and also make the book more future-friendly as HTML5 comes into its own.

Cascading Style Sheets (CSS)

Specifying the look and feel and speech of a web page is the job of Cascading Style Sheets (CSS). Given a document's structure as spelled out by its HTML markup, a style sheet defines the layout, colors, fonts, voices, and other visual and aural characteristics to present the content. Applying a different set of CSS definitions to the same document can make it look and sound entirely different, even though the words and images are the same.

(CSS 2.1 is the version of the W3C style sheet specification most widely supported by today's user agents. Aural style sheets that let us assign voices and other sounds to the markup of a web page are a module of the CSS 3 specification and, at this writing, are supported only by Opera and the FireVox extension to Firefox, although other browsers are sure to follow.)

Mastery of the fine points of CSS takes time and experimentation, but the results are worth the effort. The days of using HTML tables and transparent "spacer" images to generate elaborate multicolumn layouts are very much on the wane. Every web developer should have a solid grounding in CSS.

The learning curve for CSS can be steep because of the inconsistent support for its many features from one browser to the next. You can make your life much easier by triggering *standards mode* in the browser, whereby it adheres more closely to the W3C CSS specification. We recommend triggering standards mode by using a correct `DOCTYPE` at the top of your markup, such as one of these:

HTML 4.01 Strict:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
   "http://www.w3.org/TR/REC-html40/strict.dtd">
```

HTML 5:

```
<!DOCTYPE html>
```

```
XHTML 1.0 Strict:

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/xhtml1-strict.dtd">
```

For a clear explanation of `DOCTYPE` switching, read Eric Meyer's essay "Picking a Rendering Mode" (`http://www.ericmeyeroncss.com/bonus/render-mode.html`).

Server-side programming

Web sites that rely on database access, or change their content very frequently, incorporate programming on the server that generates the HTML output for browsers and/or processes forms that site visitors fill out on the page. Even submissions from a simple login or search form ultimately trigger some server process that sends the results to your browser. Server programming takes on many guises, the names of which you may recognize from your surfing through web development sites. PHP, ASP, .NET, JSP, and ColdFusion are among the most popular. Associated programming languages include Perl, Python, Java, C++, C#, Visual Basic, and even server-side JavaScript in some environments.

Whatever language you use, the job definitely requires the web-page author to be in control of the server, including whatever *back-end* programs (such as databases) are needed to supply results or massage the information coming from the user. Even with the new, server-based web site design tools available, server scripting is often a task that a content-oriented HTML author will need to hand off to a more experienced programmer.

Client-side JavaScript is not a replacement for server-side scripting. Any web site that reacts dynamically to user actions or that saves data must be driven by server-side scripting, even if JavaScript is used in the browser to enhance the user's experience. The reasons are simple: First, JavaScript itself cannot write to files on the server. It can help the user make choices and prepare data for upload, but after that it can only hand off data to a sever-side script for database updating. Second, not all user agents run JavaScript. Screen readers, mobile devices, search engines, and browsers installed in certain corporate contexts are among those that don't invoke JavaScript, or that receive web pages with the JavaScript stripped out. Therefore, your web site should be fully functional with JavaScript turned off. Use JavaScript to make the browsing experience faster, cooler, or more fun when it's present, but don't let your site be broken if JavaScript isn't running.

As powerful and useful as server-side scripting can be, its speed of interaction with the user is limited by the speed of the Internet connection between server and user agent. Obviously, any process that results in updating data on the server must include some client-server dialog, but there are many aspects of user interaction that, with the help of JavaScript, can take place entirely within the browser — form validation and drag-&-drop are two examples — then update the server when the response-sensitive process is complete.

One way that server programming and browser scripting work together is with what has become known as *Ajax* — Asynchronous JavaScript and XML. The "asynchronous" part runs in the browser, requesting XML data from, or posting data to, the server-side script entirely in the background. XML data returned by the server can then be examined by JavaScript in the browser to update portions of the web page. That's how many popular web-based email user interfaces work, as well as the draggable satellite-photo closeups of Google Maps (`http://maps.google.com`).

Working together, server programming and browser scripting can make beautiful applications together. You'll want to write in a server-side language such as PHP, or team up with someone who does, to lay the foundations for the JavaScript-enhanced pages you'll be creating.

Of helpers and plug-ins

In the early days of the World Wide Web, a browser needed to present only a few kinds of data before a user's eyes. The power to render text (tagged with HTML) and images (in popular formats such as GIF and JPEG) was built into browsers intended for desktop operating systems. Not wanting to be limited by those data types, developers worked hard to extend browsers so that data in other formats could be rendered on the client computer. It was unlikely, however, that a browser would ever be built that could download and render, say, any of several sound-file formats.

One way to solve the problem was to allow the browser, upon recognizing an incoming file of a particular type, to launch a separate application on the client machine to render the content. As long as this helper application was installed on the client computer (and the association with the helper program was set in the browser's preferences), the browser would launch the program and send the incoming file to that program. Thus, you might have one helper application for a MIDI sound file and another for an animation file.

Beginning with Netscape Navigator 2 in early 1996, software *plug-ins* for browsers enabled developers to extend the capabilities of the browser without having to modify the browser. Unlike a helper application, a plug-in can enable external content to blend into the document seamlessly.

The most common plug-ins are those that facilitate the playback of audio and video from the server. Audio may include music tracks that play in the background while visiting a page, or live (streaming) audio, similar to a radio station. Video and animation can operate in a space on the page when played through a plug-in that knows how to process such data.

Today's browsers tend to ship with plug-ins that decode the most common sound-file types. Developers of plug-ins for Internet Explorer for the Windows operating system commonly implement plug-ins as ActiveX controls — a distinction that is important to the underpinnings of the operating system, but not to the user.

Plug-ins and helpers are valuable for more than just audio and video playback. A popular helper application is *Adobe Acrobat Reader,* which displays Acrobat files that are formatted just as though they were being printed. But for interactivity, developers today frequently rely on the *Flash* plug-in by Macromedia (now owned by Adobe). Created using the Flash authoring environment, a Flash application can have active clickable areas and draggable elements, animation, and embedded video. Some authors simulate artistic video games and animated stories in Flash. A browser equipped with the Flash plug-in displays the content in a rectangular area embedded within the browser page. A variant of JavaScript called *ActionScript* enables Flash to interact with the user and components of the HTML page. Like JavaScript, ActionScript gets access to outside resources by making data-read and -write requests of the server. YouTube.com is a popular example of a web site with richly integrated Flash.

One potential downside for authoring interactive content in Flash or similar environments is that if the user does not have the correct plug-in version installed, it can take some time to download the plug-in (if the user even wants to bother). Moreover, once the plug-in is installed, highly graphic and interactive content can take longer to download to the client (especially on a dial-up connection) than some users are willing to wait. This is one of those situations in which you must balance your creative palette with the user's desire for your interactive content.

Another client-side technology — the Java applet — was popular for a while in the late 1990s, but has fallen out of favor for a variety of reasons (some technical, some corporate-political). But this has not diminished the use of Java as a language for server and even cellular telephone programming, extending well beyond the scope of the language's founding company, Sun Microsystems.

JavaScript: A Language for All

Sun's Java language is derived from C and C++, but it is a distinct language. Its main audience is the experienced programmer. That leaves out many web-page authors. Java's preliminary specifications in 1995 were dismaying. How much more preferable would have been a language that casual programmers and scripters who were comfortable with authoring tools such as Apple's once-formidable HyperCard and Microsoft's Visual Basic could adopt quickly. As these accessible development platforms have shown, nonprofessional authors can dream up many creative applications, often for very specific tasks that no professional programmer would have the inclination to work on. Personal needs often drive development in the classroom, office, den, or garage. But Java was not going to be that kind of inclusive language.

Spirits lifted several months later, in November 1995, when we heard of a scripting language project brewing at Netscape Communications, Inc. Born under the name *LiveScript,* this language was developed in parallel with a new version of Netscape's web server software. The language was to serve two purposes with the same syntax. One purpose was as a scripting language that web server administrators could use to manage the server and connect its pages to other services, such as back-end databases and search engines for users looking up information. Extending the "Live" brand name further, Netscape assigned the name *LiveWire* to the database connectivity usage of LiveScript on the server.

On the client side — in HTML documents — authors could employ scripts written in this new language to enhance web pages in a number of ways. For example, an author could use LiveScript to make sure that the user had filled in a required text field with an e-mail address or credit card number. Instead of forcing the server or database to do the data validation (requiring data exchanges between the client browser and the server), the user's computer handles all the calculation work — putting some of that otherwise-wasted computing horsepower to work. In essence, LiveScript could provide HTML-level interaction for the user.

LiveScript becomes JavaScript

In early December 1995, just prior to the formal release of Navigator 2, Netscape and Sun Microsystems jointly announced that the scripting language thereafter would be known as JavaScript. Though Netscape had several good marketing reasons for adopting this name, the changeover may have contributed more confusion to both the Java programming world and HTML scripting world than anyone expected.

Before the announcement, the language was already related to Java in some ways. Many of the basic syntax elements of the scripting language were reminiscent of the Java style. However, for client-side scripting, the language was intended for very different purposes than Java — essentially to function as a programming language integrated into HTML documents rather than as a language for writing applets that occupy a fixed rectangular area on the page (and that are oblivious to anything else on the page). Instead of Java's full-blown programming language vocabulary (and conceptually more difficult-to-learn object-oriented approach), JavaScript had a small vocabulary and a more easily digestible programming model.

The true difficulty, it turned out, was making the distinction between Java and JavaScript clear to the world. Many computer journalists made major blunders when they said or implied that JavaScript provided a simpler way of building Java applets. To this day, some new programmers believe JavaScript is synonymous with the Java language: They post Java queries to JavaScript-specific Internet newsgroups and mailing lists.

The fact remains that client-side Java and JavaScript are more different than they are similar. The two languages employ entirely different interpreter engines to execute their code.

Enter Microsoft and others

Although the JavaScript language originated at Netscape, Microsoft acknowledged the potential power and popularity of the language by implementing it (under the JScript name) in Internet Explorer 3. Even if Microsoft might prefer that the world use the VBScript (Visual Basic Script) language that it provides in the Windows versions of IE, the fact that JavaScript is available on more browsers and operating systems makes it the client-side scripter's choice for anyone who must design for a broad range of users.

With scripting firmly entrenched in the mainstream browsers from Microsoft and Netscape, newer browser makers automatically provided support for JavaScript. Therefore, you can count on fundamental scripting services in browsers such as Opera or the Apple Safari browser (the latter built upon an Open Source browser called KHTML). Not that all browsers work the same way in every detail — a significant challenge for client-side scripting that is addressed throughout this book.

JavaScript versions

The JavaScript language has its own numbering system, which is completely independent of the version numbers assigned to browsers. The Mozilla Foundation, successor to the Netscape browser development group that created the language, continues its role as the driving force behind the JavaScript version numbering system.

The first version, logically enough, was JavaScript 1.0. This was the version implemented in Navigator 2, and the first release of Internet Explorer 3. As the language evolved with succeeding browser versions, the JavaScript version number incremented in small steps. JavaScript 1.2 is the version that has been the most long lived and stable, currently supported by Internet Explorer 7. Mozilla-based browsers and others have inched forward with some new features in JavaScript 1.5 (Mozilla 1.0 and Safari), JavaScript 1.6 (Mozilla 1.8 browsers), and JavaScript 1.7 (Mozilla 1.8.1 and later).

Each successive generation of JavaScript employs additional language features. For example, in JavaScript 1.0, arrays were not developed fully, causing scripted arrays not to track the number of items in the array. JavaScript 1.1 filled that hole by providing a constructor function for generating arrays and an inherent `length` property for any generated array.

The JavaScript version implemented in a browser is not always a good predictor of core language features available for that browser. For example, although JavaScript 1.2 (as implemented by Netscape in Netscape Navigator 4) included broad support for regular expressions, not all of those features appeared in Microsoft's corresponding JScript implementation in Internet Explorer 4. By the same token, Microsoft implemented `try-catch` error handling in its JScript in Internet Explorer 5, but Netscape didn't include that feature until the Mozilla-based Netscape Navigator 6 implementation of JavaScript 1.5. Therefore, the language version number is an unreliable predictor in determining which language features are available for you to use.

Core language standard: ECMAScript

Although Netscape first developed the JavaScript language, Microsoft incorporated the language in Internet Explorer 3. Microsoft did not want to license the Java name from its trademark owner (Sun Microsystems), which is why the language became known in the Internet Explorer environment as

JScript. Except for some very esoteric exceptions and the pace of newly introduced features, the two languages are essentially identical. The levels of compatibility between browser brands for a comparable generation are remarkably high for the core language (unlike the vast disparities in object model implementations discussed in Chapter 25, "Document Object Model Essentials").

As mentioned, standards efforts have been under way to create industry-wide recommendations for browser makers to follow (to make developers' lives easier). The core language was among the first components to achieve standard status. Through the European standards body called ECMA, a formal standard for the language was agreed to and published. The first specification for the language, dubbed ECMAScript by the standards group, was roughly the same as JavaScript 1.1 in Netscape Navigator 3. The standard (ECMA-262) defines how various data types are treated, how operators work, what a particular data-specific syntax looks like, and other language characteristics. A newer version (called version 3) added many enhancements to the core language (version 2 was just version 1 with errata fixed).

The current version of the ECMAScript specification is known as ECMAScript, Fifth Edition, published online at `www.ecma-international.org`. To quote the ECMA, "The Fifth Edition codifies de facto interpretations of the language specification that have become common among browser implementations and adds support for new features that have emerged since the publication of the Third Edition. Such features include accessor properties, reflective creation and inspection of objects, program control of property attributes, additional array manipulation functions, support for the JSON object encoding format, and a strict mode that provides enhanced error checking and program security."

If you are a student of programming languages, you will find the document fascinating; if you simply want to script your pages, you might find the minutia mind-boggling.

All mainstream browser developers have pledged to make their browsers compliant with the ECMA standard. The vast majority of the ECMAScript standard has appeared in Navigator since version 3 and Internet Explorer since version 4, and as new features are added to the ECMA standard, they tend to find their way into newer browsers as well. The latest version of ECMAScript is version 5. The previous edition, the 3rd, has been supported in all mainstream browsers for the past few years.

Note

Even as ECMAScript, Fifth Edition was in the works, The Mozilla Foundation and Microsoft were implementing comparable versions of JavaScript 2.0 and JScript, respectively. An extension to ECMAScript called E4X (ECMAScript for XML) was finalized in late 2005 and is implemented in browsers based on Mozilla 1.8.1 or later (for example, Firefox 2.0). The Adobe ActionScript 3 language, which is used in the development of Flash animations, fully supports E4X. E4X is a significant addition to JavaScript because it makes XML (Extensible Markup Language) a native data type within the syntax of the language, making the processing of data in XML format much easier. XML is the data format used by many data exchange processes, including Ajax (see Chapter 39). ■

JavaScript: The Right Tool for the Right Job

Knowing how to match an authoring tool to a solution-building task is an important part of being a well-rounded web site author. A web designer who ignores JavaScript is akin to a plumber who bruises his knuckles by using pliers instead of the wrench from the bottom of the toolbox.

By the same token, JavaScript won't fulfill every dream. The more you understand about JavaScript's intentions and limitations, the more likely you will be to turn to it immediately when it is the proper tool. In particular, look to JavaScript for the following kinds of solutions:

- Getting your web page to respond or react directly to user interaction with form elements (input fields, text areas, buttons, radio buttons, checkboxes, selection lists) and hypertext links
- Distributing small collections of database-like information and providing a friendly interface to that data
- Controlling multiple-frame navigation, plug-ins, or Java applets based on user choices in the HTML document
- Preprocessing data on the client before submission to a server
- Changing content and styles in modern browsers dynamically and instantly, in response to user interaction
- Requesting files from the server, and making read and write requests of server-side scripts

At the same time, it is equally important to understand what JavaScript is *not* capable of doing. Scripters waste many hours looking for ways of carrying out tasks for which JavaScript was not designed. Most of the limitations are intentional, to protect visitors from invasions of privacy or unauthorized access to their desktop computers. Therefore, unless a visitor uses a modern browser and explicitly gives you permission to access protected parts of his or her computer, JavaScript cannot surreptitiously perform any of the following actions:

- Setting or retrieving the browser's preferences settings, main window appearance features, action buttons, and printing capability
- Launching an application on the client computer
- Reading or writing files or directories on the client computer (with one exception: cookies)
- Writing directly to files on the server
- Capturing live data streams from the server for retransmission
- Sending secret e-mails from web site visitors to you (although it can send data to a server-side script capable of sending email)

Web site authors are constantly seeking tools that will make their sites engaging (if not cool) with the least amount of effort. This is particularly true when the task of creating a web site is in the hands of people more comfortable with writing, graphic design, and page layout than with hard-core programming. Not every webmaster has legions of experienced programmers on hand to whip up some special, custom enhancement for the site. Neither does every web author have control over the web server that physically houses the collection of HTML and graphics files. JavaScript brings programming power within reach of anyone familiar with HTML, even when the server is a black box at the other end of a telephone line.

CHAPTER

2

Developing a Scripting Strategy

IN THIS CHAPTER

How leapfrogging browser developments help and hurt web developers

Separating the core JavaScript language from document objects

The importance of developing a cross-browser strategy

If you are starting to learn JavaScript at this point in the history of scriptable browsers, you have both a distinct advantage and a disadvantage. The advantage is that you have the wonderful capabilities of mature browser offerings from Microsoft, The Mozilla Foundation (under brand names such as Firefox, Netscape, and Camino), Apple, and others at your bidding. The disadvantage is that you have not experienced the painful history of authoring for older browser versions that were buggy and at times incompatible with one another due to a lack of standards. You have yet to learn the anguish of carefully devising a scripted application for the browser version you use, only to have site visitors send you voluminous email messages about how the page triggers all kinds of script errors when run on a different browser brand, generation, or operating system platform.

Welcome to the real world of scripting web pages with JavaScript. Several dynamics are at work to help make an author's life difficult if the audience for the application uses more than a single type of browser. This chapter introduces you to these challenges before you type your first word of JavaScript code. Our fear is that the subjects we raise may dissuade you from progressing further into JavaScript and its powers. But as developers ourselves — and some of us have been using JavaScript since the earliest days of its public prerelease availability — we dare not sugar-coat the issues facing scripters today. Instead, we want to make sure you have an appreciation for what lies ahead to assist you in learning the language. We believe that if you understand the big picture of the browser-scripting world as it stands today, you will find it easier to target JavaScript usage in your web application development and be successful at it.

Browser Leapfrog

Browser compatibility has been an issue for authors since the earliest days of the web gold rush — long before JavaScript. Despite the fact that browser developers and other interested parties voiced their opinions during formative stages of standards development, HTML authors could not produce a document that appeared

the same, pixel by pixel, on all client machines. It may have been one thing to establish a set of standard tags for defining heading levels and line breaks, but it was rare for the actual rendering of content inside those tags to look identical on different brands of browsers on different operating systems.

Then, as the competitive world heated up — and web browser development transformed itself from a volunteer undertaking into profit-seeking businesses — creative people defined new features and new tags that helped authors develop more flexible and interesting-looking pages. As happens a lot in any computer-related industry, the pace of commercial development easily surpassed the studied progress of standards. A browser maker would build a new HTML feature into a browser and only then propose that feature to the relevant standards body. Web authors were using these features (sometimes for prerelease browser versions) before the proposals were published for review.

When the deployment of content depends almost entirely on an interpretive engine on the client computer receiving the data — the HTML engine in a browser, for example — authors face an immediate problem. Unlike a stand-alone computer program that can extend and even invent functionality and have it run on everyone's computer (at least for a given operating system), web content providers must rely on the functionality built into the browser. This led to questions such as, "If not all browsers coming to my site support a particular HTML feature, then should I apply newfangled HTML features for visitors only at the bleeding edge?" and "If I do deploy the new features, what do I do for those with older browsers?"

Authors who developed pages in the earliest days of the Web wrestled with these questions for many HTML features that we today take for granted. Tables and frames come to mind. Eventually, the standards caught up with the proposed HTML extensions — but not without a lot of author woe along the way.

Despite the current dominance of the Microsoft Internet Explorer browser on the dominant Windows operating system, the number of browsers that people use is not shrinking. Several recent browsers, including the modern Netscape, Firefox, and Camino browsers, are based on an Open Source browser called Mozilla. The Macintosh operating system includes its own Apple-branded browser, Safari (released in 2003). And the independent Opera browser also has a home on some users' computers. All of these non-Microsoft browser makers obviously believe that they bring improvements to the world to justify their development — building better mousetraps, you might say.

Duck and Cover

Today's browser wars are fought on different battlegrounds than in the early days of the Web. The breadth and depth of established web standards have substantially fattened the browser applications — and the books developers read to exploit those standards for their content. On one hand, most developers clamor for deeper standards support in new browser versions. On the other hand, everyday users care little about standards. All they want is to have an enjoyable time finding the information they seek on the Web. Most users are slow to upgrade, holding out until their favorite sites start breaking in their ancient browsers.

Industry standards don't necessarily make the web developer's job any easier. For one thing, the standards are unevenly implemented across the latest browsers. Some browsers go further in their support than others. Then, there are occasional differences in interpretation of vague standards details. Sometimes the standards don't provide any guidance in areas that are vital to content developers. At times, we are left to the whims of browser makers who fill the gaps with proprietary features in the hope that those features will become de facto standards.

As happens in war, civilian casualties mount when the big guns start shooting. The browser battle lines shifted dramatically in only a few years. The huge market-share territory once under Netscape's command came under Microsoft's sway. More recently, however, concerns about privacy and security on the Windows platform have driven many users to seek less vulnerable browsers. Mozilla Firefox has so far been the biggest beneficiary in the search for alternatives. Although a fair amount of authoring common ground exists between the latest versions of today's browsers, uneven implementation of the newest features causes the biggest problems for authors wishing to deploy on all browsers. Trying to define the common denominator may be the toughest part of the authoring job.

Compatibility Issues Today

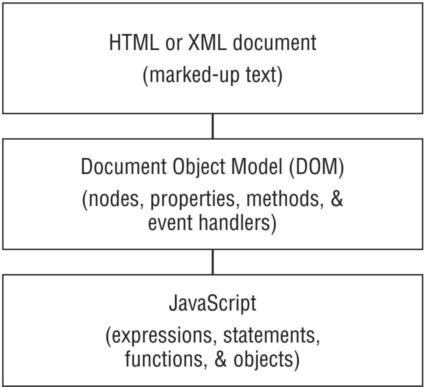
Allow us to describe the current status of the compatibility situation among the top three browser families: Microsoft Internet Explorer, browsers based on Mozilla, and Apple Safari. The discussion in the next few sections intentionally does not get into specific scripting technology very deeply; some of you may know very little about programming at this point. In many chapters throughout this book, we offer scripting suggestions to accommodate a variety of browsers.

Separating the core JavaScript language from document objects

Although early JavaScript authors initially treated client-side scripting as one environment that permitted the programming of page elements, the scene has changed as the browsers have matured. Today, a clear distinction exists between specifications for the core JavaScript language and for the elements you script in a document — for example, buttons and fields in a form (see Figure 2-1).

FIGURE 2-1

The document object model is the programming interface between the HTML or XML document and the JavaScript programming language.



On one level, this separation is a good thing. It means that one specification exists for basic programming concepts and syntax that could become the programming language in any number of other environments. You can think of the core language as basic wiring. When you know how electric

wires work, you can connect them to all kinds of electrical devices. Similarly, JavaScript today is used to wire together elements in an HTML document. Tomorrow, operating systems could use the core language to enable users to wire together desktop applications that need to exchange information automatically.

At the ends of today's JavaScript wires inside browsers are the elements on the page such as paragraphs (`p`), images (`img`), and input fields (`input`). In programming jargon, these items are known as *document objects*. By keeping the specifications for document objects separate from the wires that connect them, you can use other kinds of wires (other languages) to connect them. It's like designing telephones that can work with any kind of wire, including a type of wire that hasn't been invented yet. Today, the devices can work with copper wire or fiber-optic cable. You get a good picture of this separation in Internet Explorer, whose set of document objects can be scripted with either JavaScript or VBScript. The same objects can be connected with either of those two different sets of wiring.

The separation of core language from document objects enables each concept to have its own standards effort and development pace. But even with recommended standards for each factor, each browser maker is free to extend the standards. Furthermore, authors may have to expend more effort to devise one version of a page or script that plays on multiple browsers, unless the script adheres to a common denominator (or uses some other branching techniques to let each browser run its own way).

Core language standard

Keeping track of JavaScript language versions requires a brief history lesson. The first version of JavaScript (in Netscape Navigator 2) was version 1, although that numbering was not part of the language usage. JavaScript was JavaScript. Version numbering became an issue when Navigator 3 was released. The version of JavaScript associated with that Navigator version was JavaScript 1.1. The first appearance of the Navigator 4 generation increased the language version one more notch with JavaScript 1.2.

Microsoft's scripting effort contributes confusion for scripting newcomers. The first version of Internet Explorer to include scripting was Internet Explorer 3. The timing of Internet Explorer 3 was roughly coincidental to Navigator 3. But as scripters soon discovered, Microsoft's scripting effort was one generation behind. Microsoft did not license the JavaScript name. As a result, the company called its language JScript. Even so, the HTML tag attribute that lets you name the language of the script inside the tags could be either JScript or JavaScript for Internet Explorer. Internet Explorer 3 could understand a JavaScript script written for Navigator 2.

During this period of dominance by Navigator 3 and Internet Explorer 3, scripting newcomers were often confused because they expected the scripting languages to be the same. Unfortunately for the scripters, there were language features in JavaScript 1.1 that were not available in the older JavaScript version in Internet Explorer 3. Microsoft improved JavaScript in IE3 with an upgrade to the .dll file that gives IE its JavaScript syntax. However, it was hard to know which .dll is installed in any given visitor's IE3. The situation smoothed out for Internet Explorer 4. Its core language was essentially up to the level of JavaScript 1.2, as in early releases of Navigator 4. Almost all language features that were new in Navigator 4 were understood when you loaded the scripts into Internet Explorer 4. Microsoft still officially called the language JScript.

While all of this jockeying for JavaScript versions was happening, Netscape, Microsoft, and other concerned parties met to establish a core language standard. The standards body is a Switzerland-based organization originally called the European Computer Manufacturer's Association and now known simply as ECMA (commonly pronounced "ECK-ma"). In mid-1997, the first formal language specification was agreed on and published (ECMA-262). Due to licensing issues with the JavaScript name, the body created a new name for the language: ECMAScript.

With only minor and esoteric differences, this first version of ECMAScript was essentially the same as JavaScript 1.1, used in Navigator 3. Both Navigator 4 and Internet Explorer 4 officially supported the ECMAScript standard. Moreover, as happens so often when commerce meets standards bodies, both browsers went beyond the ECMAScript standard. Fortunately, the common denominator of this extended core language is broad, lessening authoring headaches on this front.

JavaScript version 1.3 was implemented in Netscape Navigator 4.06 through 4.7x. This language version is also the one supported in IE5, 5.5, and 6. A few new language features are incorporated in JavaScript 1.5, as implemented in Mozilla-based browsers (including Navigator 6 and later). A few more core language features were added to JavaScript 1.6, first implemented in Mozilla 1.8 (Firefox 1.5, and yet more to JavaScript 1.8 in Firefox 3).

In practice, so many browsers in use today support all but a few leading-edge features of the Mozilla browsers, that JavaScript version numbers are mostly irrelevant. Other compatibility issues with older browsers will likely get in your way before core language problems do. The time has come to forget about elaborate workarounds for the inadequacies of the oldest browsers.

Document object model

If prevalent browsers have been close to one another in core JavaScript language compatibility, nothing could be further from the truth when it comes to the document objects. Internet Explorer 3 based its document object model (DOM) on that of Netscape Navigator 2, the same browser level it used as a model for the core language. When Netscape added a couple of new objects to the model in Navigator 3, the addition caused further headaches for neophyte scripters who expected those objects to appear in Internet Explorer 3. Probably the most commonly missed object in Internet Explorer 3 was the image object, which lets scripts swap the image when a user rolls the cursor atop a graphic — *mouse rollovers,* they're commonly called.

In the level 4 browsers, however, Internet Explorer's DOM jumped way ahead of the object model that Netscape implemented in Navigator 4. The two most revolutionary aspects of IE4 were the ability to script virtually every element in an HTML document and the instant reflow of a page when the content changed. This opened the way for HTML content to be genuinely dynamic without requiring the browser to fetch a rearranged page from the server. NN4 implemented only a small portion of this dynamism, without exposing all elements to scripts or reflowing the page. It introduced a proprietary layering concept that was abandoned at the end of the Navigator 4.x lifetime. Inline content could not change in NN4 as it could in IE4. Suffice it to say that IE4 was an enviable implementation.

At the same time, a DOM standard was being negotiated under the auspices of the World Wide Web Consortium (W3C). The hope among scripters was that after a standard was in place, it would be easier to develop dynamic content for all browsers that supported the standard. The resulting standard — the W3C DOM — formalized the notion of being able to script every element on the page, as in IE4. But it also invented an entirely new object syntax that no browser had used. The race was on for browsers to support the W3C DOM standards.

An arm of the Netscape company called Mozilla.org was formed to create an all-new browser dedicated to supporting industry standards. The engine for the Mozilla browser became the basis for the all-new Navigator 6. It incorporated all of the W3C DOM Level 1 and a good chunk of Level 2. Mozilla 1.01 became the basis for the Netscape 7 browser, whereas Netscape 7.1 was built on the Mozilla 1.4 generation. In the summer of 2003, Netscape's parent company, AOL Time Warner, decided to end further Netscape-branded browser development. The work on the underlying Mozilla browser, however, continues under an independent organization called The Mozilla Foundation.

Mozilla-based browsers, and others using the same engine (such as Firefox and Camino), continue to be upgraded and released to the public. The Mozilla engine arguably offers the most in-depth support for the W3C DOM standards.

Even though Microsoft participated in W3C DOM standards development, IE5 and 5.5 implemented only some of the W3C DOM standard — in some cases, just enough to allow simple cross-browser scripting that adheres to the standard. Microsoft further filled out W3C DOM support in IE6, but chose to omit several important parts. Despite the long time gap between releases of IE6 and IE7, the latter included no additional W3C DOM support — much to the chagrin of web developers. IE8 finally did some significant catching up with the W3C DOM specification, with a switch to put IE8 back into IE7-compatibility mode for support of legacy scripts.

The Apple Safari browser has raced forward in its comparatively short life to offer substantial W3C DOM support. This is especially true of version 2, which was first released as part of Mac OS X version 10.4.

With this seemingly tortuous history of DOM development and browser support leading to the present day, you may wonder how anyone can approach DOM scripting with hope of success. Yet you'd be amazed by how much you can accomplish with today's browsers. You'll certainly encounter compatibility issues along the way, but this book will guide you through the most common problems and equip you to tackle others.

Laying a good foundation with markup

When the HTML markup on a page conforms to uniform public standards, browsers from different manufacturers tend to treat it similarly. However, in order to support the millions of web pages from past years, browsers also take a stab at guessing the author's intent when the markup differs from the specs. Because this sort of "guessing," for the most part, isn't prescribed in the HTML standards, the various browser manufacturers have come up with their own very different solutions. Therefore, you will get more consistent results and build more bullet-proof sites if your markup is easily parsable and error-free.

We encourage you produce markup that conforms to the W3C specification — for example, the HTML 4.01 spec at `http://www.w3.org/TR/html4/`. It's a good idea to rely on tools such as the W3C HTML Validator (`http://validator.w3.org/`) to check your markup against the standard. Beyond that, be consistent in your own work even when the spec is loose — for example, we suggest that you close all tags such as `td` and `li` even if the spec and the Validator don't require it. Examine and mimic the kind of markup produced by JavaScript using DOM methods, and you'll be doing fine.

Cascading Style Sheets

Navigator 4 and Internet Explorer 4 were the first browsers to claim compatibility with a W3C recommendation called *Cascading Style Sheets Level 1 (CSS1)*. This specification provided designers an organized way to customize the look and feel of a document (and thus minimized the HTML in each tag). As implementations go, NN4's had a lot of rough edges, especially when trying to mix style sheets and tables. But IE4 was no angel, either, especially when one compared the results of style sheet assignments as rendered in the Windows and Macintosh versions of the browser (developed by two separate teams).

CSS Level 2 adds more style functionality to the standard; IE6, Mozilla-based browsers, and Safari support a good deal of Level 2 (albeit unevenly) with the latest versions, such as Mozilla 1.8+ and

Safari 2+, and they are beginning to support CSS Level 3 features. Rendering of styled content is more harmonious among browsers, largely thanks to guidelines about how styles should render. Complex layouts, however, still need careful tweaking from time to time because of different interpretations of the standard.

JavaScript plays a role in style sheets in IE4+, Mozilla, and Safari because those browsers' object models permit dynamic modification to styles associated with any content on the page. Style sheet information is part of the object model and therefore is accessible and modifiable from JavaScript.

Standards compatibility modes (DOCTYPE switching)

Both Microsoft and Netscape/Mozilla discovered that they had, over time, implemented CSS features in ways that ultimately differed from the published standards that came later (usually after much wrangling among working-group members). To compensate for these differences and make a clean break to be compatible with the standards, the major browser makers decided to let the page author's choice of `<!DOCTYPE>` header element details determine whether the document was designed to follow the old way (sometimes called *quirks mode*) or the standards-compatible way. The tactic, known informally as *DOCTYPE switching,* is implemented in Internet Explorer 6 and later, Internet Explorer 5 for the Mac, and all Mozilla-based browsers.

Although most of the differences between the two modes are small, there are some significant differences between them in Internet Explorer 6 and later, particularly when styles or DHTML scripts rely on elements designed with borders, margins, and padding. Microsoft's original box model measured the dimensions of elements in a way that differed from the eventual CSS standard.

To place the affected browsers in CSS standards-compatible mode, you should include a `<!DOCTYPE>` element at the top of every document that specifies one of the following statements:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN"
        "http://www.w3.org/TR/REC-html40/loose.dtd">

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Frameset//EN"
        "http://www.w3.org/TR/REC-html40/frameset.dtd">

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0//EN"
        "http://www.w3.org/TR/REC-html40/strict.dtd">

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
        "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Frameset//EN"
        "http://www.w3.org/TR/xhtml1/DTD/xhtml1-frameset.dtd">

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
        "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
        "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
```

Be aware, however, that older versions of Internet Explorer for Windows, such as Internet Explorer 5 or Internet Explorer 5.5, are ignorant of the standards-compatible mode and

will use the old Microsoft quirks mode regardless of your `<!DOCTYPE>` setting. Still, using the standards-compatible mode DOCTYPE is more likely to force your content and style sheets to render similarly across the latest browsers.

Dynamic HTML and positioning

Perhaps the biggest improvements to the inner workings of the Level 4 browsers from both Netscape and Microsoft revolved around a concept called *Dynamic HTML (DHTML)*. The ultimate goal of DHTML was to enable scripts in documents to control content, content position, and content appearance in response to user actions. To that end, the W3C organization developed another standard for the precise positioning of HTML elements on a page, as an extension of the CSS standards effort. The CSS-Positioning recommendation was later incorporated into the CSS standard, and both are now part of CSS Level 2. With positioning, you can define an exact location on the page where an element should appear, whether the item should be visible, and what stacking order it should take among all the items that might overlap it.

IE4+ adheres to the positioning-standard syntax and makes positionable items subject to script control. Navigator 4 followed the standard from a conceptual point of view, but it implemented an alternative methodology involving an entirely new, and eventually unsanctioned, tag for layers. Such positionable items were scriptable in Navigator 4 as well, although a lot of the script syntax differed from that used in Internet Explorer 4. Fortunately for DHTML authors, Mozilla, through its adherence to the CSS standard, is more syntactically in line with the DHTML style properties employed in IE4+.

Of more interest these days is the ability to modify the inline content of a web page without reloading the entire page. Fundamental standards from the W3C DOM Level 1 are supported by a wide range of browsers, including IE5+, Mozilla, Safari, and Opera. You can accomplish quite a lot using the same basic syntax across all of these browsers. Some challenges remain, however, as you'll see throughout this book.

Developing a Scripting Strategy

How do you create web pages that work well across the board? Browsers representing the latest generation contain a hodgepodge of standards and proprietary extensions. Even if you try to script to a common denominator among today's browsers, your code probably won't take into account the earlier versions of both the JavaScript core language and the browser DOMs.

The true challenge for authors is determining the audience for which scripted pages are intended. Each new browser generation not only brings with it new and exciting features you are probably eager to employ in your pages, but also adds to the fragmentation of the audience visiting a publicly accessible page. It can take months for the latest upgrade of a browser to supplant the previous version among its users, and some people seldom or never upgrade unless the new browser comes via a new computer or operating system upgrade. As a result, web developers are writing for an incredibly mixed audience of browser makes and models, including some of the latest mobile devices that don't support JavaScript. Add to that all the user agents that aren't visual browsers, such as screen readers and search engines. How to cope?

Graceful degradation and progressive enhancement

At this stage in the history of scriptable browsers, most web surfers arrive at our sites with browsers equipped with support for at least simple W3C DOM and DHTML capabilities. But "most" isn't everyone by a long shot, so it is our obligation to apply scripting in the additive, or value-added, manner

known as *progressive enhancement*. By this we mean that your pages should convey their primary information to nonscriptable browsers designed for users with vision or motor-skill impairments, as well as less-feature-rich browsers built into the latest cellular phones. On top of that, your scripting efforts can give visitors with recent scriptable browsers a more enjoyable experience — better interactivity, faster performance, and a more engaging presentation. You will not only be contributing to the state of the art, but also carrying on the original vision of scripting in the browser.

In an early stage of the Web's evolution, many web site developers included the fancy features of the latest browser releases without much concern for folks who hadn't upgraded their browsers or switched from brand X to brand Y. This meant that a lot of new web sites were cosmetically — or, worse, functionally — broken for a lot of visitors. It was common to see banners explaining that "this website is viewed best in" such-and-such a browser, and it was left to the visitor to put up with the mess, or to install a new browser.

Over the years, web site producers have acquired two exciting new features that have changed our approach to browser support: the online business model and the social conscience. Excluding potential web site visitors from both commercial and non-profit sites is recognized as fiscally unwise. Excluding users of assistive technology is regarded as unjust — not to mention actionable in an increasing number of jurisdictions. As web developers, our concern isn't just a few obsolete browsers that can't run JavaScript. There are many other user agents that don't or can't support scripting, including search engines, some mobile devices, and some screen-readers and other assistive UAs. A lot of people simply turn off JavaScript support in their browsers. Some corporate firewalls strip JavaScript out of incoming web pages so that even entirely JavaScript-capable browsers within the organization do not see it or run it. Web stats are all over the map, but some estimates of the percentage of web site visitors who run without JavaScript for all of these reasons are 10% or higher. Even if it were only 1%, it would be significant. Imagine that you're holding the front door for customers at a physical storefront, and you choose to slam the door in the face of one out of every hundred people who approach. How long would you keep your job?

Graceful degradation is an approach to system design that provides fall-backs in case of failure. In web design, this means providing a lower level of appearance or functionality if the user agent can't handle any of the advanced technology that makes up the page. The problem with graceful degradation is the way it's been incorporated into site production. Developers who cut their teeth in the bad old days, and those who were trained by them, continued to build sites primarily for the most advanced and well-equipped browsers and then, as a final step or retrofit, tried to accommodate the rest. This approach adds a lot of work to the final stages of a project for the sake of a minority of users, so in the real world of web design, graceful degradation is often sacrificed due to all-too-common limits on time, money, and altruism. The result is an Internet full of web sites that still fail very *un*gracefully for a significant number of potential visitors.

Progressive enhancement is a newer approach that does justice to graceful degradation. A web site is first designed to run in all user agents, and then enhancements are added to make the user's experience faster or richer if they're using a more sophisticated browser. This ensures essential functionality for every reader (human or machine) and supports the fancy browsers without sacrificing the simpler ones. It puts the vegetable course before the dessert and the horse before the cart. It makes sure the car has a working engine before adding the chrome and the sound system.

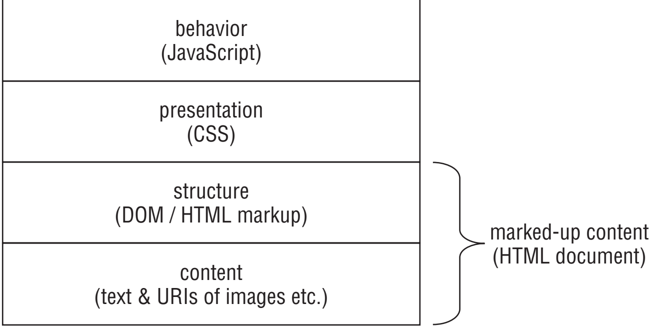
One way we manifest the principle of progressive enhancement in our work is by building pages that work in the absence of JavaScript. This might seem like heresy in a JavaScript "Bible," but the bottom line is that JavaScript is one of many tools we use to accomplish our work. No single tool is so shiny and clever that it should upstage the real star of the show — the web site visitor.

Separation of development layers

Another important principle in modern web design is the separation of layers of development. The main layers we work with are the content of a page (such as text and image names), its structure in HTML markup, its behavior in JavaScript, and its presentation in CSS (see Figure 2-2). The more you can keep these layers separate, the easier it will be to build, modify, and re-use your work.

FIGURE 2-2

Four layers of client-side web development.



Old-school web pages are easy to spot: just look at the HTML source, and you'll see JavaScript and styling attributes embedded in the markup. This is like a house built with the wiring and pipes embedded in cement walls. If you want to change or replace the wiring, you've got a real mess on your hands. These cemented pages are redundant, since identical features on different pages require the same bloated markup to be downloaded each time. For that reason, they're also slower to download.

With separated layers, each component file is fairly trim and fast to download, and if the same style sheet or JavaScript script applies to more than one page, as is often the case, the response time is even faster since the browser caches (stores) files it has already fetched from the server. Web site development and redesign are also made more efficient because several people can work on different layers without stepping on one another's toes. It becomes possible to develop *modules* of markup, script, and style that can more easily be tweaked and re-used for future projects.

Separate development layers aren't unrelated. Just the opposite, in fact: the style sheet and JavaScript code refer to and depend on the HTML markup structure, tags, and attributes. They are hands whose fingers are designed to mesh perfectly, but they are still separate hands.

The first step in separating layers is literally to create separate files for your HTML, JavaScript, and CSS. (The HTML files you'll see in this book are already combinations of markup and content, which, in the everyday world of web production, are often a merger of HTML templates and database fields performed by a server-side script.)

Separation of layers is also an approach we can take to our scripting logic. JavaScript has the ability to write a chunk of cemented markup and text to a document in one operation, but if we separate the new structural elements from their content in our script, we end up with code that is easier to

debug and re-use, and that contains fewer errors. Similarly, JavaScript can directly tweak the styling of an element on a page, changing its color or size or position, but instead, if we simply assign an id or class that is separately defined in the style sheet, then a graphic designer can change that presentation without anyone having to modify the JavaScript.

Further reading

Progressive Enhancement: Paving the Way for Future Web Design, by Debra Chamra
`http://hesketh.com/publications/articles/progressive-enhancement-paving-the-way-for/`

Graceful Degradation & Progressive Enhancement, by Tommy Olsson
`http://accessites.org/site/2007/02/graceful-degradation-progressive-enhancement/`

Progressive enhancement, in Wikipedia
`http://en.wikipedia.org/wiki/Progressive_enhancement`

CHAPTER 3

Selecting and Using Your Tools

IN THIS CHAPTER

How to choose basic JavaScript authoring tools

How to set up your authoring environment

How to enter a simple script to create a web page

In this chapter, you set up a productive script writing and previewing environment on your computer and learn where to find valuable resources on the Web. We also produce sample HTML, JavaScript, and CSS files to demonstrate how to use these tools to create real web pages.

Because of differences in the way various personal computing operating systems behave, we present details of environments for two popular variants: Windows (95 through XP) and Mac OS X. For the most part, your JavaScript authoring experience will be the same regardless of the operating system platform you use — including Linux or Unix. Although there may be slight differences in font designs, depending on your browser and operating system, the information remains the same. Most illustrations of browser output in this book are made from the Windows XP versions of Internet Explorer and Firefox. If you run another browser or version, don't fret if every pixel doesn't match the illustrations in this book.

The Software Tools

The best way to learn JavaScript is to type the HTML and scripting code into documents in a text editor. Your choice of editor is up to you, although we provide some guidelines for choosing.

Choosing a text editor

For the purposes of learning JavaScript in this book, avoid WYSIWYG (What You See Is What You Get) web page authoring tools such as FrontPage and Dreamweaver. These tools may come in handy later for molding your content and layout. But the examples in this book focus more on script content (which you must type anyway), so there isn't much HTML that you have to type. Files for all complete web page listings in this book (except for the tutorial chapters) also appear on the companion CD-ROM.

An important factor to consider in your choice of editor is how easy it is to save standard text files with an `.html` filename extension. In the case of Windows, any program that not only saves the file as text by default, but also enables you to set the extension to `.htm` or `.html`, prevents a great deal of problems. If you use Microsoft Word, for example, the program tries to save files as binary Word files — something that no web browser can load. To save the file initially as a `.txt` or `.html` extension file requires mucking around in the Save As dialog box. This is truly a nuisance.

Perhaps more importantly, a word processor such as Microsoft Word opens with a lot of default settings that may make desktop publishing easier but can frustrate a programmer, such as inserting spaces around words and automatically replacing straight quotes with curly quotes. Setting its defaults to make it programmer-friendly will make it less useful as a word processor.

Finally, we urge you not to create documents in Word and "save them as a web page." This will generate an HTML document that will look much like the word processing document on which it's based, but the actual HTML coding it produces will be bloated and redundant — a far cry from the sleek and elegant code that we hope you will be producing after you read this book. Word just isn't the right tool for this job.

Nothing's wrong with using bare-essentials text editors. In Windows, that includes the WordPad program or a more fully featured product such as Visual Studio or the shareware editor called TextPad. For Mac OS X, the bundled TextEdit application is also fine. Favorites among Mac HTML authors and scripters include BBEdit (Bare Bones Software) and SubEthaEdit (`www.codingmonkeys.de/subethaedit`).

Choosing a browser

The other component that is required for learning JavaScript is the browser. You don't have to be connected to the Internet to test your scripts in the browser. You can perform all testing offline. This means you can learn JavaScript and create cool, scripted web pages with a laptop computer — even on a boat in the middle of an ocean.

The browser brand and version you use are up to you. Because the tutorial chapters in this book teach the W3C DOM syntax, you should be using a recent browser. Any of the following will get you through the tutorial: Internet Explorer 5 or later (Windows or Macintosh); any Mozilla-based browser (including Firefox, Netscape 7 or later, and Camino); Apple Safari; and Opera 7 or later.

Note

Many example listings in Parts III and IV of this book demonstrate language or document object model (DOM) features that work on only specific browsers and versions. Check the compatibility listing for that language or DOM feature to make sure you use the right browser to load the page. ■

Setting Up Your Authoring Environment

To make the job of testing your scripts easier, you want to have your text editor and browser running simultaneously. You need to be able to switch quickly between editor and browser as you experiment and repair any errors that may creep into your code. The typical workflow entails the following steps:

1. Enter HTML, JavaScript, and CSS into the source documents in the text editor.
2. Save them to disk.
3. Switch to the browser.

4. Do one of the following:
   - If this is a new document, open the file through the browser's Open menu.
   - If the document is already loaded, reload the file into the browser.

Steps 2 through 4 are actions you will take frequently. We call this three-step sequence the *save-switch-reload* sequence. You will perform this sequence so often as you script that the physical act will quickly become second nature to you. How you arrange your application windows and effect the save-switch-reload sequence varies according to your operating system.

In web site production, after you tweak your markup, style sheet, and scripts to your liking on your own computer, you'll upload them to your server using an FTP (File Transfer Protocol) program.

There are also online editors, but for now let's keep it simple. One of the advantages of doing your work off-line on your own computer is that you don't even have to have an active Internet connection during this stage of web site development.

Windows

You don't have to have either the editor or browser window maximized (at full screen) to take advantage of them. In fact, you may find them easier to work with if you adjust the size and location of each window so that both windows are as large as possible, while still enabling you to click a sliver of the other's window. Or, you can leave the taskbar visible so you can click the desired program's button to switch to its window (see Figure 3-1). A monitor that displays more than 800×600 pixels certainly helps in offering more screen real estate for the windows and the taskbar.

FIGURE 3-1

Editor and browser window arrangement in Windows XP.

```
function walkChildNodes(objRef, n) {
    var obj
    if (objRef) {
        if (typeof objRef == "string") {
            obj = document.getElementById(objRef)
        } else {
            obj = objRef
        }
    } else {
        obj = (document.body.parentElement) ?
            document.body.parentElement : document.body.parentNode
    }
    var output = ""
    var indent = ""
    var i, group, txt
    if (n) {
        for (i = 0; i < n; i++) {
            indent += "+---"
        }
    } else {
        n = 0
        output += "Child Nodes of <" + obj.tagName
        output += ">\n=====================\n"
    }
    group = obj.childNodes
```

In practice, however, the Windows Alt+Tab task-switching keyboard shortcut makes the job of the save-switch-reload steps outlined earlier a snap. If you run Windows and also use a Windows-compatible text editor (which more than likely has a Ctrl+S file-saving keyboard shortcut), you can effect the save-switch-reload sequence from the keyboard with your the left hand: Ctrl+S (save the source file), Alt+Tab (switch to the browser), and Ctrl+R (reload the saved source file).

As long as you keep switching between the browser and the text editor via Alt+Tab task-switching, either program is always just an Alt+Tab away.

Mac OS X

In Mac OS X, you can change between your text editor and browser applications via the Dock, or, more conveniently, by pressing ⌘+Tab. As long as you stay in those two applications, the other program is only one ⌘+Tab away (see Figure 3-2).

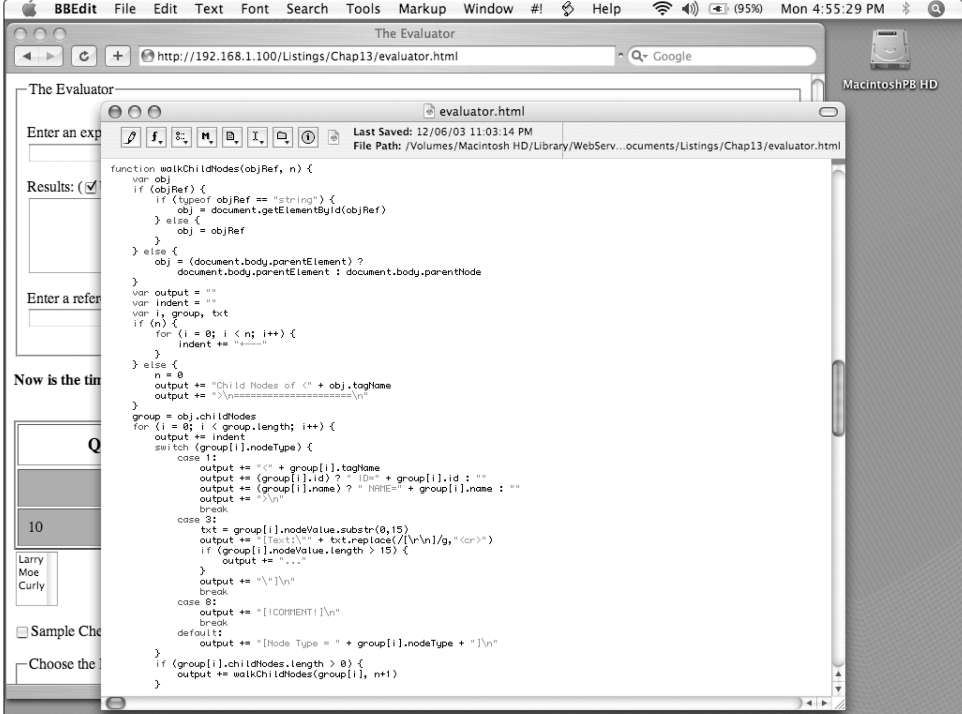
With this setup, the save-switch-reload sequence is a simple affair:

1. Press ⌘+S (save the source file).
2. Press ⌘+Tab (switch to the browser).
3. Press ⌘+R (reload the saved source file).

To return to editing the source file, press ⌘+Tab again.

FIGURE 3-2

Editor and browser window arrangement on the Macintosh screen.



Reloading issues

For the most part, a simple page reload is enough to let you test a revised version of a script right away. But sometimes the browser's cache (with its default settings) can preserve parts of the previous page's attributes when you reload, even though you have changed the source code. To perform a more thorough reload, hold down the Shift key while clicking the browser's Reload/Refresh button. Alternatively, you can turn off the browser's cache in the preferences area, but that setting may negatively affect the overall performance of the browser during your regular web surfing.

Validate, Validate, Validate

You can save yourself hours of debugging by checking to make sure your HTML is valid. If your markup is flawed, chances are your JavaScript and CSS will not work as expected, because they both depend on HTML elements (tags) and their attributes. The more closely your markup matches the industry-standard specs, the more likely it is that you'll get consistent results across browsers that are all approaching the holy grail of web standards conformance. You should always proofread your own code, of course, but an automated validation tool helps for large, complicated pages, and for programmers who are still learning all the rules for correct markup.

The World Wide Web Consortium (W3C), which wrote the HTML specification, has developed just such a validator. It checks a page with the rules specific to the `DOCTYPE` that begins the page. Besides catching our typos, an added advantage of running the Validator is that it helps us learn the fine points of HTML markup. Now repeat after me: The Validator Is My Friend.

The W3C Validator is at `http://validator.w3.org/`. It offers three ways to input your markup — by entering its online URL, by uploading the `.html` file from your computer, and by copying and pasting the markup into the Validator directly. Then, click the Check button and read the results. Often, a single error of markup will result in several items in the Validator error report. If it found any errors in your markup, make the necessary corrections and run your code through the Validator again until you get the green success banner.

Using the Validator will probably make you curious about the HTML specifications. You can read them on the W3C web site:

- **HTML 4.01:** `http://www.w3.org/TR/html4/`
- **HTML5:** `http://www.w3.org/TR/html5/`
- **XHTML 1.0:** `http://www.w3.org/TR/xhtml1/`

As time goes on, you'll discover other useful specs and validators out there, such as those for CSS (Cascading Style Sheets) and web accessibility guidelines:

- **CSS 2.1 Specification:** `http://www.w3.org/TR/CSS21/`
- **CSS Validation Service:** `http://jigsaw.w3.org/css-validator/`
- **Web Content Accessibility Guidelines:** `http://www.w3.org/TR/1999/WAI-WEBCONTENT-19990505/`
- **Web Accessibility Evaluation Tools:** `http://www.w3.org/WAI/ER/tools/`

Creating Your First Script

To demonstrate how we'll use these tools, let's create a classic "Hello, World" page in three stages: first as plain HTML, then augmented with JavaScript, and finally styled with CSS.

For the sake of simplicity, the kind of scripts we're building now are the kind that run automatically, immediately after the browser loads the HTML page. Although all scripting and browsing work here is done offline, the behavior of the page is identical if you place the source file on a server and someone accesses it through the web.

To begin, open your text editor and your browser. Also run Windows File Manager/Macintosh Finder to create a folder in which to save your scripts.

Stage 1: static HTML

Create a new text file and enter the markup in Listing 3-1.

LISTING 3-1

Source Code for hello-world.html

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Hello, World</title>
   </head>
   <body>
      <h1>Hello, World</h1>
      <p>This is HTML.</p>
   </body>
</html>
```

Save this file to your hard drive as `hello-world.html`, and open it in your browser. Figure 3-3 shows the page as it appears in the browser after you're finished. The precise appearance (or voice, if you're using a screen-reader) may vary slightly from one browser to the next, since at this point we're relying on the default style sheet in the browser to render the page.

The `head` of this page contains two required elements: a `meta` tag that declares the MIME type and character set, and a `title`. The `body` consists of a headline and a paragraph of text, pure and simple.

Stage 2: connecting with JavaScript

Now let's add JavaScript to the page. Create a new text file on your computer and enter Listing 3-2. Be sure and spell everything exactly, including upper- and lowercase:

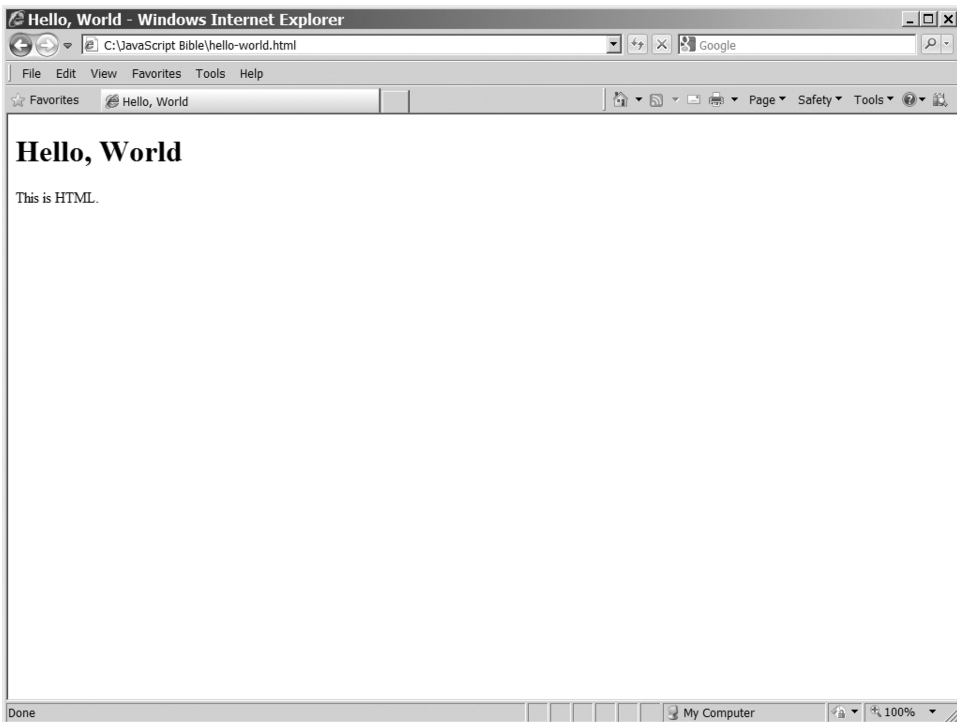
LISTING 3-2

Source Code for hello-world.js

```
var today = new Date();
var msg = "This is JavaScript saying it's now " + today.toLocaleString();
alert(msg);
```

FIGURE 3-3

The static HTML file displayed in a browser.



Save this file as `hello-world.js`. It consists of just three JavaScript statements: the first gets the current date, the second composes a brief message, and the third displays the message.

To enable this JavaScript program to act on your HTML document, add a new `script` tag to the `head` section of the HTML (shown highlighted in Listing 3-3):

LISTING 3-3

Revised Source Code for hello-world.html

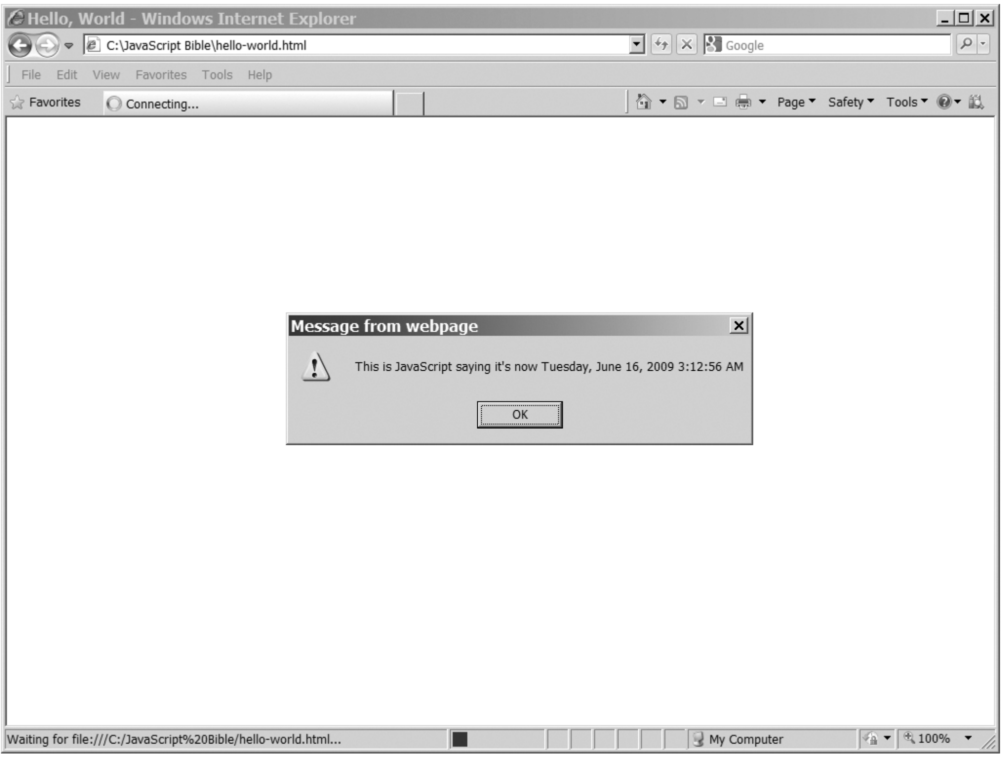
```
...
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Hello, World</title>
      <script type="text/javascript" src="hello-world.js"></script>
   </head>
...
```

The HTML `script` element tells the browser the type and name of the script file to combine with the document. Because the `src` (source) attribute doesn't include a path, the system assumes it's in the same folder as the HTML file. Unlike the `meta` tag, every `<script>` tag must be closed with `</script>`.

When you save the new JavaScript file and the modified HTML file, reload the HTML file in your browser. It should look something like Figure 3-4.

FIGURE 3-4

A JavaScript alert().



The text we passed to `alert()` appears in what's called a *modal dialog*. "Modal" means that you can't do anything more with the application (your browser) until you close the dialog. Notice that the HTML headline and paragraph seem to have disappeared! No worries; they will appear as soon as you click OK on the modal dialog. Why? The browser renders the HTML file from top to bottom. It encounters the `script` tag in the `head` and runs the named JavaScript program before it renders the HTML `body`. In this case, the modal dialog produced by the `alert()` method halts the rendering of the rest of the page until we respond.

Stage 3: styling with CSS

Let's add some styling to the page so that you can see how HTML and CSS interact. Create a new text file on your computer and enter Listing 3-4:

LISTING 3-4

Source Code for hello-world.css

```
h1
{
   font: italic 3em Arial;
}
p
{
   margin-left: 2em;
   font-size: 1.5em;
}
```

Save this file to your hard drive as `hello-world.css`.

This style sheet applies a particular font and color to all the `h1` elements on the page, and a left margin and font-size to all the `p` elements. Of course, in our document, there is only one of each.

To enable this style sheet to affect your HTML document, add a `link` tag to the `head` section of the HTML (shown highlighted in Listing 3-5):

LISTING 3-5

Revised Source Code for hello-world.html

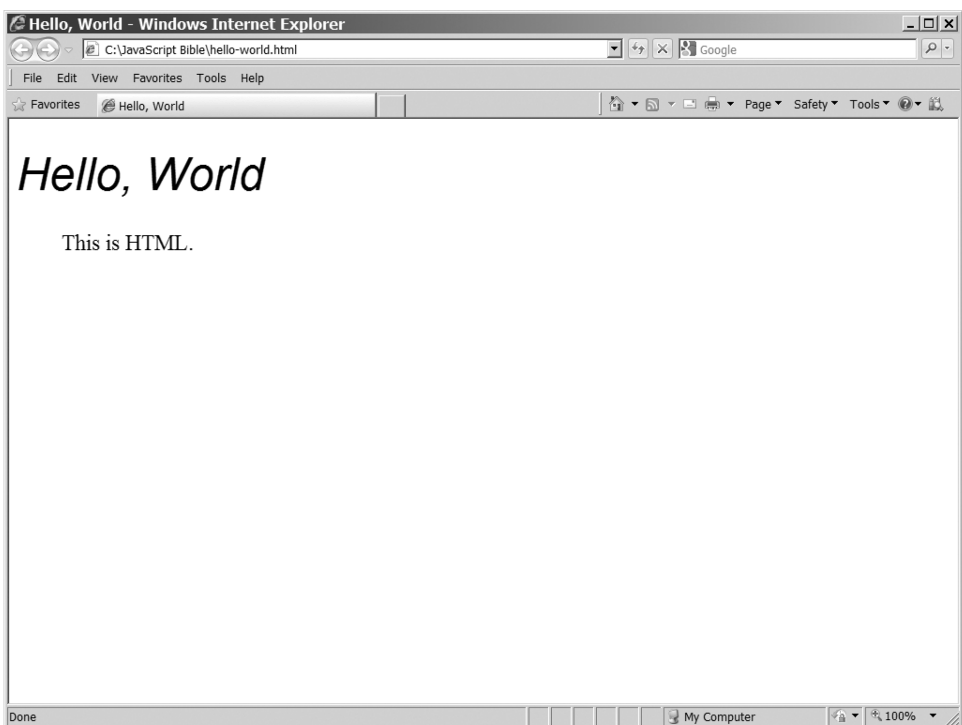
```
...
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Hello, World</title>
      <script type="text/javascript" src="hello-world.js"></script>
      <link type="text/css" rel="stylesheet" href="hello-world.css">
   </head>
...
```

The `link` element tells the browser the type, relationship, and name of the style sheet to combine with the document. Note that, like the `meta` tag and unlike the `script` tag, `link` is an "empty" tag and does not take a separate closing tag. Because the `src` (source) attribute doesn't include a path, the system assumes it's in the same folder as the HTML file. Every `<script>` tag must be closed with `</script>`.

When you save these files and reload the HTML document in your browser (and click OK on that modal dialog), it will look something like Figure 3-5.

FIGURE 3-5

A styled page.



CHAPTER

JavaScript Essentials

When first learning to use JavaScript in the context of the web browser environment, it can be easy to confuse the objects of the JavaScript language with the document objects that we control with JavaScript. It's important to separate the language from the Document Object Model (DOM) to help you make important decisions when designing JavaScript-enhanced pages. You may come to appreciate the separation in the future if you use JavaScript for other object models, such as in server-side programming or scripting Flash animations. All the basics of the language are identical from one context to another; only the objects differ.

This chapter introduces many aspects of the core JavaScript language, particularly as they relate to deploying scripts in a world in which visitors to your pages may use a wide variety of browsers. Along the way, you'll receive additional insights into the language itself. Fortunately, browser differences, as they apply to JavaScript, have lessened considerably as modern browsers continue to inch closer to consistently supporting the JavaScript (ECMAScript) standard. You can find details about the JavaScript core language syntax in Part III.

IN THIS CHAPTER

How to combine JavaScript with HTML

How to accommodate different versions of JavaScript

Language highlights for experienced programmers

Combining JavaScript with HTML

Scriptable browsers offer several ways to include scripts or scripted elements in your HTML documents. Not all approaches are recommended in today's best practices, but it's useful to learn even the old-school techniques so that you can understand legacy code when you come across it on the Web.

<script> tags

We marry JavaScript code to an HTML document using a `<script></script>` tag set that specifies the scripting language

through the `type` attribute. You can have any number of such tag sets in your document. We recommend that all your scripts be external files linked to the HTML:

```
<script type="text/javascript" src="example.js"></script>
```

However, you can also embed JavaScript directly into the HTML by wrapping it in a `<script>...</script>` tag:

```
<script type="text/javascript">
// JavaScript code goes here
</script>
```

There are distinct advantages to linking external JavaScript to HTML rather than embedding the scripts directly in the markup. The principal of *separation of development layers* encourages us to keep different aspects of a document apart, typically in separate, linked files. Doing so facilitates cleaner development, faster downloads, and more modular, re-purposable code that can be much easier to modify than mixtures of technologies cemented together in a single file.

Linking to script libraries (.js files)

The advantages of linking to external JavaScript files become immediately obvious when the same script needs to run on more than one page of a site. If the script were embedded in the HTML markup, each page would be bloated with redundant content, and modifying the script would necessitate repeating the same changes in more than one file. In contrast, linking to an external file adds only a single line to each HTML file, and the external JavaScript file can be modified just once to immediately affect every page that includes it. Another plus is that, by separating the script code from the HTML document, you will not have to worry about comment-hiding or CDATA section tricks (see below).

Such an external script file contains nothing but JavaScript code — no `<script>` tags, no HTML. The script file you create must be a text-only file. Its filename extension can be anything, but the common convention is to use `.js`. To instruct the browser to load the external file at a particular point in your regular HTML file, you add a `src` attribute to the `<script>` tag as follows:

```
<script type="text/javascript" src="example.js"></script>
```

If you load more than one external library, you may include a series of these tag sets in the head of the document.

Notice that the `<script></script>` tag pair is required, even though nothing appears between them. Don't put any script statements between the start and end tags when the start tag contains a `src` attribute.

How you reference the source file in the `src` attribute depends on its physical location and your HTML coding style. In the preceding example, the `.js` file is assumed to reside in the same directory as the HTML file containing the tag. But you can just as easily link to JavaScript files located in other directories on your server or on other domains entirely. Just as with an HTML file, if you want to refer to an absolute URL, the protocol for the file is `http://`.

```
<script type="text/javascript" src="../scripts/example.js"></script>

<script type="text/javascript" src="http://www.example.com/example.js"></script>
```

(A critical prerequisite for using script libraries with your documents is that your web server software must know how to map files with the .js extension to a MIME type of application/x-javascript. Test your web server and, if necessary, arrange for server configuration adjustments. It is rare that a modern server will not correctly recognize .js files.)

When a user views the source of a page that links in an external script library, code from the .js file does not appear in the window, even though the browser treats the loaded script as part of the current document. However, the name or URL of the .js file is plainly visible (displayed exactly as it appears in your source code). Anyone can open that file in their browser (using the http:// protocol) to view the .js file's source code. In other words, an external JavaScript source file is no more hidden from view than JavaScript embedded directly in an HTML file. Nothing that your browser downloads to render a web page can be hidden from view.

Specifying the MIME type & language

Every opening <script> tag should specify the type attribute. script is a generic element indicating that the contained statements are to be interpreted as executable script and not rendered as HTML. The element is designed to accommodate any scripting language that the browser knows — in other words, for which it contains an interpreter. For example, Internet Explorer carries interpreters for both JScript (Microsoft's version of JavaScript) and VBScript.

```
<script type="text/javascript" src="example.js"></script>
<script type="text/vbscript" src="example.vb"></script>
```

Both of these scripts will be interpreted by IE, but a non-VBScript-aware browser such as Mozilla will attempt to interpret only the JavaScript file.

Specifying the language version

It is rarely necessary to specify which version of JavaScript you wish to invoke in your documents. If your script relies on the objects and methods of a recent version of JavaScript that some older browsers won't support, it's better to test for that support directly. For example:

```
// if this browser isn't DOM-aware, exit gracefully
if (!document.getElementById) return;

// now it's safe to use the DOM method
var oExample = document.getElementById("example");
```

However, it isn't always possible to test for the existence of language features at run-time. If the new feature is built into JavaScript's syntax, its mere presence in the script can stop the interpreter from compiling the script to the point where it can't even execute an if-test.

For example, take the E4X syntax introduced in JavaScript 1.6 (see Chapter 20, "E4X — Native XML Processing"):

```
var xAuthor = <name>
                  <first>George</first>
                  <last>Sand</last>
              </name>;
```

If you include this syntax in a script run by a browser that doesn't "know" E4X syntax, the script will simply not run, in some cases with no meaningful error message to alert the visitor or the developer. No code in the same script will run, even if it's within the abilities of the browser.

You can isolate the new syntax with a version-specific `type` attribute for the `script` element:

```
<script src="example-all.js" type="text/javascript"></script>
<script src="example-1-6.js" type="text/javascript;version=1.6"></script>
```

or:

```
<script type="text/javascript">
   // script for all browsers goes here
</script>
<script type="text/javascript;version=1.6">
   // script for browsers that know JavaScript 1.6 and later
</script>
```

Be aware, though, that specifying the version as a special flag is non-standard so not all browsers support it. In the specific case of E4X syntax, Mozilla has introduced another special flag for the `type` attribute: `e4x=1` meaning, essentially, "E4X support = true."

```
<script src="example-e4x.js" type="text/javascript;e4x=1"></script>
```

The practical question then arises: If you have to wall off part of your script from older browsers, what alternative code do you give to those older browsers in place of the new code? And if you're writing that work-around code, why not let that be the only code? What advantage is there in writing the same logic in two different ways in the same script? It's a question to ponder when working with leading-edge language enhancements.

The deprecated language attribute

The `type` attribute is required for the `<script>` tag as of HTML 4, formally published in 1998. Earlier browsers recognized the `language` attribute instead of `type`. The `language` attribute is *deprecated*, which means that it's an outdated feature of the language, has been replaced by newer constructs, and may become obsolete in future versions of HTML. Browsers continue to recognize it in order to support legacy web pages, but there's no reason to use it in industry-standard web sites today. We include it here so that you'll understand it when you encounter it in old markup.

The `language` attribute allowed the scripter to write for a specific minimum version of JavaScript or, in the case of Internet Explorer, other languages such as VBScript. For example, the JavaScript interpreter built into Navigator 3 knows the JavaScript 1.1 version of the language; Navigator 4 and Internet Explorer 4 include the JavaScript 1.2 version. For versions beyond the original JavaScript, you could specify the language version by appending the version number after the language name without any spaces, as in:

```
<script language="JavaScript1.1">...</script>

<script language="JavaScript1.2">...</script>
```

The `type` attribute didn't gain browser support until Internet Explorer 5, Mozilla, and W3C DOM-compatible browsers. If you need to inform even older browsers which language and

version of scripting you're using, you can specify both the `type` and `language` attributes in your `<script>` tags, as older browsers will ignore the `type` attribute:

```
<script type="text/javascript" language="JavaScript1.5">...</script>
```

Of course, if you're depending on features in JavaScript 1.5, you've forgone legacy browsers anyway. In this case, just take the forward-looking approach and test for modern methods at the beginning of the script.

The proprietary for and event attributes

Internet Explorer versions 4 through 7 offered a variation on the `<script>` tag that bound statements of a script to a specific object and an event generated by that object. In addition to the `language` attribute, the tag must include both `for` and `event` attributes (not part of the HTML specification). The value assigned to the `for` attribute was a reference to the desired object. Most often, this was simply the identifier assigned to the object's `id` attribute. The `event` attribute was the event handler name that you wanted the script to respond to. For example, if you designed a script to perform some action upon a `mousedown` event in a paragraph whose ID was `myParagraph`, the script statements were enclosed in the following tag set:

```
<script for="myParagraph" event="onmousedown" type="text/javascript">
...
</script>
```

Statements inside the tag set executed only upon the firing of the event. No function definitions were required.

Because the `for` and `event` attributes were proprietary creations of Microsoft, this way of binding an object's event to a script guaranteed that only Internet Explorer would execute the script when the named event occurred.

Even in its day, the `script for` tag was not a winner. Its attributes constituted a lot of source code overhead for each object's script, so it was never a very efficient technique for linking script statements to multiple objects and events. In addition, non-Internet Explorer and pre-Internet Explorer 4 browsers would execute the script statements as the page loaded, making this proprietary language feature impractical in any cross-browser script.

Old-school inline JavaScript

Back in the bad old days of "tag soup" web development, it was common practice to insert JavaScript statements right in the HTML markup:

```
<select id="nations" name="nations" onchange="doSomething(this);">
```

In that example, the `change` event for the control object `nations` is set to trigger the JavaScript statements provided: a custom function call. We'll show you how this works elsewhere in the book to help you read legacy code, but combining markup and scripting like that today is commonly considered to be "very '90s."

There are many disadvantages to inserting JavaScript into markup this way: it's awkward and expensive to change the markup or the scripting independently when they're this intermeshed; the downloaded markup is unnecessarily bloated and doesn't take full advantage of the centralized

modularity of externally linked scripts; and user agents running versions of JavaScript that can't handle the code are unable to avoid the script.

In a modern page, this markup would be cleaned up to

```
<select id="nations" name="nations">
```

with the event handler assigned entirely in JavaScript with something like

```
AddEvent("nations", "change", doSomething);
```

where the `AddEvent()` function can cater to a variety of event models in different browsers, can be entirely separate from the markup to protect user agents that can't handle scripting, and can be firewalled by defensive if-tests to protect it from legacy JavaScript interpreters that don't play well with modern code.

Accommodating the JavaScript-incapable user agent

There are several categories of user agent that can't or don't support JavaScript, representing, by some accounts, some ten to fifteen percent of all internet usage today. These user agents include search engine spiders, mobile device browsers, browsers inside government and corporate firewalls that strip out script for security reasons, assistive technologies, and browsers in which, for any of a variety of reasons, their users have turned scripting off. There is sure to be a small number of legacy browsers still in use today that cannot run JavaScript or that support very early versions, but the vast majority of non-JavaScript-using internet visitors are using contemporary hardware and software.

It is our responsibility as web developers to acknowledge that JavaScript is an option, not a given, and to ensure that our pages are functional even in its absence. It's our mission to use JavaScript to enhance the visitor's experience and not to supply critical elements without which a page would be broken or dead in the water. We make sure that the HTML markup provides all the fundamental content, hyperlinks, and form fields, and that server-side scripting completes the dialog between client and server that is the core of the HTTP request model. Then, when JavaScript is running, it can morph the page, pretty up the user interface, and quicken the response time. That is the win-win model of modern web scripting.

The question then comes, how can we add JavaScript to an HTML document in a way that leaves the page fully functional when the script interpreter is not running, not present, or packing an obsolete version? Our tactics are several:

- We begin our scripts by testing for the presence of modern DOM methods such as `document.getElementById` in order to turn away obsolete versions of the JavaScript interpreter.
- We export JavaScript to external files and link them to the HTML document with `script` tags. User agents not running a scripting interpreter won't even go there.
- Corollary to the above, we omit JavaScript from event attributes in HTML tags, and instead use JavaScript to assign event handlers to page elements.
- If it is ever unavoidable to include JavaScript in the HTML document itself, enclose the script in HTML comments so that it won't be rendered by not-script-savvy browsers.

Let's take a moment to expand on that last point.

Commenting out script in HTML

Non-scriptable browsers do not know about the <script> tag. Normally, browsers ignore tags that they don't understand. That's fine when a tag is just one line of HTML, but a <script>...</script> tag pair can enclose any number of script statement lines. Old and compact browsers don't know to expect a closing </script> tag. Therefore, their natural inclination is to render any lines they encounter after the opening <script> tag. Seeing JavaScript code spilling out onto a page is sure to confuse or erode the confidence of any visitor.

You can, however, exercise a technique that tricks most non-scriptable browsers into ignoring the script statements: surround the script statements (inside the <script> tag set) with HTML comment markers. An HTML comment begins with the sequence <!— and ends with —>. Therefore, you should embed these comment sequences in your scripts according to the following format:

```
<script type="text/javascript">
<!--
script statements here
//-->
</script>
```

JavaScript interpreters know to ignore a line that begins with the HTML beginning comment sequence <!--, but they need a little help with the ending sequence. The close of the HTML comment starts with a JavaScript comment sequence (//). This tells JavaScript to ignore the line; but a non-scriptable browser sees the ending HTML symbols and begins rendering the page with the next HTML tag or other text in the document. An older browser doesn't know what the </script> tag is, so the tag is ignored and rendering resumes after that.

Commenting out script in XHTML as character data

If your document is marked up with XHTML, the comment syntax is significantly different:

```
<script type="text/javascript">
<!--//--><![CDATA[//><!--
    // script statements here
//--><!]]>
</script>
```

That's quite a mouthful and deserves a bit of explanation.

XHTML parsers are intolerant of < symbols that don't begin elements and & symbols that don't begin HTML entities, yet these characters are commonplace JavaScript operators. The solution is to encase your script statements in what is known as a *CDATA* (pronounced "see-day-tah") section:

```
<![CDATA[
    XHTML permits any character here including < and &
]]>
```

We can then add the HTML comment markers to stop legacy browsers from rendering the script:

```
<script type="text/javascript">
<!--<![CDATA[
```

```
    // script statements here
//--]]>
</script>
```

However, we can't stop here because XHTML also obeys the `<!--` and `-->` comment tags and an XHTML-aware browser will ignore the CDATA section and not execute the script. So, we have to close the comment tag before the CDATA begins in order to satisfy XHTML:

```
<!-- --><![CDATA[
```

and use // comment syntax to make most HTML parsers ignore the rest of that line:

```
<!--//--><![CDATA[
```

Some HTML parsers will treat the line after // as a comment and others will close the comment area with `-->`, so we have to make the CDATA open-tag not show up. The technique we're using is to make the HTML parsers think it's a closed tag

```
<![CDATA[>
```

but to keep XHTML from considering it closed we'll add another comment marker

```
<![CDATA[//>
```

and open the block comment tag once more to stop some browsers from attempting to render the JavaScript code as plain text:

```
<!--//--><![CDATA[//><!--
```

Finally, the CDATA close-tag is commented out for HTML parsers:

```
//--]]>
```

Whew! That's some of the fun web developers get to have when making multiple standards play nicely with one another. How much cleaner to simply export the JavaScript to an external file!

The noscript tag

The `noscript` element is the converse of `script`. A JavaScript-aware browser will render the contents of the `noscript` block if scripting has been turned off or if the browser doesn't support a scripting language invoked by a `script` element in the document. (Significantly, a legacy browser that doesn't support scripting at all and doesn't recognize script-related tags will likewise render the contents of both `script` and `noscript` blocks, which is why we comment out the contents of a `script` block. A `noscript` block will therefore be rendered by all browsers when scripting is not supported.)

You can display any standard HTML within the `noscript` tag set. It is a block-level element that can contain paragraphs, lists, divisions, headings, and so on.

The conventional use of `noscript` is to present HTML content to display in the absence of scripting support — a message such as, "This page would be more interesting if viewed with JavaScript running," for example, or an HTML hyperlink to a page that provides content that would otherwise be inserted by Javascript.

That use of `noscript` has commonly been superseded in recent years by the practice of providing only the non-scripting markup in the HTML, then replacing or enhancing that structure with scripting. Rather than telling visitors that they ought to upgrade their browsers or turn on scripting, we begin with the assumption that they're looking at our pages with a user agent that they need or want to use and give them decent, functional pages; then use scripting to give JavaScript-capable browsers a nicer page. It seems unlikely that there is anyone today using a browser with scripting deliberately turned off, or running behind a corporate firewall that strips out scripting, or using a mobile device with no scripting support, who isn't aware of the additional delights that the internet has to offer with the help of JavaScript. An admonition to upgrade may have been useful advice back in the '90s, but today it seems condescending and redundant.

Listing 4-1 is an example of presenting different content to scriptable and non-scriptable browsers. The HTML includes a plain paragraph that everyone will see and another inside a `noscript` element that will only be seen by user agents that don't support scripting, have JavaScript turned off, or are running an old JavaScript interpreter that's not DOM-aware. The accompanying JavaScript inserts another paragraph that will only be revealed by user agents with running JavaScript interpreters that are aware of the W3C DOM methods `appendChild()` and `createTextNode()`.

LISTING 4-1

Presenting Different Content for Scriptable and Nonscriptable Browsers

HTML: jsb-04-01.html

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Presenting Different Content for Scriptable and
      Nonscriptable Browsers</title>
      <script type="text/javascript" src="../jsb-global.js"></script>
      <script type="text/javascript" src="jsb-04-01.js"></script>
   </head>
   <body>
      <h1>Presenting Different Content for Scriptable and Nonscriptable
      Browsers</h1>

      <p>This text will be revealed to everyone.</p>

      <noscript>
         <p>Only user agents that don't or can't support W3C DOM
         scripting will reveal this text.</p>
      </noscript>

   </body>
</html>
```

continued

LISTING 4-1 *(continued)*

```
JavaScript: jsb-04-01.js
// initialize when the page has loaded
addEvent(window, 'load', addText);

function addText()
{
      // ensure a DOM-aware user agent
      if (!document.appendChild || !document.createTextNode) return;

   // create a text node
   var oNewText = document.createTextNode("Only DOM-aware user agents ↩
running JavaScript will reveal this text.");

   // create a paragraph
   var oParagraph = document.createElement('p');

   // insert the text into the paragraph
   oParagraph.appendChild(oNewText);

   // insert the paragraph at the end of the document
   document.body.appendChild(oParagraph);
}
```

Hiding scripts (not!)

A common question from beginning JavaScript programmers is how to hide scripts from visitors to a page, perhaps to protect a cherished bit of clever code. The short answer is that this isn't possible. In order to run, client-side JavaScript must be downloaded with the page, and therefore is visible in the browser's source view. The URL for an external linked script can simply be pasted into a browser's address bar.

If you are worried about other scripters stealing your scripts, you could include a copyright notice in your source code. Not only are your scripts visible to the world, but so are a thief's scripts. This way, you can easily see when someone lifts your scripts verbatim. Of course, there's nothing stopping someone from copying your script and deleting the notice of ownership.

The most you can hope for along these lines is to *obfuscate* your code, or make it difficult, time-consuming, and tiresome for a human to read. Obfuscation techniques include removing carriage returns and unnecessary spaces and tabs, using meaningless function and variable names, using variables to hold commonly named objects, using Boolean logic in place of `if` and other syntactical branching, writing self-generating code, using recursion, fragmenting logic sequences into multiple nested function calls, and encoding text in decimal, hexadecimal, or octal expressions. At the end of the day, however, global-replacement, careful reading, and patience can expose the logic of any text-based script. The most productive use of a JavaScript obfuscator is simply to produce a "compressed" copy of a script using as few characters as possible for fast download of enormous scripts. However, even that use is questionable since external scripts are cached by the browser after the first download.

If keeping your source code private is a great concern, consider using another programming language that compiles to machine code. Because of the existence of *disassemblers* that turn machine language back into source code, compiling can be viewed as just another form of obfuscation, but it will certainly reduce the number of people willing to put effort into decoding it.

However, it's nearly impossible to conceal anything significant. Even if they can't read your script, good programmers can look at the way your software behaves and write a fresh script that accomplishes much the same thing. The time you take to try to conceal your script could be spent writing new, exciting code!

We suggest that you consider flipping the paradigm on its head. Instead of attempting to conceal your programming code, flaunt it! Add helpful comments to the source code, publish it on your blog and on programming web sites, and otherwise assist as many people as possible to learn about and use it. (Add a Creative Commons license to encourage people to copy your code and keep it free for public use.) If you gain a reputation as a clever and helpful programmer, people will come to you for more.

Scripting for different browsers

Cross-browser compatibility is a concern for every JavaScript programmer. Even though the most recent browsers are doing a decent job of providing a workable lowest common denominator of scriptability, you will still have to consider a range of JavaScript support, from the most advanced browser to the least. Even the differences between the latest Firefox and the latest Internet Explorer are significant. Planning for compatibility begins with deciding on your goals for various classes of user agent.

Building the foundation beneath the towers

The purpose of most web pages can be boiled down to "delivering particular content." Where one web site differs from another is the specific content being delivered and its "look and feel" — the graphic appearance of the page and the style of the user interface with which visitors request further information.

The principal of progressive enhancement guides us first to deliver the content from the server to the client in HTML markup, without relying on JavaScript for core functionality. A web site should be fully usable and navigable, with all its public content exposed to all user agents, including those that don't support scripting . . . such as search engines! After that, we use JavaScript to enhance the visitor's experience and make it faster, easier, more attractive, or more fun in browsers with JavaScript enabled.

One of the consequences of considering JavaScript last instead of first in the planning process is that we'll often discover that much of a web site's functionality that we might have expected to hand over to JavaScript is instead handled perfectly well by other, more universally supported technologies. We should ask JavaScript to perform only those functions that are outside the reach of core browser functionality such as HTML rendering and CSS styling. For example, navigation links should be marked up as functional hyperlinks regardless of whether JavaScript adds any spice to the mix. Multi-level menus, pop-up dialogs, and roll-over image changes can often be implemented using just HTML and CSS, and therefore work for a broader audience.

Ajax (Asynchronous JavaScript and XML) is an interesting case in point. Often developers and even clients will choose Ajax to deliver content just because it's perceived to be sexy technology, but without pausing to consider whether it's really more useful, faster, or more effective than normal page navigation. Reading content from the server takes a certain amount of time regardless; if all the images on a page are already downloaded and therefore cached, it's only the HTML markup that will be the overhead of reading a new page, and that can be the smallest slice of a page. Navigating to individual pages to reveal different content means that visitors can easily bookmark and share links to

specific chunks of content, which isn't normally true of Ajax-retrieved content. Just because we *can* use JavaScript for a particular purpose doesn't mean we have to or should.

Once the core page or site is planned in detail using server-side technologies and HTML/CSS downloads, we can plot out the remainder of the user interface with JavaScript — the sweet dessert that follows the entrée!

JavaScript can be used to add animation to an otherwise static presentation, replace a navigation link with an Ajax download of additional content, augment a button-click editing process with drag-and-drop, fancify a form, and so on. While adding these nice-to-haves, make sure that all the content you're providing through JavaScript is also accessible without scripting. If you find yourself adding critical content to the site that can only been retrieved using JavaScript, you know that you've gone too far. Step back, add the new content to the core web site, then resume your JavaScript planning.

Choosing your battles

Any public web site can, and ultimately will, be viewed using every sort of user agent in existence. Because the range of JavaScript support is so broad — ranging from Early Primitive to Leading Edge — it can be quite challenging, verging on impossible, to write for every browser that's got a JavaScript interpreter. Part of web development planning can therefore be viewed as 'choosing your battles' or a kind of triage — drawing the line between interpreters you'll support and those you won't, and language features you'll use and those you'll avoid. According to this approach, instead of spending time trying to write script that will run in ancient browsers, simply block them from the playing field and focus your attention on developing code for the modern ones.

If, as we recommend, you've engineered your site so that all the core content and functionality are provided by HTML delivered from the server, you won't have to feel negligent if you prevent legacy browsers from trying to run your script. Relax — everyone will be able to use your site. By limiting the JavaScript interpreters you support to the current crop of modern ones and shutting out the older ones, you can save yourself hours of painstaking development work for a small minority of visitors, a decision that can salvage your deadline, your budget, and your sanity.

For example, most of the scripts in this book begin with the simple if-test

```
if (!document.getElementById) return;
```

In other words, "if the JavaScript interpreter running in this browser does not know about the `getElementById` method of the `document` object, stop running this script." This one statement shuts out interpreters that are too old to know about modern Document Object Model methods, allowing you to write most of your code for a single audience. At the other end of the spectrum, you may wish to avoid, for now, using leading-edge JavaScript syntax such as E4X that will stop most current browsers cold in their tracks. The middle ground — ECMAScript 3- and W3C DOM2-aware browsers — constitutes the vast majority in use today, and presents a very comfortable compromise with relatively consistent language support.

In planning your application or web site, it makes far more sense to deliver a single stream of content that can be viewed through all user agents rather than duplicating content in different modes. The latter approach was sometimes used in years past, for a site using frames and a parallel no-frames version, for example. In case you have a notion of creating an application or site that has multiple paths for viewing the same content, don't forget that maintenance chores lie ahead as the site evolves. Will you have the time, budget, and inclination to keep all paths up to date? Despite whatever good intentions a designer of a new web site may have, in our experience, the likelihood that a site will be maintained properly diminishes rapidly with the complexity of the maintenance task.

That said, providing multiple channels does not necessarily mean a duplication of effort. A site may be marked up to navigate to separate pages to deliver different content, but when scripting is supported the site could deliver the same content using Ajax. Since both apparent channels of content come from the server, we could use Ajax to request the same data stream that each stand-alone page requests. When the data is changed on the server, say, by modifying text in a database, both the stand-alone page and the Ajax content injection change accordingly.

Object detection

The methodology of choice by far for implementing browser version branching is *object detection*. The principle is simple: If an object type exists in the browser's object model, it is safe to execute script statements that work with that object.

One example of object detection that you'll see in a lot of older scripts uses the `images` collection. A script can change the `src` attribute of an image — effectively replacing it with another image read from the server — only if the version of JavaScript running in the browser recognizes images as objects. Object models that implement images always include an array of image objects belonging to the `document` object. The `document.images` array always exists, even with a length of zero when no images are on the page. Therefore, if you wrap the image-swapping statements inside an `if` construction that lets browsers pass only if the `document.images` array exists, older browsers simply skip the statements:

```
function imageSwap(imgName, url)
{
   if (document.images)
   {
      document.images[imgName].src = url;
   }
}
```

Object detection works best when you use it to test for the same object you need to use. It can get you into trouble when you use it to make assumptions about browser version and support for language features other than the one being tested.

For example, Internet Explorer 4 introduced a `document` object array called `document.all`, which is used very frequently in building references to HTML element objects. Netscape Navigator 4, however, did not implement that array; instead, it had a document-level array object called `layers`, which was not implemented in Internet Explorer 4. Unfortunately, many scripters used the existence of these array objects not as prerequisites for addressing those objects, but as determinants for the browser version. They set global variables signifying a minimum version of Internet Explorer 4 if `document.all` existed and Netscape Navigator 4 if `document.layers` existed. This was dangerous because there was no way of knowing whether a future version of a browser might adopt the object of the other browser brand or eliminate a language feature. In fact, when the Mozilla-based Netscape version first arrived, it did indeed remove all the layers stuff, replacing it with W3C standards-based features. Tons of scripts on the web used the existence of `document.layers` to branch to Netscape-friendly code that didn't even use `document.layers`. Thus, visitors using Netscape 6 or 7 found that scripts either broke or didn't work, even though the browsers were more than capable of doing the job.

This is why we recommend object detection, not for browser version sniffing, but for object availability branching, as previously shown for images. Moreover, it is safest to implement object detection only when all major browser brands (and the W3C DOM recommendation) have adopted the object so that behavior is predictable wherever your page loads in the future.

Techniques for object detection include testing for the availability of an object's method. A reference to an object's method returns a value, so such a reference can be used in a conditional statement. For example, the following code fragment demonstrates how a function can receive an argument containing the string ID of an element and convert the string to a valid object reference for three different DOMs:

```
function myFunc(elemID)
{
   var obj;
      if (document.getElementById)
      {
         obj = document.getElementById(elemID);
      }
      else if (document.all)
      {
         obj = document.all(elemID);
      }
      else if (document.layers)
      {
         obj = document.layers[elemID];
      }

      if (obj)
      {
         // statements that work on the object
      }
}
```

With this object detection scheme, it doesn't matter which browser brand, version, or operating system support a particular way of changing an element ID to an object reference. Whichever `document` object property or method is supported by the browser (or the first one, if the browser supports more than one), that is the property or method used to accomplish the conversion. If the browser supports none of them, no further statements execute in this function. Keep in mind, however, that the first approach in this example is sufficient (and recommended) as the technique for obtaining all objects from an ID in modern browsers.

If your script wants to check for the existence of an object's property or method, you may also have to check for the existence of the object beforehand, if that object is not part of all browsers' object models. An attempt to reference a property of a nonexistent object in a conditional expression generates a script error. To prevent the error, you can cascade the conditional tests with the help of the && operator. The following fragment tests for the existence of both the `document.body` object and the `document.body.style` property:

```
if (document.body && document.body.style)
{
   // statements that work on the body's style property
}
```

This is the functional equivalent of:

```
if (document.body)
{
```

```
        if (document.body.style)
        {
           // statements that work on the body's style property
        }
}
```

In both examples, if the test for `document.body` fails, JavaScript bypasses the second test.

One potential "gotcha" in using conditional expressions to test for the existence of an object's property is that if the property exists, but its value is zero or an empty string, the conditional test responds the same as it would if the property did not exist. To work around this potential problem, the conditional expression can examine the data type of the value to ensure that the property genuinely exists. A nonexistent property for an object reports a data type of `undefined`. Use the `typeof` operator (discussed in Chapter 22, "JavaScript Operators") to test for a valid property:

```
if (document.body && typeof document.body.scroll != "undefined")
{
   // statements that work on the body's scroll property
}
```

We wholeheartedly recommend designing your scripts to take advantage of object detection in lieu of branching on particular browser name strings and version numbers. Scriptable features are gradually finding their way into browsers embedded in a wide range of nontraditional computing devices. These browsers may not go by the same names and numbering systems that we know today, yet such browsers may be able to interpret your scripts. By testing for browser functionality, your scripts will likely require less maintenance in the future.

Browser version detection

Years ago, before object detection caught on, the commonplace way to pilot a script through the rocky shoals of language support was *browser sniffing*. As described more fully in Chapter 42, "The Navigator and Other Environment Objects," your scripts can examine the `navigator` object to discover which make and model the current browser claims to be. After examining the browser's purported name and version number, a script can branch to different bits of logic specific to the various browsers available at the time. We don't use this technique any more, chiefly because browsers can and do lie about what they are, because browser version sniffing isn't future-proof, and because it's a hack — it tests for one thing in order to act on another.

Browser sniffing is like letting someone fly a jet because they say they're a pilot. They might not actually be a pilot just because they claim to be, and even if they are, they might have gotten their pilot's license before jets were invented. In contrast, with object detection, you find out if they know how to fly the particular aircraft at hand and you let them fly it only if they can — simple, direct, and foolproof.

Designing for Compatibility

Each new major release of a browser brings compatibility problems for page authors. It's not so much that old scripts break in the new versions (well-written scripts rarely break in new versions). The problems center on the new features that attract designers when the designers

forget to accommodate visitors who have not yet advanced to the latest and greatest browser version or who don't share your browser brand preference.

Catering only to the lowest common denominator can more than double your development time, due to the expanded testing matrix necessary to ensure a good working page in all operating systems and on all versions. Decide how important the scripted functionality you employ in a page is for every user. If you want some functionality that works only in a later browser, you may have to be a bit autocratic in defining the minimum browser for scripted access to your page; any lesser browser gets shunted to a simpler presentation of your site's data.

Another possibility is to make a portion of the site accessible to most, if not all, browsers, and restrict the scripting to the occasional enhancement that non-scriptable browser users won't miss. When the application reaches a certain point in the navigation flow, the user needs a more capable browser to get to the really good stuff. This kind of design is a carefully planned strategy that lets the site welcome all users up to a point, but then enables the application to shine for users of, say, W3C DOM–compatible browsers.

The ideal page is one that displays useful content in any browser but whose scripting enhances the experience of the page visitor — perhaps by offering more efficient site navigation or interactivity with the page's content. That is certainly a worthy goal to aspire to. But even if you can achieve this ideal on only some pages, you will reduce the need for defining entirely separate, difficult-to-maintain paths for browsers of varying capabilities.

Regardless of your specific browser compatibility strategy, the good news is that time tends to minimize the problem. Web standards have solidified greatly in the past few years, and browser vendors have made significant strides toward fully supporting those standards.

Dealing with beta browsers

If you have crafted a skillfully scripted web page or site, you may be concerned when a prerelease (or *beta*) version of a browser available to the public causes script errors or other compatibility problems to appear on your page. Do yourself a favor: Don't overreact to bugs and errors that occur in prerelease browser versions. If your code is well written, it should work with any new generation of browser. If the code doesn't work correctly, consider the browser to be buggy. Report the bug (preferably with a simplified test-case script sample) to the browser maker.

One exception to the "it's a beta bug" rule arose in the transition from Netscape Navigator 4 to the Mozilla engine (first released as Netscape Navigator 6). A conscious effort to eliminate a proprietary Netscape Navigator 4 feature (the `<layer>` tag and corresponding scriptable object) caused many Netscape Navigator 4 scripts to break on Moz1 betas (and final release). Had scripters reported the problem to the new browsers' developer (Mozilla), they would have learned about the policy change and planned for the new implementation. It is extremely rare for a browser to eliminate a popular feature so quickly, but it can happen. Stronger web standards have probably minimized the chances of this situation happening again any time soon.

It is often difficult to prevent yourself from getting caught up in a browser maker's enthusiasm for a new release. But remember that a prerelease version is not a shipping version. Users who visit your page with prerelease browsers should know that there may be bugs in the browser. That your code does not work with a prerelease version is not a sin; neither is it worth losing sleep over. Just be sure to connect with the browser's maker either to find out whether the problem will continue in the final release or to report the bug so that the problem doesn't make it into the release version.

Compatibility ratings in reference chapters

With the proliferation of scriptable browser versions since Navigator 2, it is important to know up front whether a particular language or object model, property, method, or event handler is supported in the lowest common denominator for which you are designing. Therefore, in this book, we include frequent compatibility ratings, such as the following example:

Compatibility: WinIE5+, MacIE5+, NN4+, Moz+, Safari+, Opera+, Chrome+

A plus sign after a browser version number means that the language feature was first implemented in the numbered version and continues to be supported in succeeding versions. A minus sign means that the feature is not supported in that browser. The browsers tested for compatibility include Internet Explorer for Windows and Macintosh, Netscape Navigator, Mozilla (including all browsers based on the Mozilla engine), Apple Safari, Opera, and Google Chrome. We also recommend that you print the JavaScript and Browser Object Quick Reference file shown in Appendix A. The file is on the companion CD-ROM in PDF format. This quick reference clearly shows each object's properties, methods, and event handlers, along with keys to the browser version in which each language item is supported. You should find the printout to be valuable as a day-to-day resource.

This is a great place to clarify what we mean by "all browsers based on the Mozilla engine." Once upon a time, *Mozilla* pretty much meant *Netscape*, but those days are long gone. Now there are several viable Mozilla-based browsers that fall under the Moz+ designation in the compatibility charts throughout this book, including Netscape, Firefox, Camino, SeaMonkey, Flock, and others.

Here we're using "Mozilla" to refer to the Gecko (née NGLayout) layout engine. The numbering systems of the individual browser brands are not synchronized to the underlying Mozilla engine versions, making it difficult to know exactly which browser supports what feature. The following table shows which individual browser brands and versions correspond to the Mozilla engine numbering system:

Mozilla	Netscape	Firefox	Camino
m18	6.0		
0.9.2	6.1		
0.9.4	6.2		
1.0.1	7.0		
1.2b		0.1	
1.3a		0.5	
1.4	7.1		
1.5		0.7	
1.7		1.0	
1.7.2	7.2		
1.7.5	8.0–8.1		
1.8		1.5	1.0
1.8.1	9.0	2.0	1.6.5

Mozilla	Netscape	Firefox	Camino
1.9		3.0	2.0
1.9.1		3.5	
1.9.2		3.6	
1.9.3		3.7	
2.0		4.0	

As you can see, Netscape 6.0 and 6.2 were based on Mozilla versions of less than 1. It is rare to see either of these versions "in the wild" these days. The focus, therefore, is on Moz1 and later. Thus, the compatibility charts use Moz1 as the baseline feature set.

In summary, when you see Moz+ in the compatibility charts, it ultimately resolves to Netscape 7 or later, Firefox 1 or later, and Camino 1 or later, to name the most popular Mozilla-based browsers currently in use.

Language Essentials for Experienced Programmers

In this section, experienced programmers can read the highlights about the core JavaScript language in terms that may not make complete sense to those with limited or no scripting experience. Here, then, is the quick tour of the essential issues surrounding the core JavaScript language:

- **JavaScript is a scripting language.** The language is intended for use in an existing *host environment* (for example, a web browser) that exposes objects whose properties and behaviors are controllable via statements written in the language. Scripts execute within the context of the host environment. The host environment controls which, if any, external environmental objects may be addressed by language statements running in the host environment. For security and privacy reasons, web browsers generally afford little or no direct access through JavaScript to browser preferences, the operating system, or other programs beyond the scope of the browser. The exception to this rule is that modern browsers allow deeper client access (with the user's permission) through trust mechanisms such as signed scripts (Mozilla) or trusted ActiveX controls (Microsoft).
- **JavaScript is object based.** Although JavaScript exhibits many syntactic parallels with the Java language, JavaScript is not as pervasively object-oriented as Java. The core language includes several built-in static objects from which working objects are generated. Objects are created through a call to a constructor function for any of the built-in objects plus the `new` operator. For example, the following expression generates a `String` object and returns a reference to that object:

  ```
  new String("Hello");
  ```

  Table 4-1 lists the built-in objects with which scripters come into contact.

TABLE 4-1

JavaScript Built-In Objects

Core Objects	Error Objects	XML Objects	JSON Object
Array[1]	Error[2]	Namespace[4]	JSON[5]
Boolean	EvalError[2]	QName[4]	
Date	RangeError[2]	XML[4]	
Function[1]	ReferenceError[2]	XMLList[4]	
Global	SyntaxError[2]		
Math	TypeError[2]		
Number[1]	URIError[2]		
Object[1]			
RegExp[3]			
String[1]			

[1]Although defined in ECMA Level 1, was first available in NN3 and IE3/J2.
[2]Defined in ECMA Level 3; implemented in Moz1.
[3]Defined in ECMA Level 3; implemented fully in NN4 and IE6.
[4]Defined in E4X; implemented in Mozilla 1.8.1 (Firefox 2.0).
[5]Defined in ECMA Level 5; implemented in Mozilla 1.9.1 (Firefox 3.5).

- **JavaScript is loosely typed.** Variables, arrays, and function return values are not defined to be of any particular data type. In fact, an initialized variable can hold different data type values in subsequent script statements (obviously not good practice but possible nonetheless). Similarly, an array may contain values of multiple types. The range of built-in data types is intentionally limited:
  - Boolean (`true` or `false`)
  - Null
  - Number (double-precision 64-bit format IEEE 734 value)
  - Object (encompassing the `Array` object)
  - String
  - Undefined
  - XML (in E4X)
- **The host environment defines global scope.** Web browsers traditionally define a browser window or frame to be the global context for script statements. When a document unloads, all global variables defined by that document are destroyed.
- **JavaScript variables have either global or local scope.** A global variable is initialized in `var` statements outside of all functions. In a web browser, it typically executes as the document loads. All statements in that document can read or write that global variable. A local variable is also initialized with the `var` operator but inside a function. Only statements inside that function may access that local variable.

- **Scripts sometimes access JavaScript static object properties and methods.** Some static objects encourage direct access to their properties or methods. For example, all properties of the `Math` object act as constant values (for example, `Math.PI`).
- **You can add properties or methods to working objects at will.** To add a property to an object, simply assign a value of any type to it. For example, to add an `author` property to a string object named `myText`, use:

  ```
  myText.author = "Jane";
  ```

  Assign a function reference to an object property to give that object a new method:

  ```
  // function definition
  function doSpecial(arg1)
  {
     // statements
  }
  // assign function reference to method name
  myObj.handleSpecial = doSpecial;
  ...
  // invoke method
  myObj.handleSpecial(argValue);
  ```

  Inside the function definition, the `this` keyword refers to the object that owns the method.
- **JavaScript objects employ prototype-based inheritance.** All object constructors create working objects whose properties and methods inherit the properties and methods defined for the *prototype* of that object. Scripts can add and delete custom properties and methods associated with the static object's prototype so that new working objects inherit the current state of the prototype. Scripts can freely override prototype property values or assign different functions to prototype methods in a working object without affecting the static object prototype. But if inherited properties or methods are not modified in the current working object, any changes to the static object's prototype are reflected in the working object. (The mechanism is that a reference to an object's property works its way up the prototype inheritance chain to find a match to the property name.)
- **JavaScript includes a large set of operators.** You can find most operators that you are accustomed to working with in other languages.
- **JavaScript provides typical control structures.** All versions of JavaScript offer `if`, `if...else`, `for`, and `while` constructions. JavaScript 1.2 (NN4+, IE4+, and all modern mainstream browsers) added `do while` and `switch` constructions. Iteration constructions provide `break` and `continue` statements to modify control structure execution.
- **JavaScript functions may or may not return a value.** There is only one kind of JavaScript function. A value is returned only if the function includes a `return` keyword followed by the value to be returned. Return values can be of any data type.
- **JavaScript functions cannot be overloaded.** A JavaScript function accepts zero or more arguments, regardless of the number of parameter variables defined for the function. All arguments are automatically assigned to the `arguments` array, which is a property of a function object. Parameter variable data types are not predefined.
- **Values are passed by reference and by value.** An object passed to a function is actually a reference to that object, offering full read/write access to properties and methods of that object. But other types of values (including object properties) are passed by value, with no reference

chain to the original object. Thus, the following nonsense fragment empties the text box when the `onchange` event fires:

```
function emptyMe(arg1)
{
   arg1.value = "";
}
...
<input type="text" value="Howdy" onchange="emptyMe(this)">
```

But in the following version, nothing happens to the text box:

```
function emptyMe(arg1)
{
   arg1 = "";
}
...
<input type="text" value="Howdy" onchange="emptyMe(this.value)">
```

The local variable (`arg1`) simply changes from `"Howdy"` to an empty string.

Note

The property assignment event handling technique in the previous example is a deliberate simplification to make the example very brief. It is generally better to use the more modern approach of binding events using the `addEventListener()` (NN6+/Moz/W3C) or `attachEvent()` (IE5+) methods. A modern cross-browser event handling technique is explained in detail in Chapter 32, "Event Objects." ■

- **Error trapping techniques depend on JavaScript version.** There was no error trapping in NN2 or IE3. Error trapping in NN3, NN4, and IE4 was event-driven in the web browser object model. JavaScript, as implemented in IE5+ and Mozilla, Safari, and other recent browsers, supports `try-catch` and `throw` statements, as well as built-in error objects that are not dependent on the host environment.
- **Memory management is not under script control.** The host environment manages memory allocation, including garbage collection. Different browsers may handle memory in different ways.
- **Whitespace (other than a line terminator) is insignificant.** Space and tab characters may separate lexical units (for example, keywords, identifiers, and so on).
- **A line terminator is usually treated as a statement delimiter.** Except in very rare constructions, JavaScript parsers automatically insert the semicolon statement delimiter whenever they encounter one or more line terminators (for example, carriage returns or line feeds). A semicolon delimiter is required between two statements on the same physical line of source code. Moreover, string literals may not have carriage returns in their source code (but an escaped newline character (`\n`) may be part of the string).

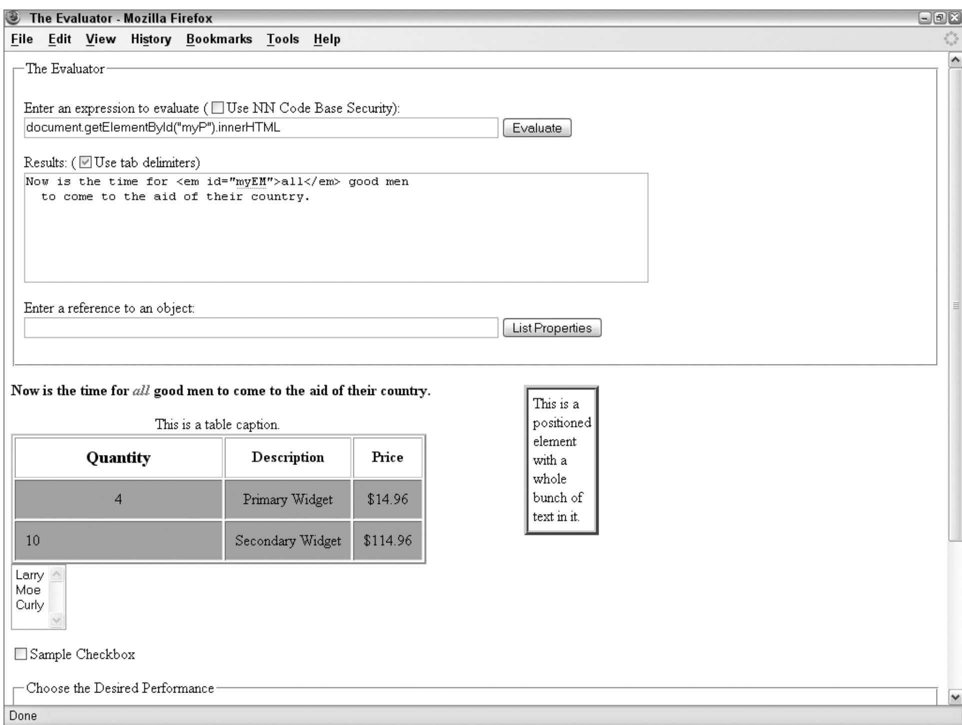
The Evaluator Sr.

The Evaluator Sr. is a tool you can use in succeeding chapters to help you learn both core JavaScript and DOM terminology. The Evaluator provides an interactive workbench to experiment with expression evaluation and object inspection. (In Chapter 8, "Programming Fundamentals, Part I," we will introduce a slimmed-down version we call The Evaluator Jr.)

Figure 4-1 shows the top part of the page. Two important features differentiate this full version from the Jr. version in Chapter 8.

FIGURE 4-1

The Evaluator Sr.



First, you can try some Mozilla secure features if you have Code Base Principles turned on for your browser (see Chapter 49, "Security and Netscape Signed Scripts," on the CD-ROM) and you check the Use Code Base Security check box (Netscape Navigator 4 or later/Moz only). Second, the page has several HTML elements preinstalled, which you can use to explore DOM properties and methods. As with the smaller version, a set of 26 one-letter global variables (`a` through `z`) are initialized and ready for you to assign values for extended evaluation sequences.

Copy the `evaluator.html` and `evaluator.js` files from the companion CD-ROM to a local hard disk and set a bookmark for it in all your test browsers. Feel free to add your own elements to the bottom of the page to explore other objects. We describe a version of The Evaluator for embedding in your projects as a debugging tool in Chapter 48, "Debugging Scripts" (on the CD-ROM), where you can learn more built-in functionality of The Evaluator.

Part II

JavaScript Tutorial

IN THIS PART

Chapter 5
Your First JavaScript Script

Chapter 6
Browser and Document Objects

Chapter 7
Scripts and HTML Documents

Chapter 8
Programming Fundamentals, Part I

Chapter 9
Programming Fundamentals, Part II

Chapter 10
Window and Document Objects

Chapter 11
Forms and Form Elements

Chapter 12
Strings, Math, and Dates

Chapter 13
Scripting Frames and Multiple Windows

Chapter 14
Images and Dynamic HTML

CHAPTER

5

Your First JavaScript Script

IN THIS CHAPTER

Creating a simple application consisting of an HTML page, a JavaScript script, and a CSS style sheet

In this chapter, you write a simple script, the results of which you can see in your JavaScript-compatible browser.

Two common ways of teaching spoken human languages are a) grammar lessons leading to eventual conversation and b) immersion in conversation leading to eventual grammar lessons.

We like b). It's more fun (play enhances learning), you get to see results right away, and it more closely resembles the way we each learned our own native tongue. The grammar's always there to refer to, but let's kick things off to a good start by plunging in and making a little magic happen on the screen.

Don't worry about memorizing every command and detail of syntax discussed in this chapter. Instead, relax and watch how the HTML markup and JavaScript statements become what you see in the browser. The details will soak in more quickly if you're enjoying yourself.

We'll explain each line of code as we go along, although all the methods we use in this first tutorial will be explained in much greater detail later in the book. For now, just enter the scripts as presented and see that they do what they're supposed to do.

What Your First Script Will Do

Our first tutorial script will insert the current date and time into a web page. (This is a further evolution of the "Hello, World" script we showed you in Chapter 3, "Selecting and Using Your Tools.") First, we'll create an HTML page that simply displays static content; then we'll add JavaScript to make it dynamic; and then we'll add styling to jazz it up.

It's worth noting that this sequence — building HTML structure and then adding dynamic JavaScript and CSS for presentation — is not useful merely when learning; it's also a very decent model for everyday, real-world web site production. It's part of a larger sequence of development that might

include information design, graphic design, wireframe and proof-of-concept, HTML markup, server-side scripting, CSS styling, and JavaScript. Although it might be tempting to jump to the scripting first, that's sort of like spreading the icing before the cake is baked. JavaScript, like CSS, needs to know its context — the structure of the page in which it's operating. Also, working in this sequence encourages us to adhere to the principle of progressive enhancement, whereby everyone can get something out of the fundamental page, but those with user agents (such as browsers) with more capabilities can get even more out of it. By starting with the HTML page, we ensure that it's complete and makes sense even without JavaScript.

These days most serious web production includes a server-side scripting language and database, such as PHP and MySQL, to provide the fundamental dynamic content as HTML markup. As server-side scripting is outside the realm of this book, we'll mention it from time to time, but we will otherwise assume a truly static HTML page.

Entering Your First Script

Launch your text editor and browser. You may also want to launch your standard file manager utility to monitor the files that you'll be creating in a folder on your computer.

We'll be working offline throughout this book. All operations will happen locally on your computer and without Internet connectivity. If your browser offers to dial your Internet service provider (ISP) or begins dialing automatically, you can cancel or quit the dialing operation. If the browser's Stop button is active, you can click it to halt any network searching it may try to do. You may receive a dialog-box message or page indicating that the URL for your browser's home page (usually the home page of the browser's publisher — unless you've changed the settings) is unavailable. That's fine. You want the browser open, but you don't need to be connected to your ISP. If you're automatically connected to the Internet through a local area network in your office or school, or through cable modem or DSL, that's also fine. However, you don't need the network connection for now.

Step 1: The HTML document

Figure 5-1 shows our first goal: a simple page with a headline and a paragraph of text. This is what our initial page will look like, allowing for differences in the way various browsers present content by default (in other words, that hasn't yet been styled by us).

Enter the following HTML markup into a new text file and save it (with UTF-8 character encoding) with the name `date-time.html`:

LISTING 5-1

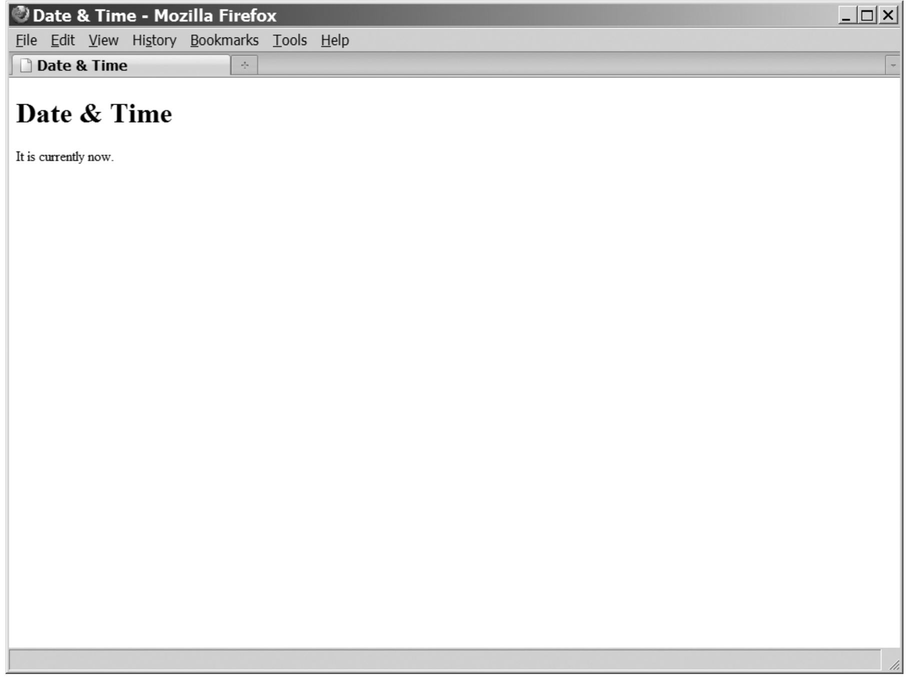
HTML Markup for "Date & Time": date-time.html

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Date &amp; Time</title>
   </head>
```

```
   <body>
      <h1>Date &amp; Time</h1>
      <p>It is currently <span id="output">now</span>.</p>
   </body>
</html>
```

FIGURE 5-1

Step 1: Static HTML page.



Note

Throughout this book, we will consistently use UTF-8 character encoding for our HTML files. A character encoding tells the browser how to display text characters, and the Unicode standard of UTF-8 is a splendid system that enables us to include virtually any world language in our HTML files. It will serve you well in your web site production.

From the start, get into the habit of saving your files with UTF-8 encoding. Many text editors enable you to choose the encoding in the Save As... dialog, as well as set the default encoding for future files so that you don't have to specify it each time. If your text editor doesn't give you this option, keep using it for now, but start looking around for another editor that will. ■

Now open the document in your browser. You can do this by double-clicking on the file in the file manager screen or by choosing Open or Open File in the File menu of your browser.

If you typed all lines as shown, the document in the browser window should look like the one in Figure 5-1 (with minor differences for your computer's operating system and browser version).

Let's examine the details of the document so that you understand some of the finer points of what the markup is doing.

DOCTYPE

```
<!DOCTYPE html>
```

The DOCTYPE tells the browser how to render the document. Which DOCTYPE you use greatly affects CSS styling and markup validation. The DOCTYPE we're using in this book is the one for HTML5, but you can replace it with either the DOCTYPE for HTML 4.01-Strict for validation purposes:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01//EN"
   "http://www.w3.org/TR/html4/strict.dtd">
```

or with the one for XHTML 1.0-Strict (along with a few other changes in markup style):

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
```

Note

XHTML style, if you intend to follow its conventions, requires all lowercase tags and attribute names. While the listings in this book are HTML, which accepts either upper- or lowercase tags and attributes, we will use lowercase consistently in order to remain compatible with both standards.

XHTML also requires several modifications to the HTML we're using in these examples. For more details on this, see the Introduction. ■

html

The markup continues:

```
<html>
   ...
</html>
```

The entire page will always reside inside the html element, which declares the type of markup. The html element has just two children, head and body.

head

```
<head>
   ...
</head>
```

The head element may contain a variety of child elements that convey information *about* a document but that are not normally rendered as document content. The head must have, at minimum, two children, a meta element defining the character set, and a title:

```
<meta http-equiv="content-type" content="text/html;charset=utf-8">
<title>Date &amp; Time</title>
```

meta tags are a lengthy topic unto themselves. For now, just enter this one exactly as above. HTML5 can accept a shorter version:

```
<meta charset="UTF-8">
```

However, the longer version works for all three markup types (HTML 4.01, HTML5, and XHTML 1.0). Note that the meta tag is "empty" and does not take a closing </meta> tag. All the information it conveys is in its attributes, here http-equiv and content.

title, on the other hand, does require a closing tag; together they surround their content. The title "Date & Time" is entered as Date & Time, using the HTML entity & for the ampersand. There are four characters that must always be converted to HTML entities if they appear in the *content* of the page — attribute values and the text enclosed by tags — in order not to confuse the parsing engine:

```
< &lt;
> &gt;
& &amp;
" &quot;
```

This last one is the straight double quote used in HTML markup, not the curly quotes common in typeset text.

body

```
<body>
   ...
</body>
```

The body element contains the document's content — its structure, text, images, multimedia objects, and so on.

headline

```
<h1>Date &amp; Time</h1>
```

We use the headline tags h1 through h6 to create an outline structure in a document. Ordinarily, a document page will have only one h1 to reiterate the page title, with all other headlines, starting with h2, nested beneath.

In the bad old days, people chose headline tags purely for their default appearance. Today, we use them just as you would use the numbers in a nested outline structure, and we control the appearance of all text from the style sheet.

paragraph

```
<p>It is currently <span id="output">now</span>.</p>
```

Finally, this is the paragraph of text where JavaScript will insert the current date and time. The span element has an ID (output) that JavaScript will use to locate it so that our script can replace the contents of the span with the actual date and time.

So, that's the static HTML page. Even after we plug in the JavaScript, anyone looking at the page with a user agent that doesn't support JavaScript (such as a search engine), or that has it turned off, will

see just what we're seeing here. The page isn't terrifically informative in its current state, but at least it isn't broken. For public web sites, scripting should add value to the page rather than be mission critical.

Step 2: Adding JavaScript

Next, let's create the JavaScript for our page. Now is a good time to instill some good JavaScript habits that will be important to you throughout all your scripting ventures. First, JavaScript is case-sensitive. Therefore, you must type every word in your script with the correct uppercase and lowercase letters. When a line of JavaScript doesn't work, look for the wrong case first. Always compare your typed code against the listings printed in this book and against the various vocabulary entries discussed throughout it.

Second, notice that each JavaScript statement ends in a semicolon. These trailing semicolons — which you can think of as periods at the end of sentences — technically are optional in JavaScript, but we strongly recommend that you use them to remove ambiguity from your scripts. If you someday investigate other programming languages such as PHP, Java, or C++, you'll find that those semicolons are required. All the JavaScript listings in this book use semicolons.

Create a second text file in your editor named `date-time.js` and enter the following script:

Note

A JavaScript comment is any text contained between a double forward slash (//) and the end of the current line, or between slash-asterisk (/*) and asterisk-slash (*/). Comments are ignored by the JavaScript interpreter and are added purely to help human readers understand what's going on in the code. Therefore, you don't have to enter the comments exactly as you see here, but we do recommend that you get in the habit of commenting your code. Other programmers and your own future self will appreciate it! ■

LISTING 5-2

JavaScript Code for "Date & Time": date-time.js

```
// tell the browser to run this script when the page has finished loading
window.onload = insertDateTime;

// insert the date & time
function insertDateTime()
{
      // ensure a DOM-aware user agent
      if (!document.getElementById) return;
      if (!document.createTextNode) return;

   // create a date-time object
   var oNow = new Date();

   // get the current date & time as a string
   var sDateTime = oNow.toLocaleString();

   // point to the target element where we want to insert the date & time
   var oTarget = document.getElementById('output');
```

```
      // make sure the target is found
      if (!oTarget) return;

   // delete everything inside the target
   while (oTarget.firstChild)
   {
      oTarget.removeChild(oTarget.firstChild);
   }

   // use the date-time string to create a new text node for the page
   var oNewText = document.createTextNode(sDateTime);

   // insert the new text into the span
   oTarget.appendChild(oNewText);
}
```

To connect this JavaScript file to the HTML file, we need to add a `script` element to the HTML `head`. Bring `date-time.html` back into your text editor and add the line highlighted below:

```
...
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Date &amp; Time</title>
      <script type="text/javascript" src="date-time.js"></script>
   </head>
...
```

The `script` tag identifies the MIME type of the linked JavaScript file and gives its name (and path, in case it's not located in the same folder as the HTML file itself). This causes the browser to read the JavaScript file as it's building the document page. The JavaScript interpreter within the browser validates and compiles the script, and runs it immediately, even before the rest of the HTML document has been read.

Now let's walk through the JavaScript code line by line so that you can see what it's doing.

Triggering the event

Here's a problem: Our script is going to insert the date and time into a paragraph in the page, but the browser reads and runs our JavaScript code before it's had a chance to read the HTML body. The paragraph into which we want to insert the new content won't even exist yet in the browser's memory when the JavaScript starts running.

Our solution is simply to tell JavaScript not to perform the insertion until the document has been fully read into memory. We do this using the `onload` event. Elements of the page — called *objects* in the Document Object Model (DOM) — can be made sensitive to a number of events, such as the click of a mouse or the press of a key. The `onload` event of the `window` object is triggered when the document finishes loading into the browser.

```
// tell the browser to run this script when the page has finished loading
window.onload = insertDateTime;
```

```
// insert the date & time
function insertDateTime()
{
   ...
}
```

In this case, we're telling JavaScript to call the `insertDateTime()` function when the document has finished loading. The date and time insertion routine that depends on the document having loaded is thus deferred until the browser has the target paragraph in memory.

Ensuring a safe environment

Once the document finishes loading into the browser, the `window.onload` event has been triggered, and the `insertDateTime()` function is called, the first thing the script does is to make sure it's being interpreted by a modern browser that speaks the same language:

```
// ensure a DOM-aware user agent
if (!document.getElementById) return;
if (!document.createTextNode) return;
```

`getElementById()` and `createTextNode()` are two DOM methods we're using in this function that some old JavaScript interpreters won't understand. The exclamation mark (`!`) means "not," and `return` tells JavaScript to stop running the current function; so, the first statement means, "If you aren't aware of the `getElementById()` method for the `document` object, exit this function." If we didn't include these safeguards, a legacy browser or its JavaScript interpreter would likely crash when trying to run the code that follows.

A crash can be as minor as the JavaScript interpreter stopping cold (leaving us with a plain HTML page and no JavaScript functionality at all) or as major as the whole browser freezing up and needing to be rebooted. Neither event is welcome. For many pages, we'll be writing JavaScript to perform several tasks, some of which can be executed by legacy browsers and others that can't. We want to guide the browser around the parts it can't handle, so it will run the parts it can. Preventing old JavaScript versions from crashing on the reefs of modern methods is a hallmark of professional piloting.

We subscribe to the principles of *graceful degradation* and *progressive enhancement*, whereby browsers that can't take advantage of particular features are let down gently and allowed to do the best they can. A fundamental courtesy of scripts that interact with the DOM is to turn away legacy JavaScript interpreters that aren't DOM-aware.

The old-school way of checking for old JavaScript interpreters is *browser-sniffing*, which examines the browser type as reported by the `navigator` object. Unfortunately, these claims of identity can't be relied on. What can be trusted is the JavaScript interpreter's own knowledge of which methods it's capable of executing, and that's the principal safety feature we use today.

Generating the date and time

The text we're going to insert into this document is the current date and time (according to the clock in the user's computer). JavaScript makes this easy: we create a `Date` object and use one of its many methods to produce the string:

```
// create a date-time object
var oNow = new Date();
```

```
// get the current date & time as a string
var sDateTime = oNow.toLocaleString();
```

This code sets the variable `oNow` to a new `Date` object. Once we have that, we use its `toLocaleString()` method to output a text string containing something like:

```
Thursday, August 20, 2009 4:16:05 PM
```

We capture the date and time string in variable `sDateTime`, which we'll use in a moment to output to the page.

The beauty of the `toLocaleString()` method is that it outputs the date and time formatted according to the user's settings and in the user's time zone. If two people run this exact same code at the same moment on opposite sides of the earth, they're likely to get two different dates and times expressed in two different formats and in two different languages. You'll learn a lot more about the Date object in the chapters ahead, particularly Chapter 12, "Strings, Math, and Dates," and Chapter 17,
"The Date Object."

Note

In this example script, `oNow` is a `Date` object and `sDateTime` is a text string. One of the zillion flavors of programming style is the so-called *Systems Hungarian notation* in which each variable name begins with an indication of its type — here `o` for object, `s` for string, `i` for integer, and so on. Naming variables like this is purely optional, and for the programmer's own benefit. JavaScript doesn't care what the names are, as long as no keywords or illegal characters are used. JavaScript is a "dynamically typed" language in which the same variable can be set arbitrarily with numeric, string, Boolean, and object reference values. Some programmers like to maintain only one type of value in each variable to help keep the code easy to manage and debug. Using Hungarian notation can make these artificial variable types easier to remember. ■

Finding the target

The next step is to insert the date and time into the document exactly where we want it. To do that, we need to:

1. Refer to the target `span` element.
2. Delete any text already inside the element.
3. Insert new text inside the element.

To refer to the `output span`, the script uses the industry-standard way to refer to any HTML element that has an ID attribute:

```
// point to the target element where we want to insert the date & time
var oTarget = document.getElementById('output');
```

The first statement locates the target element in the HTML page by its ID (`output`) and stores a reference to that object in variable `oTarget`. The HTML specification mandates that every element ID must be unique on the current page. Even if we accidentally use the same ID more than once on the page, the `getElementById()` method returns only the first instance. So, we know that this method will locate just one element (if it's there at all) — in this case, the span we marked up in the HTML:

```
<p>It is currently <span id="output">now</span>.</p>
```

Notice that when we call `getElementById()`, we specify the element's ID (`output`) but not its tag name (`span`). In the HTML markup we could move the attribute `id="output"` from the `span` to any other element within the page, and our script would attempt to operate on that element instead. If you think about it, that's pretty cool. It means that, even though the markup and the script have to agree about the target element's ID, the script can potentially operate regardless of which tags are used in the markup. It's that kind of flexibility that can make for some truly versatile JavaScript functions down the road.

If we omit or misspell the ID in the markup, `getElementById()` will return a null value. To cover our bases, we test for that:

```
// make sure the target is found
if (!oTarget) return;
```

Similarly to the previous if-tests, this one says, "If the variable `oTarget` doesn't exist or has a null value, return from this function." Adding such reality checks to a script will help make it bulletproof. Developing habits like this will really benefit you when you're juggling dozens of elements on a page or working on pages that have been generated by numerous people and processes.

This is an opportunity to warn the user in case something is seriously wrong — for example:

```
if (!oTarget) return alert("Warning: output element not found");
```

However, since in most cases there's really nothing that a typical web site visitor can do about a problem like this, it might be best for the script to stop quietly or find something else to do, such as notify the web master.

Deleting what's there

Our HTML document comes pre-populated with some text we want to replace:

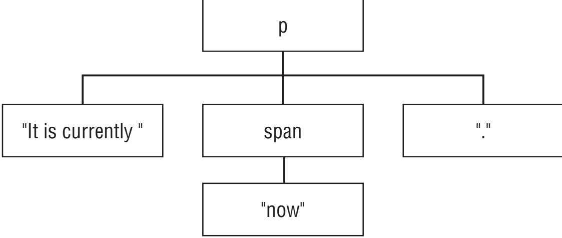
```
<p>It is currently <span id="output">now</span>.</p>
```

We want to delete the text within the `span` and then insert new content. In the DOM that represents the contents of the page, the `span` is the *parent* of everything within it. Right now there's just a single *child* — the word "now" — but we'd like this script to run successfully even if the HTML is later modified to include other markup within the target element.

Figure 5-2 shows that paragraph from a DOM perspective. The `p` element has three children: two text nodes and the span. The span has just one child: a text node containing the value "now".

FIGURE 5-2

A DOM's-eye view of the date and time paragraph before insertion.



To remove the contents of the span, execute a little loop that deletes child elements one at a time, as long as there are any to delete:

```
// delete everything inside the target
while (oTarget.firstChild)
{
   oTarget.removeChild(oTarget.firstChild);
}
```

Given our current markup, this loop will execute just once. A `while` loop will repeat as long as the expression inside the parentheses tests true. When it begins, the property `oTarget.firstChild` is evaluated: Does the object `oTarget` have a first child node? The answer is yes (`true`) because the text node "now" is there. Inside the loop, the `removeChild()` method deletes that first child. Then JavaScript loops and asks the same question again: Does `oTarget.firstChild` exist? The answer is no (`false`), so JavaScript stops processing the `while` loop and resumes execution after the closing brace.

Inserting the date and time

Finally, we create a new node in the DOM that contains the text we want, and insert that into the `span`:

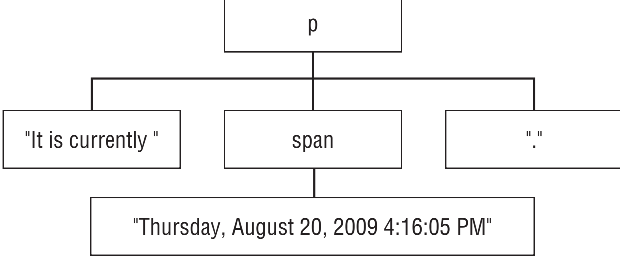
```
   // use the date-time string to create a new text node for the page
   var oNewText = document.createTextNode(sDateTime);

   // insert the new text into the span
   oTarget.appendChild(oNewText);
```

`sDateTime` is the string we generated from the `Date` object a few statements before. We call a method attached to the `document` object that creates a new text node with the date and time string as its value. At this point, the text node isn't yet part of the visible document content; it's waiting in the wings. We bring it on stage using the target element's `appendChild()` method.

The paragraph structure now looks like that shown in Figure 5-3. It's the same structural shape, but the `span` now has a new child.

FIGURE 5-3

The date and time paragraph after insertion.



Debugging

When you've saved the JavaScript file and the HTML file with its script tag, you should be able to load the HTML page into your browser and see the word "now" replaced by the current date and time.

If you can't, take the time now to track down the problem and fix it. In the process, you'll learn important details about JavaScript that will help your future coding go faster and more smoothly.

Proofread the HTML markup and JavaScript code to make sure that they match the preceding listings. The two critical points in the markup will be the `script` tag and its attributes that point to the JavaScript file, and the `span` tag and its ID that JavaScript will look for. In the JavaScript file, make sure that every statement is spelled accurately, including spaces, punctuation, and upper- and lowercase letters.

If you are unsure about the HTML, it's always a good idea to validate it in the W3C HTML Validator, available at `http://validator.w3.org/`. Select either Validate by File Upload to upload the HTML file from your computer or Validate by Direct Input to copy and paste the HTML markup from your text editor into the Validator. At this writing, all the examples in this book with an HTML5 `DOCTYPE` will register one warning: `Using experimental feature: HTML5 Conformance Checker`. This is an informational message, not an error. Your goal is to get a green banner at the top of the Validator results page that says, `This document was successfully checked as HTML5!`

Check your browser's error console to see if it's reporting an error. If so, it will likely tell you the position of the error in the script file.

For further steps, see Chapter 48, "Debugging Scripts."

Step 3: Adding style

Finally, to illustrate how a third component of a typical web page ties in, let's add a simple style sheet to the mix. Style sheets tell the browser how the page should appear — everything from the layout to font choices to background colors.

Create a new text file in your editor, enter the following style sheet script, and save it as `date-time.css`:

LISTING 5-3

CSS Code for "Date & Time": date-time.css

```
@charset "utf-8";

*
{
   margin: 0;
   font-family: sans-serif;
}
h1
{
   font-size: 10em;
   color: #DDF;
```

```
}
p
{
   position: absolute;
   top: 2.5em;
   left: 1.5em;
   font-size: 2em;
   font-weight: bold;
   color: #338;
}
#output
{
   font-style: italic;
   color: #C33;
}
```

To link this style sheet to the HTML file, add a `link` element to the HTML `head`. Bring `date-time.html` back into your text editor and add this line:

```
...
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Date &amp; Time</title>
      <link href="date-time.css" rel="stylesheet" type="text/css" media="all">
      <script type="text/javascript" src="date-time.js"></script>
   </head>
...
```

Save the style sheet, save the modified HTML file, and reload the page in your browser. It should look pretty much like Figure 5-4.

Briefly, the style sheet is a list of rendering rules to be applied to the elements of the page. The basic syntax is as follows:

```
selector { property: value; [...] }
```

Our style sheet begins with a character-encoding directive:

```
@charset "utf-8";
```

Note that this matches the character encoding for the HTML file.

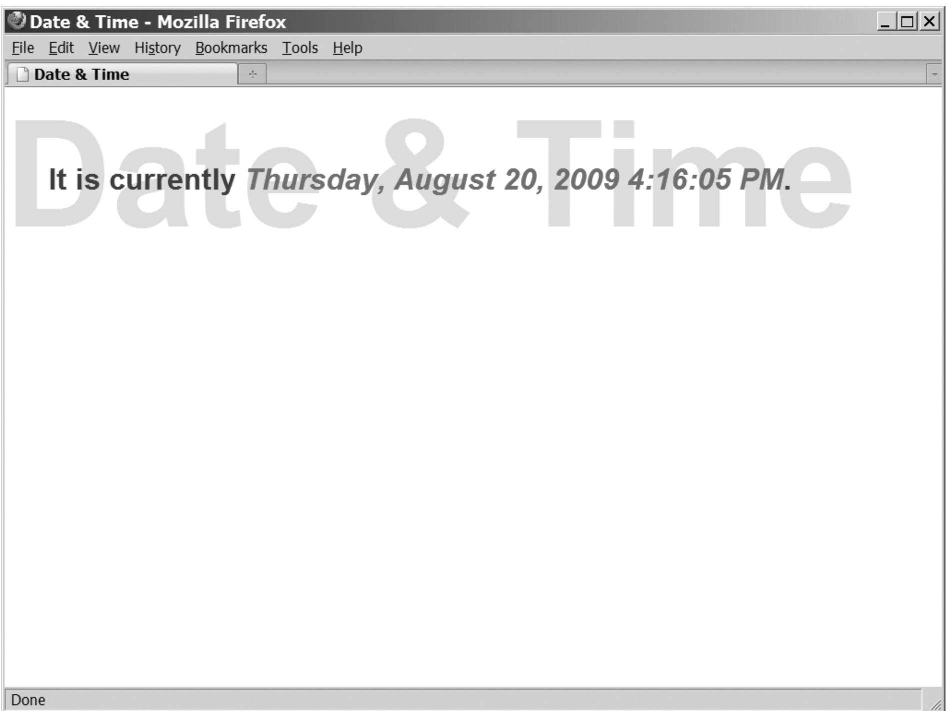
The `*` (wildcard) selector applies to all elements on the page. Here, it zeroes out default margins and sets the font-family for all text.

The `h1` selector sets the headline font size to ten times the base font size (1em = 100% of base) and sets the font color to a pale blue with the RGB (red-green-blue) hexadecimal code of DDF.

The `p` selector applies to all paragraphs on the page (there is only one). The `position`, `top`, and `left` properties allow us to position the paragraph on top of the headline. We set the font weight to bold to make sure it's readable against the pale headline, and the text color to a dark blue with RGB #338.

FIGURE 5-4

Step 3: The styled page.



The `#output` selector applies to the element with an ID of `output`. Here, we change the font-style to italic and the text color to a dark red with RGB #C33. This element also inherits properties from its parent, including the bold font-weight.

To learn more about Cascading Style Sheets (CSS) go to `www.w3.org/Style/CSS/`.

Have Some Fun

Once you get this script working properly, play around with it a bit. Move the `output` ID from one element to another. Change the text that gets inserted. Get comfortable tweaking the markup, the script, and the style sheet to get effects that you want. After each change, save the file and reload the page in your browser to check your progress. Don't be afraid to make mistakes while you're learning. You can always undo a change that goes awry. Remember that play is the best way to learn!

Exercises

1. Perform a single cut-and-paste operation on the HTML markup in Listing 5-1 to cause JavaScript to replace the entire paragraph with the date and time, not just the span. *Hint:* After your edit, the paragraph will still contain text and a `span` element.
2. With the change of markup you've just made, consider the `while` loop in JavaScript Listing 5-2. Write out each step that the loop will take in deleting the contents of the paragraph.
3. Again given the HTML edit you made in Exercise 1, decide which change(s) to the style sheet in Listing 5-3 would be necessary in order to assign the paragraph the same font and color attributes that the span had before.

CHAPTER 6

Browser and Document Objects

In this chapter, you'll see several practical applications of JavaScript and begin to see how a JavaScript-enabled browser turns familiar HTML elements into objects that your scripts control. This tutorial teaches concepts and terminology that apply to modern browsers, with special focus on standards compatibility to equip you to work with today's and tomorrow's browsers. You should study this tutorial in conjunction with any of the following browsers: Internet Explorer 5 or later (Windows or Macintosh), any Mozilla-based browser (Firefox, Netscape 7 or later, or Camino), Apple Safari, or Opera 7 or later.

IN THIS CHAPTER

What client-side scripts do

What happens when a document loads

How the browser creates objects

How scripts refer to objects

What distinguishes one object from another

Scripts Run the Show

If you have authored web pages with HTML, you are familiar with how HTML tags influence the way content is rendered on a page when viewed in the browser. As the page loads, the browser recognizes angle-bracketed tags as formatting instructions. Instructions are read from the top of the document downward, and elements defined in the HTML document appear on-screen in the same order in which they appear in the document's source code. As an author, you do a little work one time and up front — adding the tags to text content — and the browser does a lot more work every time a visitor loads the page into a browser.

Assume for a moment that one of the elements on the page is a text input field inside a form. The user is supposed to enter some text in the text field and then click the Submit button to send that information back to the web server. If that information must be an Internet email address, how do you ensure the user includes the @ symbol in the address?

One way is to have a program running on the web server — such as one written in PHP, ASP, ColdFusion, or a Common Gateway Interface (CGI) language, such as Perl — inspect the submitted form data after the user clicks the Submit button and the form information is transferred to the server. If the user omits or forgets the @ symbol, the server-side program sends the page back to the browser — but this time with an instruction to include the symbol in the

address. Nothing is wrong with this exchange, but it can mean a slight delay for the user in finding out that the address does not contain the crucial symbol. Moreover, the web server has to expend some of its resources to perform the validation and communicate back to the visitor. If the web site is a busy one, the server may try to perform hundreds of these validations at any given moment, probably slowing the response time to the user even more.

Now imagine that the document containing that text input field has some intelligence built into it that makes sure the text-field entry contains the @ symbol before ever submitting one bit (literally!) of data to the server. That kind of intelligence would have to be embedded in the document in some fashion — downloaded with the page's content so it can stand ready to jump into action when called upon. The browser must know how to run that embedded program. Some user action must start the program, perhaps when the user clicks the Submit button. If the program runs inside the browser and detects the lack of the @ symbol, an alert message should appear, to bring the problem to the user's attention. The same program should also be capable of deciding whether the actual submission can proceed or whether it should wait until a valid email address is entered in the field.

This kind of pre-submission data entry validation is but one of the practical ways JavaScript adds intelligence to an HTML document. Considering this example, you might recognize that a script must know how to look into what is typed in a text field; a script must also know how to let a submission continue and how to abort the submission. A browser capable of running JavaScript programs conveniently treats elements such as the text field as *objects*. A JavaScript script controls the action and behavior of objects, most of which you see onscreen in the browser window.

When to Use JavaScript

With so many web-oriented development tools and languages at your disposal, you should focus your client-side JavaScript efforts on tasks for which they are best suited. When faced with a web application task, we look to client-side JavaScript for help with the following requirements:

- **Data entry validation.** If form fields need to be filled out for processing on the server, we let client-side scripts prequalify the data entered by the user.
- **Serverless CGIs.** We use this term to describe processes that, were it not for JavaScript, would be programmed as CGIs on the server, yielding slower performance because of the interactivity required between the program and the user. This includes tasks such as small data collection lookup, modification of images, and generation of HTML in other frames and windows based on user input.
- **Dynamic HTML interactivity.** It's one thing to use DHTML's capabilities to position elements precisely on the page; you don't need scripting for that. But if you intend to make the content dance on the page, scripting makes that happen.
- **CGI prototyping.** Sometimes you want a server-side program to be at the root of your application because it reduces the potential incompatibilities among browser brands and versions. It may be easier to create a prototype of the CGI in client-side JavaScript. Use this opportunity to polish the user interface before implementing the application in your server-side script of choice.
- **Offloading a busy server.** If you have a highly-trafficked web site, it may be beneficial to convert frequently-used CGI processes to client-side JavaScript scripts. After a page is downloaded, the server is free to serve other visitors. Not only does this lighten server load, but users also experience quicker response from the application embedded in the page.

- **Adding life to otherwise-dead pages.** HTML by itself is pretty flat. Adding a blinking chunk of text doesn't help much; animated GIF images more often distract from, rather than contribute to, the user experience at your site. But if you can dream up ways to add some interactive zip to your page, it may engage the user and encourage a recommendation to friends or repeat visits.
- **Creating web pages that "think."** If you let your imagination soar, you may develop new, intriguing ways to make your pages appear "smart." For example, in the application Intelligent "Updated" Flags (Chapter 57 on the CD-ROM), you see how (without a server CGI or database) an HTML page can "remember" when a visitor last came to the page. Then, any items that have been updated since the last visit — regardless of the number of updates you've done to the page — are flagged for that visitor. That's the kind of subtle, thinking web page that best displays JavaScript's powers.

Note

Web pages and applications intended for public access should not rely exclusively on JavaScript to supply critical functions. Make sure that your primary data and web site functionality are accessible to visitors who have JavaScript turned off and to user agents that don't interpret JavaScript, such as search engines and mobile devices. Let your scripting enhance the experience for the majority of visitors who have JavaScript-enabled browsers, but don't let your web site be broken for the rest. ■

The Document Object Model

Before you can truly start scripting, you should have a good feel for the kinds of objects you will be scripting. A scriptable browser does a lot of the work of creating software objects that generally represent the visible objects you see in an HTML page in the browser window. Obvious objects include form controls (text boxes and buttons) and images. However, there may be other objects that aren't so obvious by looking at a page, but that make perfect sense when you consider the HTML tags used to generate a page's content — paragraph objects or frames of a frameset, for example.

To help scripts control these objects — and to help authors see some method to the madness of potentially dozens of objects on a page — the browser makers define a *document object model (DOM)*. In this context, a model is the organization of objects on the page.

Evolution of browser DOMs has caused much confusion and consternation among scripters due to a lack of compatibility across succeeding generations and brands of browsers. Fortunately, the DOM world is stabilizing around a formal specification published by the World Wide Web Consortium (W3C). Today's modern browsers continue to support some of the "old ways" of the earliest DOM because so much existing script code on the Web relies on these traditions continuing to work. (You'll see some of these in Chapter 11, "Forms and Form Elements.") But with the vast majority of browsers in use today supporting the basic W3C DOM syntax and terminology, scripters should aim toward standards compliance whenever possible.

HTML structure and the DOM

An important trend in HTML markup is applying markup solely to define the structure of a document and the context of each piece of content in the document. The days of using only HTML tags to influence the *appearance* of a chunk of text are drawing to a close. It is no longer acceptable to enclose a line of text in, say, an `<h1>` tag because you want the line to appear in the text size and weight that browsers automatically apply to text tagged in that way. An `<h1>` element has a special context

within a document's structure: a first-level heading. In today's HTML world, if you wish to display a stand-alone line of text with a particular style, the text would likely be in a simple paragraph (<p>) tag; the precise look of that paragraph would be under the control of a Cascading Style Sheet (CSS) rule. Current practice even frowns upon the application of <b> and <i> tags to assign boldface and italic styles to a span of text. Instead, surround the text with a contextual tag (such as the <em> element to signify emphasis), and define the CSS style you wish applied to any emphasized text in the document.

The result of applying strict structural design to your HTML tagging is a document that has a well-defined *hierarchy* of elements based on their nesting within one another. For example, a well-formed HTML 4.01 document has the following minimum elements:

- Document type declaration
- HTML
  - Head
    - Character encoding
    - Title
  - Body
    - Block-level content

The HTML markup for this empty document might look like this:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
   "http://www.w3.org/TR/html4/strict.dtd">
<html>
   <head>
      <meta http-equiv="Content-Type" content="text/html;
charset=utf-8">
      <title></title>
   </head>
   <body>
      <p></p>
   </body>
</html>
```

Viewed as a family tree in which every child has just one parent, as shown in Figure 6-1, an HTML document must always have two children, `DOCTYPE` and `html`.

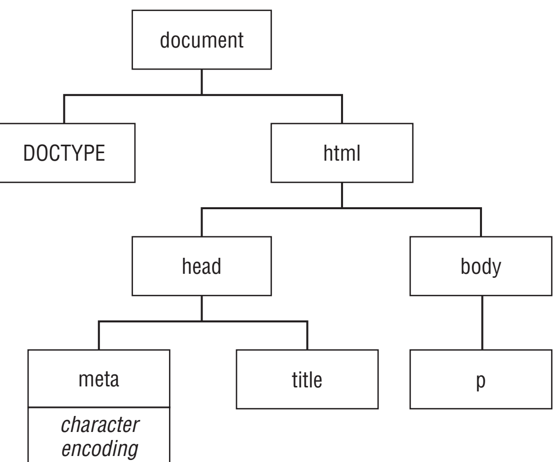
The `DOCTYPE` tells the browser which version of markup is being used, critical information for markup validation and the interpretation of style sheet rules. `html` has two required children, `head` and `body`. The `head` must minimally contain a `meta` element that specifies the character encoding and a `title`, and may also contain a variety of other children such as script and style sheet links. Elements in the `head` ordinarily don't appear as objects on the document page, but tell the browser how to configure the document, and can radically affect its appearance and behavior. The `body` contains the text, images, and other elements that we ordinarily think of as being page content; it must minimally contain at least one block-level element such as a headline, paragraph, or division.

Although we're using the much simpler `DOCTYPE` tag for HTML5 in this book,

```
<!DOCTYPE html>
```

FIGURE 6-1

Element hierarchy of an empty HTML 4.01 document.



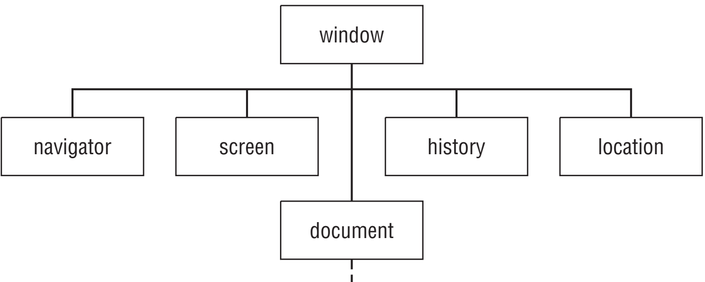
we adhere to the markup requirements for HTML 4.01 so that our HTML markup will validate, to the greatest extent possible, as both HTML 4.01 and HTML5 in order to provide valid examples for whichever branch of markup you choose to pursue.

The DOM in a browser window

As its name implies, the formal DOM focuses primarily on the HTML document and the content nested inside it. From a practical standpoint, however, scripters often need to control the environment that contains the document: the window. The `window` object is the top of the hierarchy that browser scripts work with. The basic structure of the object model in modern browsers (given an empty HTML document) is shown in Figure 6-2.

FIGURE 6-2

Basic object model for all modern browsers.



To give you a sense of the relationships among these top-level objects, the following describes their respective roles:

- `window` **object.** At the very top of the hierarchy is the window. This object represents the content area of the browser window where HTML documents appear. In a multiple-frame environment, each frame is also a window (but don't concern yourself with this just yet). Because all document action takes place inside the window, the window is the outermost element of the object hierarchy. Its physical borders contain the document.
- `navigator` **object.** This is the closest your scripts come to accessing the browser program, primarily to read the browser's claims of brand and version. This object is read-only, protecting the browser from inappropriate manipulation by rogue scripts; however, as you'll see, `navigator` can't be relied on to report the actual make, model, and version of the current browser.
- `screen` **object.** This is another read-only object that lets scripts learn about the physical environment in which the browser is running. For example, this object reveals the numbers of pixels high and wide available in the monitor.
- `history` **object.** Although the browser maintains internal details about its recent history (such as the list available under the Back button), scripts have no access to the details. At most, this object assists a script in simulating a click of the Back or Forward button.
- `location` **object.** This object is the primary avenue to loading a different page into the current window or frame. URL information about the window is available under very controlled circumstances so that scripts cannot track access to other web sites.
- `document` **object.** Each HTML document that gets loaded into a window becomes a `document` object. The `document` object contains the content that you are likely to script. Except for the `html`, `head`, and `body` element objects that are found in every HTML document, the precise makeup and structure of the element object hierarchy of the document depend on the content you put into the document.

When a Document Loads

Programming languages such as JavaScript are convenient intermediaries between your mental image of how a program works and the true inner workings of the computer. Inside the machine, every word of a program code listing influences the transformation and movement of bits (the legendary 1s and 0s of the computer's binary universe) from one RAM storage slot to another. Languages and object models are inside the computer (or, in the case of JavaScript and the DOM, inside the browser's area of the computer) to make it easier for programmers to visualize how a program works and what its results will be. The relationship reminds us of knowing how to drive an automobile from point A to point B without knowing exactly how an internal-combustion engine, steering linkages, and all that other internal "stuff" works. By controlling high-level objects such as the ignition key, gearshift, gas pedal, brake, and steering wheel, we can get the results we need.

Of course, programming is not exactly like driving a car with an automatic transmission. Even scripting requires the equivalent of opening the hood and perhaps knowing how to check the transmission fluid or change the oil. Therefore, now it's time to open the hood and watch what happens to a document's object model as a page loads into the browser.

A simple document

Figure 6-3 shows the HTML and corresponding object model for a document that we'll be adding to in a moment. The figure shows only the `document` object portion; the `window` object and its

other top-level objects (including the `document` object) are always there, even for an empty document. When this page loads, the browser maintains in its memory a map of the objects generated by the HTML tags in the document. The `body` must contain block-level content, but we haven't added that yet.

FIGURE 6-3

Object map of an empty document.

```
<!DOCTYPE ...>
 <html>
  <head>
   <meta ...>
   <title></title>
  </head>
  <body>
   ...
  </body>
</html>
```

document
DOCTYPE
html
head
body
meta
character encoding
title

Add a paragraph element

Now, we modify the HTML file to include an empty paragraph element and reload the document. Figure 6-4 shows what happens to both the HTML and the object map, as constructed by the browser (changes shown in boldface). Even though no content appears in the paragraph, the `<p>` tags are enough to tell the browser to create that `p` element object. Also note that the `p` element object is contained by the `body` element object in the hierarchy of objects in the current map. In other words, the `p` element object is a *child* of the `body` element object. The object hierarchy matches the HTML tag containment hierarchy.

Add paragraph text

We modify and reload the HTML file again, this time inserting the text of the paragraph between the element's start and end tags, as shown in Figure 6-5. A run of text extending between tags is a special kind of object in the DOM called a *text node*. A text node always has an element acting as its container. Applying the official genealogy metaphor to this structure, this text node is a child of its parent `p` element. We now have a branch of the document object tree that runs several generations: `document` ➪ `html` ➪ `body` ➪ `p` ➪ text node.

Make a new element

The last modification we make to the file is to wrap a portion of the paragraph text in an `<em>` tag to signify emphasis for the enclosed text. This insertion has a large effect on the hierarchy of the `p` element object, as shown in Figure 6-6. The `p` element goes from having a single (text node) child to having three children: two text nodes with an element between them. In the W3C DOM, a text node

cannot have any children and therefore cannot contain an element object. The bit of the text node now inside the `em` element is no longer a child of the `p` element, but a child of the `em` element. That text node is now a grandchild of the `p` element object.

Now that you see how objects are created in memory in response to HTML tags, the next step is to figure out how scripts can communicate with these objects. After all, scripting is mostly about controlling these objects.

FIGURE 6-4

Adding an empty paragraph element.

```
<!DOCTYPE ...>
 <html>
  <head>
   <meta ...>
   <title></title>
  </head>
  <body>
   <p></p>
  </body>
</html>
```

document
DOCTYPE
html
head
body
meta
character encoding
title
p

FIGURE 6-5

Adding a text node to the p element object.

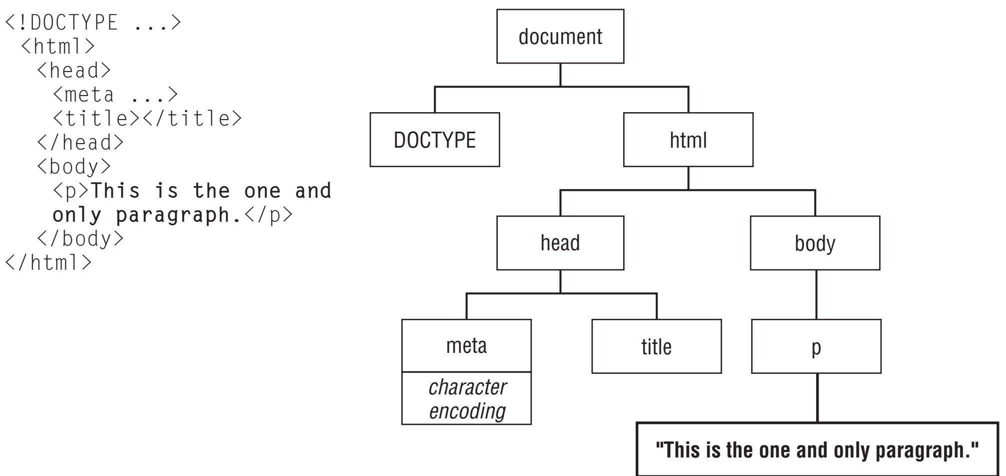
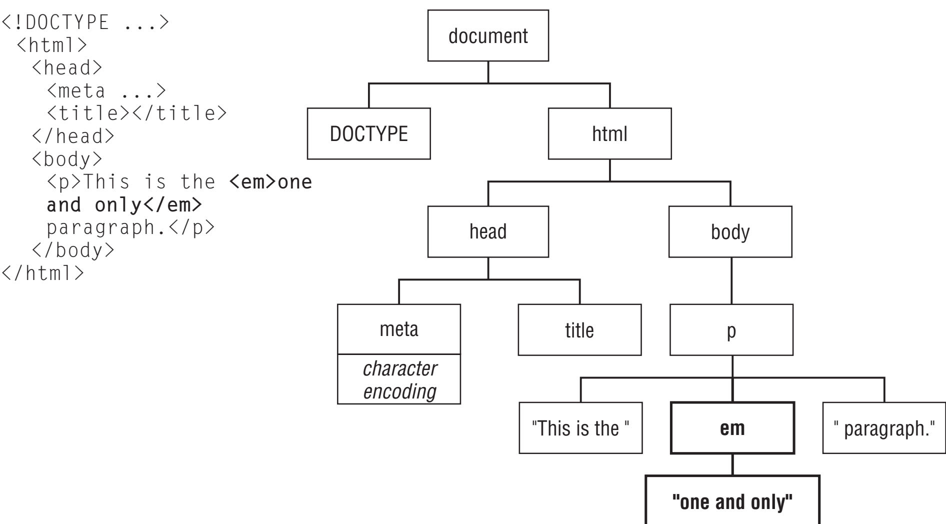


FIGURE 6-6

Inserting an element into a text node.



Object References

After a document is loaded into the browser, all of its objects are safely stored in memory in the containment hierarchy structure specified by the browser's DOM. For a script to control one of those objects, there must be a way to communicate with an object and find out something about it (such as "Hey, Mr. Text Field, what did the user type?"). To let your scripts talk to an object, you need a way to refer to that object. That is precisely what an *object reference* in a script does for the browser.

Object naming

The biggest aid in creating script references to objects is assigning a name to every scriptable object in your HTML. In the W3C DOM (and current HTML specification), the way to assign a name to an element is by way of the `id` attribute. This attribute is optional, but if you plan to use scripts to access an element in the page, it is most convenient to assign a name to that element's `id` attribute directly in the HTML code. Here are some examples of `id` attributes added to typical tags:

```
<p id="firstParagraph">

<img id="logo-1" src="images/logo.jpg" alt="Corporate Logo">

<div class="draggable" id="puzzle_piece">
```

The only rules about object IDs (also called *identifiers*) are that they:

- May not contain spaces
- May not contain punctuation except for the underscore, hyphen, period, and colon characters

- Must be inside quotes when assigned to the `id` attribute
- Must not start with a numeric character
- May not occur more than once in the same document

Think of assigning IDs as the same way as sticking name tags on everyone attending a conference meeting. To find a particular conference attendee whose name you know, you could wait at the entrance and scan each name tag until you find the name you're looking for, or you could bump around the attendees at random in the hope that you'll find a known name. But it would be more efficient if you had a way to target an attendee by name immediately — such as broadcasting the name on the public address system to the whole crowd.

Referencing a particular object

The W3C DOM provides that kind of instant access to any named element in the document. Here is the syntax you will use frequently in your browser scripting:

```
window.document.getElementById("elementID")
```

You substitute the ID of the element you wish to reference for `elementID`. For example, if you want to reference the paragraph element whose ID is `firstParagraph`, the reference would be

```
window.document.getElementById("firstParagraph")
```

JavaScript allows us to shorten this by omitting the `window` reference, so we often use this briefer form:

```
document.getElementById("firstParagraph")
```

Be careful! JavaScript is case sensitive. Be sure that you use uppercase for the three uppercase letters in the command and a lowercase *d* at the end, and that you spell the ID itself accurately as well.

The `getElementById()` command belongs to the `document` object, meaning that the entire document's collection of elements is subject to this instantaneous search for a matching ID. The dot — a traditional period character — is the JavaScript way of indicating that the item to the left of the dot (the `document` object here) has the item to the right of the dot (`getElementById()` here) as a resource to call upon whenever needed. Each type of object has a list of such resources, as you'll see in a moment (and as summarized in Appendix A).

id and name Attributes

Prior to the HTML 4.0 specification's introduction of the `id` attribute, scripts could access a handful of elements that also supported the `name` attribute. Elements supporting the `name` attribute are predominantly related to forms, images, and frames. You will see how `name` attributes work in forms in Chapter 11, ''Forms and Form Elements.'' In fact, most browsers still require the `name` attribute for forms and form controls (text fields, buttons, and select lists) for their data to be submitted to a server. It is permissible to assign the same identifier to both the `id` and `name` attributes of an element, and, in fact, in XHTML it is recommended that an `id` and `name` assigned to the same element have the same value. For example:

```
<input type="text" id="firstname" name="firstname" />
```

Node Terminology

W3C DOM terminology uses metaphors to assist programmers in visualizing the containment hierarchy of a document and its content. One concept you should grasp early in your learning is that of a *node;* the other concept is the family relationship among objects in a document.

About nodes

Although the English dictionary contains numerous definitions of *node,* the one that comes closest to its application in the W3C DOM implies a knob or bump on a tree branch. Such nodules on a branch usually lead to one of two things: a leaf or another branch. A leaf is a dead end in that no further branches emanate from the leaf; the branch kind of node leads to a new branch that can itself have further nodes, whether they be leaves or more branches. When you define the structure of an HTML document, you also define a node structure (also called a *node tree*) in which the placement of branches and leaves depends entirely on your HTML elements and text content.

In the W3C DOM, the fundamental building block is a simple, generic node. But inside an HTML document, we work with special kinds of nodes that are tailored to HTML documents. The two types of nodes that scripts touch most often are *element nodes* and *text nodes*. These node types correspond exactly to HTML elements and the text that goes between an element's start and end tags. You've been working with element and text nodes in your HTML authoring, and you may not have even known it.

Look again at the simple document we've just assembled, along with its containment hierarchy diagram in Figure 6-7. All of the boxes representing HTML elements (`html`, `head`, `body`, `p`, and `em`) are element nodes; the three boxes containing the actual text that appears in the rendered document are text nodes. You saw in the transition from one long text node (Figure 6-5) to the insertion of the `em` element (Figure 6-6) that the long text node divided into three pieces. Two text node pieces stayed in the same position in the hierarchy relative to the containing `p` element. We inserted the new `em` element into the tree between the two text nodes and shifted the third text node one level away from the `p` element.

Parents and children

Looking more closely at the p element and its content in Figure 6-7, you can see that element has three child nodes. The first and last are of the text node type, whereas the middle one is an element node. When an element contains multiple child nodes, the sequence of child nodes is entirely dependent upon the HTML source code order. Thus, the first child node of the p element is the text node containing the text "This is the ". The em element has a single child text node as its sole descendant.

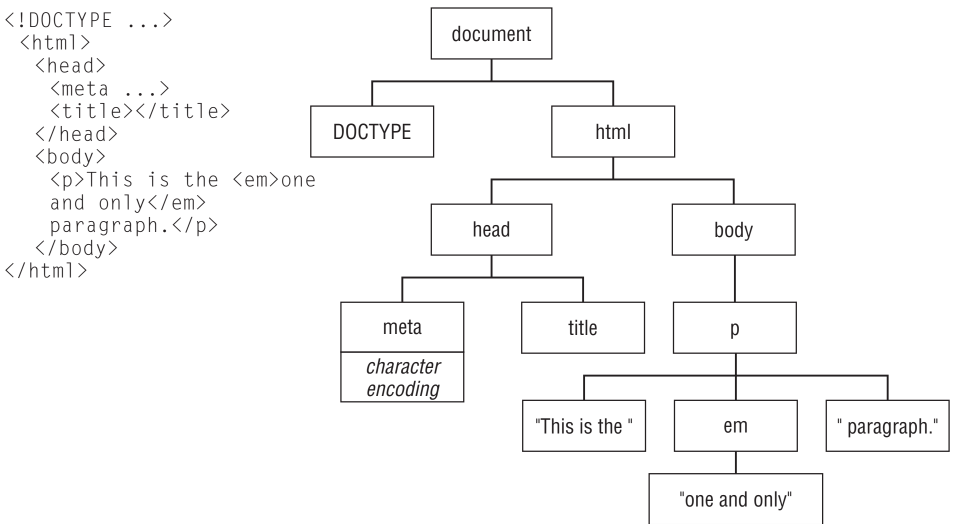
Element node children are not always text nodes; neither do branches always end in text nodes. In Figure 6-7, the `html` element has two child nodes, both of which are element nodes; the `body` element has one child node, the `p` element. A tag in the HTML indicates an element node, whether or not it has any child nodes. By contrast, a text node can never contain another node; it's one of those dead-end leaf type of nodes.

Notice that a child node is always contained by one element node. That container is the *parent* node of its child or children. For example, from the point of view of the `em` element node, it has both one child (a text node) and one parent (the `p` element node). A fair amount of W3C DOM terminology (which you'll meet in Chapter 25, "Document Object Model Essentials") concerns itself with assisting scripts starting at any point in a document hierarchy and obtaining a reference to a related node if necessary. For instance, if a Dynamic HTML script wants to modify the text inside the `em` element

of Figure 6-7, it typically would do so by starting with a reference to the `em` element via the `document.getElementById()` command (assuming that the `em` element has an ID assigned to it) and then modifying the element's child node.

FIGURE 6-7

A simple HTML document node tree.



In case you're wondering, the `document` object at the top of the node tree is itself a node. Its place in the tree is special and is called simply the *document node*. Each loaded HTML document contains a single document node, and that node becomes the scripter's gateway to the rest of the document's nodes. It's no accident that the syntax for referencing an element node — `document.getElementById()` — begins with a reference to the `document` object.

What Defines an Object?

When an HTML tag defines an object in the source code, the browser creates a slot for that object in memory as the page loads. But an object is far more complex internally than, say, a mere number stored in memory. The purpose of an object is to represent some thing. In the browser and its DOM, the most common objects are those that correspond to elements, such as a text input form field, a table element, or the entire rendered body of the document. Outside the pared-down world of the DOM, an object can also represent abstract entities, such as a calendar program's appointment entry or a layer of graphical shapes in a drawing program. It is common for your browser scripts to work with both DOM objects and abstract objects of your own design.

Every type of DOM object is unique in some way, even if two or more objects look identical to you in the browser. Three very important facets of an object define what it is: what it "looks" like, how

it behaves, and how scripts control it. Those three facets are properties, methods, and events (also known as handlers). They will play key roles in your future DOM scripting efforts. The Quick Reference in Appendix A summarizes the properties, methods, and events for each object in the object models implemented in the various browser generations.

Properties

Any physical object you hold in your hand has a collection of characteristics that defines it. A coin, for example, has shape, diameter, thickness, color, weight, embossed images on each side — and any number of other attributes that distinguish it from, say, a feather. Each of those features is called a *property*. Each property has a value of some kind attached to it (even if the value is empty or null). For example, the `shape` property of a coin might be *circle* — in this case, a text value. By contrast, the `denomination` property would most likely be a numeric value.

You may not have known it, but if you've written HTML for use in a scriptable browser, you have set object properties without writing one iota of JavaScript. Tag attributes are the most common way to set an HTML element object's initial properties. For example, the following HTML tag defines an `input` element object that assigns four property values:

```
<input type="button" id="clicker" name="clicker" value="Hit Me...">
```

In JavaScript parlance, then, the `type` property holds the word *button;* the `id` and `name` properties hold the same word, *clicker;* and the `value` property is the text that appears on the button label, *Hit Me* ... . In truth, a button input element has more properties than just these, but you don't have to set every property for every object. Most properties have *default values* that are automatically assigned if nothing special is set in the HTML, or later from a script.

The contents of some properties can change after a document has loaded and the user interacts with the page. Consider the following text input tag:

```
<input type="text" id="donation" name="donation" value="$0.00">
```

The `id` and `name` properties of this object are the same word: `donation`. When the page loads, the text of the `value` attribute setting is placed in the text field — the automatic behavior of an HTML text field when the `value` attribute is specified. But if a user enters some other text into the text field, the `value` property changes — not in the HTML, but in the memory copy of the object model that the browser maintains. Therefore, if a script queries the text field about the content of the `value` property, the browser yields the current setting of the property — which isn't necessarily the one specified by the HTML.

To gain access to an object's property, you use the same kind of dot-notation addressing scheme you saw earlier for objects. A property is a resource belonging to its object, so the reference to it consists of the reference to the object plus one more extension naming the property. Therefore, for the `button` and `text` object tags just shown, references to various properties are

```
document.getElementById("clicker").name
document.getElementById("clicker").value
document.getElementById("entry").value
```

Tip

JavaScript will fail or "abort" if it tries to reference the property of a non-existent element. Therefore, a safer, more bulletproof way of referencing these properties is to first test for the existence of the object:

```
var oClicker = document.getElementById("clicker");
   if (oClicker) var sName = oClicker.name;
```

or

```
var oClicker = document.getElementById("clicker");
   if (!oClicker) return;
var sName = oClicker.name;
```
■

You might wonder what happened to the `window` part of the reference. It turns out that there can be only one document contained in a window, so references to objects inside the document can omit the `window` portion and start the reference with `document`. You cannot omit the `document` object from the reference, however.

Methods

If a property is like a descriptive adjective for an object, a *method* is a verb. A method is all about action related to the object. A method does something either to the object or with the object that affects other parts of a script or document. Methods are commands of a sort whose behaviors are tied to a particular object.

Internet Explorer References

Before the W3C DOM came into existence, Microsoft had created its own way of referencing element objects by way of their `id` attributes. You will find many instances of this syntax in existing code that has been written only for Internet Explorer 4 or later. The syntax uses a construction called `document.all`. Although there are a few different ways to use this construction, the most commonly applied way is to continue the dot notation to include the ID of the element. For example, if a paragraph element's ID is `myParagraph`, the IE-only reference syntax is

```
document.all.myParagraph
```

You can also omit the lead-in parts of the reference and simply refer to the ID of the element:

```
myParagraph
```

Be aware, however, that none of these approaches is supported in the W3C DOM standard. Both the IE-specific and W3C DOM reference syntax styles are implemented in IE5 or later. Also the `document.all.elementId` syntax cannot tolerate some of the valid HTML ID characters, such as hyphen, colon, and period, because JavaScript would interpret them as its own syntax.

Although it's important to be familiar with `document.all` in order to make sense of legacy code, you should stick to W3C DOM methods such as `getElementById()` in your own code to be compatible with forward-looking modern browsers.

Note

Note that the document.all syntax cannot tolerate any hyphens, colons, or periods in the ID. Although these are valid HTML IDs

```
my-paragraph
my.paragraph
my:paragraph
```

the resulting document.all expressions would cause JavaScript to give erroneous results or crash:

```
document.all.my-paragraph
document.all.my.paragraph
document.all.my:paragraph
```

Using the modern standard getElementById() is safer because the ID is always enclosed in quotation marks. ■

An object can have any number of methods associated with it (including none at all). To set a method into motion (usually called *invoking a method*), a JavaScript statement must include a reference to it, via its object, with a pair of parentheses after the method name, as in the following examples:

```
var oForm = document.getElementById("orderForm");
oForm.submit();

var oEntry = document.getElementById("entry");
oEntry.focus();
```

The first is a scripted way of sending a form (named `orderForm`) to a server. The second gives focus to a text field named `entry`.

Sometimes a method requires that you send additional information with it so that it can do its job. Each chunk of information passed with the method is called a *parameter* or *argument* (you can use the terms interchangeably). The `document.getElementById()` method is one that requires a parameter: the identifier of the element object to be addressed for further action. This method's parameter must be in a format consisting of straight text, signified by the quotes around the identifier.

Some methods require more than one parameter. If so, the multiple parameters are separated by commas. For example, modern browsers support a `window` object method that moves the window to a particular *coordinate point* onscreen. A coordinate point is defined by two numbers that indicate the number of pixels from the left and top edges of the screen where the top-left corner of the window should be. To move the browser window to a spot 50 pixels from the left and 100 pixels from the top, the method is

```
window.moveTo(50,100)
```

As you learn more about the details of JavaScript and the document objects you can script, pay close attention to the range of methods defined for each object. They reveal a lot about what an object is capable of doing under script control.

Events

One last characteristic of a DOM object is the *event*. Events are actions that take place in a document, usually as the result of user activity. Common examples of user actions that trigger events include

clicking a button or typing a character in a text field. Some events, such as the act of loading a document into the browser window or experiencing a network error while an image loads, are not so obvious.

Almost every DOM object in a document receives events of one kind or another — summarized for your convenience in the Quick Reference of Appendix A. Your job as scripter is to write the code that tells an element object to perform an action whenever the element receives a particular type of event. The action is simply executing some additional JavaScript code.

A simple way to begin learning about events is to apply one to an HTML element. Listing 4-1 shows a very simple document that displays a single button and a script that applies one event handler to the button. The event's name consists of the type of event (for example, `click`) preceded by the preposition *on* (as in, "on receiving the `click` event ... ": `onclick`).

LISTING 6-1

A Simple Button with an Event Handler

HTML: jsb-06-01.html

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>A Simple Button with an Event Handler</title>
      <script type="text/javascript" src="jsb-06-01.js"></script>
   </head>
   <body>
      <form action="">
         <div>
            <button id="clicker">Click me</button>
         </div>
      </form>
   </body>
</html>
```

JavaScript: jsb-06-01.js

```
// tell the browser to run this script when the page has finished loading
window.onload = applyBehavior;

// apply behavior to the button
function applyBehavior()
{
      // ensure a DOM-aware user agent
      if (document.getElementById)
      {
         // point to the button
         var oButton = document.getElementById('clicker');

            // if it exists, apply behavior
            if (oButton)
            {
```

```
                oButton.onclick = behave;
            }
        }
}

// what to do when the button is clicked
function behave(evt)
{
   alert('Ouch!');
}
```

When the browser loads the HTML page, it also loads the linked JavaScript file, compiles it, and begins to execute its instructions. In this case, the only instruction it will actually execute immediately is the `window.onload` statement because all the other statements are inside functions and won't execute until the functions are called by name. Only when the HTML page has finished loading (and when its DOM has been completely built) will our function `applyBehavior()` run. If the JavaScript interpreter that's running is aware of the DOM method `getElementById()`, the function uses it to look up the element with the ID of `clicker`. If found, it applies the onclick behavior to the button, telling it to execute the function `behave()` when clicked. When the user clicks the button, the `behave()` function issues an alert (a popup dialog box) that says, "Ouch!"

Cross-Reference

You will learn about other ways to connect scripting instructions to events in Chapter 25, "Document Object Model Essentials," and Chapter 32, "Event Objects." ■

Exercises

1. Which of the following applications are well suited to client-side JavaScript? Why or why not?
   a. Product catalog page that lets visitors view the product in five different colors
   b. A counter that displays the total number of visitors to the current page
   c. Chat room
   d. Graphical Fahrenheit-to-Celsius temperature calculator
   e. All of the above
   f. None of the above
2. Which of the following element IDs are valid in HTML? For each one that is invalid, explain why.
   a. `lastName`
   b. `company_name`
   c. `1stLineAddress`
   d. `zip code`
   e. `today's_date`
   f. `now:you-hear.this`

3. Using the diagram from Figure 6-7 for reference, draw a diagram of the object model containment hierarchy that the browser would create in its memory for the following HTML. Write the script reference to the first paragraph element using W3C DOM syntax.

```
<!DOCTYPE html>
<html>
<head>
<meta http-equiv="content-type" content="text/html;charset=utf-8">
<title>Search Form</title>
</head>
<body>
<p id="logoPar"><img src="images/logo.jpg" height="90" width="300"
     alt="Logo"></p>
<div id="searchForm">
<form action="cgi-bin/search.pl" method="post">
<div>
<label for="searchText">Search for:</label>
<input type="text" id="searchText" name="searchText">
<input type="submit" value="Search">
</div>
</form>
</div>
</body>
</html>
```

4. Describe at least two characteristics that a text node and an element node have in common; describe at least two characteristics that distinguish a text node from an element node.

5. Write the HTML and JavaScript for a button input element named `Hi`, whose visible label reads `Howdy`, and whose action upon being clicked displays an alert dialog box that says, `Hello to you, too!`

CHAPTER

Scripts and HTML Documents

IN THIS CHAPTER

Where to place scripts in HTML documents

What a JavaScript statement is

What makes a script run

Viewing script errors

Chapter 4, "JavaScript Essentials," covered many of the basics of how to combine JavaScript with HTML documents. This chapter's tutorial reviews how scripts are linked to HTML documents and what comprises a script statement. You also see how script statements can run when the document loads or in response to user action. Finally, you find out where script error information may be hiding.

Connecting Scripts to Documents

We use the `script` element to tell the browser to treat a segment of text as script and not as HTML. Whether our script is an external file linked to the HTML document or embedded directly in the page itself, the `<script>...</script>` tag set encapsulates the script.

Depending on the browser, the `<script>` tag has a variety of attributes you can set that govern the script. One attribute, `type`, advises the browser to treat the code within the tag as JavaScript. Some other browsers accept additional languages (such as Microsoft's VBScript in Windows versions of Internet Explorer). The following setting is one that all modern scriptable browsers accept:

```
<script type="text/javascript" ...>...<script>
```

Be sure to include the ending tag for the script. Your JavaScript program code goes either into an external file specified in the `src` attribute:

```
<script type="text/javascript" src="example.js"><script>
```

or between the two tags:

```
<script  type="text/javascript">
  one or more lines of JavaScript code here
</script>
```

If you forget the closing script tag, the script may not run properly, and the HTML elsewhere in the page may look strange.

The Old language Attribute

Another `<script>` tag attribute, `language`, used to be the way to specify the scripting language for the enclosed code. That attribute allowed scripters to specify the language version. For example, if the scripts included code that required JavaScript syntax available only in version 4 browsers (which implemented JavaScript version 1.2), the `<script>` tag used to be written as follows:

```
<script language="JavaScript1.2">...</script>
```

The `language` attribute was never part of the HTML 4.0 specification and is now falling out of favor. If W3C validation is one of your development concerns, you should be aware that the language attribute does not validate in strict versions of HTML 4.01 or XHTML 1.0. Older browsers that do not know the `type` attribute automatically default to JavaScript anyway. Use only the `type` attribute.

Tag positions

Where do these tags go within a document? The answer is anywhere they're needed in the document. Most of the time, it makes sense to include the tags nested within the `<head>...</head>` tag set; other times, it is essential that you drop the script into a very specific location in the `<body>...</body>` section.

In the following three listings, we demonstrate — with the help of a skeletal HTML document — some of the possibilities of `<script>` tag placement. Later in this lesson, you see why scripts may need to go in different places within a page, depending on the scripting requirements. In each example, we show two `script` tags – one a link to an external script and one for internal embedded scripting – just to show you how they both appear. In practice, it's most likely that you will consistently link only to external scripts.

Listing 7-1 shows the outline of what may be the most common position of a `<script>` tag set in a document: in the `<head>` tag section. Typically, the `head` is a place for tags that influence non-content settings for the page — so-called HTML *directive* elements, such as `<meta>` tags and the document title. It turns out that this is also a convenient place to plant scripts that are called in response to a page load or a user action.

LISTING 7-1

Scripts in the head

```
<html>
   <head>
      <title>A Document</title>
      <script type="text/javascript" src="example.js"></script>
      <script  type="text/javascript">
         //script statement(s) here
         ...
```

```
      </script>
   </head>
   <body>
   </body>
</html>
```

On the other hand, if you need a script to run as the page loads so that the script generates content in the page, the script goes in the `<body>` portion of the document, as shown in Listing 7-2.

LISTING 7-2

A Script in the body

```
<html>
   <head>
      <title>A Document</title>
   </head>
   <body>
      <script type="text/javascript" src="example.js"></script>
      <script  type="text/javascript">
         //script statement(s) here
         ...
      </script>
   </body>
</html>
```

It's also good to know that you can place an unlimited number of `<script>` tag sets in a document. For example, Listing 7-3 shows a script in both the `head` and `body` portions. This document needs the `body` script, perhaps to create some dynamic content as the page loads, but the document also contains a button that needs a script to run later. That script is stored in the `head` portion.

LISTING 7-3

Scripts in the head and body

```
<html>
   <head>
      <title>A Document</title>
      <script type="text/javascript" src="example.js"></script>
      <script  type="text/javascript">
         //script statement(s) here
         ...
      </script>
   </head>
   <body>
      <script type="text/javascript" src="example.js"></script>
```

continued

LISTING 7-3 *(continued)*

```
      <script  type="text/javascript">
         //script statement(s) here
         ...
      </script>
   </body>
</html>
```

Handling non-JavaScript browsers and XHTML

Only browsers that include JavaScript know to interpret the lines of code between the `<script>...</script>` tag pair as script statements rather than HTML text for display in the browser. This means that a pre-JavaScript browser or a simplified browser in a cell phone would not only ignore the tags, but also treat the JavaScript code as page content. The results can be disastrous to a page.

On the other hand, you don't have to worry about non-JavaScript browsers trying to execute externally linked scripts. That's one of the advantages of linking rather than embedding your scripts. The problem only arises when JavaScript code is embedded in the HTML document.

You can reduce the risk of older, non-JavaScript browsers displaying the script lines by enclosing the script lines between HTML comment symbols, as shown in Listing 7-4. Most nonscriptable browsers ignore the content between the `<!--` and `-->` comment tags, whereas scriptable browsers ignore the opening comment symbol when it appears inside a `<script>` tag set.

LISTING 7-4

Hiding Scripts from Most Old Browsers

```
<script  type="text/javascript">
<!--
   //script statement(s) here
   ...
// -->
</script>
```

The odd construction right before the ending script tag needs a brief explanation. The two forward slashes are a JavaScript comment symbol. This symbol is necessary because JavaScript otherwise tries to interpret the components of the ending HTML symbol (`-->`). Therefore, while the forward slashes tell JavaScript to skip the line entirely, a non-scriptable browser simply treats those slash characters as part of the entire HTML comment to be ignored.

Although it's no longer really necessary to hide JavaScript in this way from older browsers, there are instances when a different type of hiding may be required. XML is more frequently being used to feed content to the browser, and often this is done using XHTML. In the XML world, all special characters need to be enclosed in a `CDATA` section, or else the file may be parsed incorrectly; at the very least, it will fail validation. Again, the trick is to enclose your script, but this time it will look like

Listing 7-5. You'll notice again that immediately before the closing script tag the JavaScript comment hides the closing of the CDATA section from JavaScript itself.

LISTING 7-5

Hiding Scripts From XML Parsers

```
<script  type="text/javascript">
<![CDATA[
   //script statement(s) here
   ...
//]]>
</script>
```

Despite the fact that these techniques are often called *script hiding,* they do not conceal the scripts from human readers. All client-side JavaScript scripts are part of, or accompany, the HTML document, and they download to the browser just like all the other assets that make up the page. You can view the JavaScript source as easily as you can view the HTML document source. Do not be fooled into thinking that you can hide your scripts from prying eyes. Some developers *obfuscate* their scripts by removing all carriage returns and using nonsensical variable and function names, but this only slows down (and doesn't stop) any inquisitive visitor from reading and understanding the code.

Since it's not possible to truly hide your JavaScript programming from the public, you might as well flaunt it: sign the scripts you're proud of, include a copyright or Creative Commons notice of authorship in script comments, and encourage people who admire the script to come to you for more.

JavaScript Statements

Virtually every line of code in an externally linked file or that sits within a `<script>...</script>` tag pair is a JavaScript *statement* (except for HTML comment tags). To be compatible with the habits of experienced programmers, JavaScript accepts a semicolon at the end of every statement (the computer equivalent of a period at the end of a sentence). This semicolon is optional, but we strongly recommend that you use it consistently to avoid ambiguity. The carriage return at the end of a statement suffices for JavaScript to know that the statement has ended. It is possible, however, that in the future the semicolon will be required, so it's a good idea to get into the semicolon habit now.

A statement must be in the script for a purpose. Therefore, every statement does something relevant to the script. The kinds of things that statements do are

- Define or initialize a variable
- Assign a value to a property or variable
- Change the value of a property or variable
- Define or invoke an object's method
- Define or invoke a function routine
- Make a decision

If you don't yet know what all of these things mean, don't worry; you will, by the end of this tutorial. The point we want to stress is that each statement contributes to the scripts you write. The only statement that doesn't perform any explicit action is the *comment*. A pair of forward slashes (no space between them) is the most common way to include a single-line comment in a script:

```
// this is a one-line comment

var a = b;   // this comment shares a line with an active statement
```

Multiple-line comments can be enclosed in slash-asterisks:

```
/*
  Any number of lines are comments if they are
  bracketed by slash-asterisks
*/
```

You add comments to a script for the benefit of yourself and other human readers. They usually explain in plain language what a statement or group of statements does. The purpose of including comments is to remind you how your script works six months from now, or to help out another developer who needs to read your code.

When Script Statements Execute

Now that you know where scripts go in a document, it's time to look at when they run. Depending on what you need a script to do, you have four choices for determining when a script runs:

- While a document loads
- Immediately after a document loads
- In response to user action
- When called upon by other script statements

The determining factor is how the script statements are positioned in a document.

While a document loads: immediate execution

Listing 7-5 is a variation of your first script from Chapter 5, "Your First JavaScript Script." In this version, the script writes the current date and time to the page while the page loads. The `document.write()` method is a common way to write dynamic content to the page *during* loading. We call the kinds of statements that run as the page loads *immediate statements*. Care must be taken when you code scripts using `document.write()`. Once the page completes loading, any further `document.write()` statements create a new page, overwriting all the content you have carefully written.

LISTING 7-6

HTML Page with Immediate Script Execution

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Date &amp; Time: Immediate Execution</title>
```

```
  </head>
  <body>
    <h1>Date &amp; Time: Immediate Execution</h1>
    <p>It is currently <span id="output">
    <script type="text/javascript">
      <!-- hide from old browsers
      var oNow = new Date();
      document.write(oNow.toLocaleString());
      // end script hiding -->
    </script>
    </span>.</p>
  </body>
</html>
```

In recent years, the use of `document.write()` has become mildly controversial. At the very least, it encourages bad structure in your document, mixing script with HTML. Good structure cleanly separates style (style sheets) and behavior (JavaScript) from the HTML markup. One issue with `document.write()` centers around the increased use of XML to feed content to the browser. XML documents must be well-formed. If a `document.write()` statement closes an element opened outside of it, or opens a new element, the XML document will fail to load because the browser will perceive it as malformed.

Another issue with `document.write()` centers around the DOM, especially with XHTML documents (which are served as XML). The use of `document.write()` means that the content will not be included in the DOM. The very important short story is that you will not be able to further manipulate such content with your JavaScript programs, limiting your dynamic response to the visitor's actions on your web page. This is an interesting conundrum because you might often choose to use `document.write()`, as we do in several examples in this book, to provide dynamic content, based on the browser environment, as the page loads.

Deferred scripts

The other three ways that script statements run are grouped together into what we call *deferred scripts*. To demonstrate these deferred script situations, we must introduce you briefly to a concept covered in more depth in Chapter 9, "Programming Fundamentals, Part II": the function. A *function* defines a block of script statements summoned to run some time after those statements load into the browser. Functions are clearly visible inside a `<script>` tag because each function definition begins with the word `function` followed by the function name (and parentheses). After a function is loaded into the browser (commonly in the `head` portion so that it loads early), it stands ready to run whenever called upon.

Run after loading

One of the most common times a function is called on to run is immediately after a page loads. Scripts using industry-standard DOM methods would fail if we attempted to operate on page elements before they appeared in the DOM, so we ask the browser to run the script only after the page has finished loading. The `window` object has an event handler property called `onload`. Unlike most event handlers, which are triggered in response to user action (for example, clicking a button), the

window's onload event handler fires the instant that all of the page's components (including images, Java applets, and embedded multimedia) are loaded into the browser.

There are two cross-browser ways to connect the onload event handler to a function: via an object event property or an HTML event attribute. The object event property we prefer is shown in Listing 7-6. The assignment of the event handler is executed in immediate mode and attaches the desired function call to the load event of the window object. At this early point in page rendering, only the window object can be guaranteed to exist in the DOM.

In old-school web development, the window.onload event assignment was written right into the HTML, with the <body> element standing in to represent the window. Therefore, you can include the onload event attribute in the <body> tag, as shown in Listing 7-7. Recall from Chapter 6, "Browser and Document Objects," that an event handler can run a script statement directly. However, if the event handler must run several script statements, it is usually more convenient to put those statements in a function definition and then have the event handler *invoke* that function. That's what happens in Listing 7-7: When the page completes loading, the onload event handler triggers the done() function. That function (simplified for this example) displays an alert dialog box.

LISTING 7-7

Running a Script from the onload Event Handler

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>An old-school HTML-based onload script</title>
      <script type="text/javascript">
         function done()
         {
            alert("The page has finished loading.");
         }
      </script>
   </head>
   <body onload="done()">
      <h1>An old-school HTML-based onload script</h1>
      <p>Here is some body text.</p>
   </body>
</html>
```

Don't worry about the curly braces or other oddities in Listing 7-7 at this point. Focus instead on the structure of the document and the flow. The entire page loads without running any script statements, although the page loads the done() function in memory so that it is ready to run at a moment's notice. After the document loads, the browser fires the onload event handler, which causes the done() function to run. Then the user sees the alert dialog box.

Although the HTML event attribute approach dates back to the earliest JavaScript browsers, the trend these days is to separate HTML markup from specifics of style (style sheets) and behavior (scripts). To the scripter's rescue come the equivalent event handler properties of objects. To get the onload

attribute out of the <body> tag, you can instead assign the desired JavaScript function to the object's event as a property, as in:

```
window.onload = done;
```

Such statements typically go near the end of scripts in the head portion of the document. Note, too, that in this version, the right side of the statement is merely the function's name, with no quotes or parentheses. Because it is easier to learn about event handlers when they're specified as HTML attributes, most examples in this tutorial continue with that approach. We needed to show you the property version, however, because you will see a lot of real-life code using that format.

Run by user

Getting a script to execute in response to a user action is very similar to the preceding example for running a deferred script right after the document loads. Commonly, a script function is defined in the head portion, and an event handler in, say, a form element calls upon that function to run. Listing 7-8 includes a script that runs when a user clicks a button.

LISTING 7-8

Running a Script from User Action

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>An onclick script</title>
      <script type="text/javascript">
         function alertUser()
         {
            alert("Ouch!");
         }
      </script>
   </head>
   <body>
      Here is some body text.
      <form>
         <input type="text" name="entry">
         <input type="button" name="oneButton"
                value="Press Me!" onclick="alertUser()">
      </form>
   </body>
</html>
```

Not every object must have an event handler defined for it, as shown in Listing 7-8 — only the ones for which scripting is needed. No script statements execute in Listing 7-8 until the user clicks the

button. The `alertUser()` function is defined as the page loads, and it waits to run as long as the page remains loaded in the browser. If it is never called upon to run, there's no harm done.

Called by another function

The last scenario for when script statements run also involves functions. In this case, a function is called upon to run by another script statement. Before you see how that works, it helps to read the next lesson, Chapter 8, "Programming Fundamentals, Part I." Therefore, we will hold off on this example until later in the tutorial.

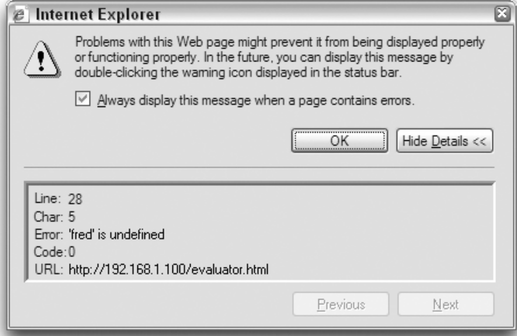
Viewing Script Errors

In the early days of JavaScript in browsers, script errors displayed themselves in very obvious dialog boxes. These boxes were certainly helpful for scripters who wanted to debug their scripts. However, if a bug got through to a page served up to a nontechnical user, the error alert dialog boxes were not only disruptive, but also scary. To prevent such dialog boxes from disturbing unsuspecting users, the browser makers tried to diminish the visual impact of errors in the browser window. Unfortunately for scripters, it is often easy to overlook the fact that your script contains an error because the error message is not so obvious.

Not only does each browser have a different way of displaying error messages, but the display can differ from version to version. The method to display errors in each of the major browsers is covered in the section "Error Message Notification" in Chapter 48, "Debugging Scripts" on the CD-ROM. While you need to read that section before testing your code, we've included a sample error dialog box as it appears in IE5+ (see Figure 7-1) and one as it appears in Mozilla 1.4+ (see Figure 7-2). Keep in mind that script-error dialog boxes are not necessarily displayed by default in these and other browsers. You have to train yourself to monitor the status bar when a page loads and after each script runs.

FIGURE 7-1

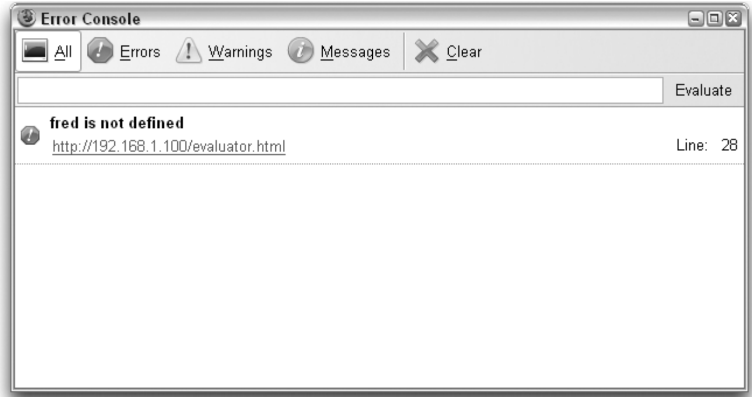
The expanded IE error dialog box.



Understanding error messages and doing something about them is a very large subject, the complete discussion of which is in reserved for Chapter 48. During this tutorial, however, you can use the error messages to see whether you mistyped a script from a listing in the book.

FIGURE 7-2

The Mozilla 1.4 JavaScript console window.



Scripting versus Programming

You may get the impression that scripting is easier than programming. *Scripting* simply sounds easier or more friendly than *programming*. In many respects, this is true. One of our favorite analogies is the difference between a hobbyist who builds model airplanes from scratch and a hobbyist who builds model airplanes from commercial kits. The "from scratch" hobbyist carefully cuts and shapes each piece of wood and metal according to very detailed plans before the model starts to take shape. The commercial kit builder starts with many prefabricated parts and assembles them into the finished product. When both builders are finished, you may not be able to tell which airplane was built from scratch and which one came out of a box of components. In the end, both builders used many of the same techniques to complete the assembly, and each can take pride in the result.

Thanks to implementations of the document object model (DOM), the browser gives scripters many prefabricated components with which to work. Without the browser, you'd have to be a pretty good programmer to develop your own application from scratch that served up content and offered user interaction. In the end, both authors have working applications that look equally professional.

Beyond the DOM, however, real programming nibbles its way into the scripting world. That's because scripts (and programs) work with more than just objects. Earlier in this lesson, we said that each statement of a JavaScript script does "something", and that "something" involves data of some kind. *Data* is the information associated with objects or other pieces of information that a script pushes around from place to place with each statement.

Data takes many forms. In JavaScript, the common incarnations of data are numbers, text (called *strings*), objects (derived from the object model or created with script), and `true` and `false` (called *Boolean values*).

Each programming or scripting language determines numerous structures and limits for each kind of data. Fortunately for newcomers to JavaScript, the universe of knowledge necessary for working with data is smaller than in a language such as Java or C++. At the same time, what you learn

about data in JavaScript is immediately applicable to future learning you may undertake in any other programming language; don't believe for an instant that your efforts in learning scripting will be wasted.

Because, deep down, scripting *is* programming, you need to have a basic knowledge of fundamental programming concepts to consider yourself a good JavaScript scripter. In the next two lessons, we set aside most of our discussion about the DOM and focus on the programming principles that will serve you well in JavaScript and future programming endeavors.

Exercises

1. Write the complete script tag set for a script whose lone statement is

   ```
   document.write("Hello, world.");
   ```

2. Build an HTML document, and include the answer to the previous question, such that the page executes the script as it loads. Open the document in your browser to test the results.
3. Add a comment to the script in the previous answer that explains what the script does.
4. Create an HTML document that displays an alert dialog box immediately after the page loads, and displays a different alert dialog box when the user clicks a form button.
5. Carefully study the document in Listing 7-9. Without entering and loading the document, predict
   - **a.** What the page looks like without further styling
   - **b.** How users interact with the page
   - **c.** What the script does

   Then type the listing as shown into a text editor. Observe all capitalization and punctuation. Do not type a carriage return after the = sign in the `upperMe` function statement; let the line word-wrap as it does in the following listing. It's okay to use a carriage return between attribute name/value pairs, as shown in the first `<input>` tag. Save the document as an HTML file, and load the file into your browser to see how well you did.

LISTING 7-9

How Does This Page Work?

```
<!DOCTYPE html>
<html>
   <head>
   <meta http-equiv="content-type" content="text/html;charset=utf-8">
   <title>Text Object Value</title>
      <script type="text/javascript">
         function upperMe()
         {
            document.getElementById("output").value =
            document.getElementById("input").value.toUpperCase();
         }
```

```
    </script>
  </head>

  <body>
    Enter lowercase letters for conversion to uppercase:<br>
    <form name="converter">
      <input type="text" name="input" id="input"
             value="sample" onchange="upperMe()" /><br />
      <input type="text" name="output" id="output" value="" />
    </form>
  </body>
</html>
```

CHAPTER

8

Programming Fundamentals, Part I

The tutorial breaks away from HTML and documents for a while, as you begin to learn programming fundamentals that apply to practically every scripting and programming language you will encounter. Here, you start learning about variables, expressions, data types, and operators — things that might sound scary if you haven't programmed before. Don't worry. With a little practice, you will become quite comfortable with these terms and concepts.

IN THIS CHAPTER

What variables are and how to use them

Why you must learn how to evaluate expressions

How to convert data from one type to another

How to use basic operators

What Language Is This?

The language you're studying is called JavaScript. But the language has some other names that you may have heard. JScript is Microsoft's name for the language. By leaving out the *ava,* the company doesn't have to license the Java name from its trademark owner: Sun Microsystems.

A standards body called ECMA (pronounced "ECK-ma") now governs the specifications for the language (no matter what you call it). The document that provides all of the details about the language is known as *ECMA-262* (it's the 262nd standard published by ECMA). Both JavaScript and JScript are ECMA-262 compatible. Some earlier browser versions exhibit very slight deviations from ECMA-262 (which came later than the earliest browsers). The most serious discrepancies are noted in the core language reference in Part III of this book.

Working with Information

With rare exceptions, every JavaScript statement you write does something with a hunk of information — *data.* Data may be text information displayed onscreen by a JavaScript statement or the on/off setting of a radio button in a form. Each single piece of information in programming is also called a *value.* Outside of programming, the term *value* usually connotes a number of some kind; in the programming world, however, the term is not as restrictive.

A string of letters is a value. A number is a value. The setting of a checkbox (whether it is checked or not) is a value.

In JavaScript, a value can be one of several types. Table 8-1 lists JavaScript's formal data types, with examples of the values you will see displayed from time to time.

TABLE 8-1

JavaScript Value (Data) Types

Type	Example	Description
String	`"Howdy"`	A series of characters inside quote marks
Number	`4.5`	Any number not inside quote marks
Boolean	`true`	A logical true or false
Null	`null`	Devoid of any content, but a value just the same
Object		A software thing that is defined by its properties and methods (arrays are also objects)
Function		A function definition

A language that contains these few data types simplifies programming tasks, especially those involving what other languages consider to be incompatible types of numbers (integers versus real or floating-point values). In some definitions of syntax and parts of objects later in this book, we make specific reference to the type of value accepted in placeholders. When a string is required, any text inside a set of quotes suffices.

You will encounter situations, however, in which the value type may get in the way of a smooth script step. For example, if a user enters a number into a form's text input field, the browser stores that number as a string value type. If the script is to perform some arithmetic on that number, you must convert the string to a number before you can apply the value to any math operations. You see examples of this later in this lesson.

Variables

Cooking up a dish according to a recipe in the kitchen has one advantage over cooking up some data in a program. In the kitchen, you follow recipe steps and work with real things: carrots, milk, or a salmon filet. A computer, on the other hand, follows a list of instructions to work with data. Even if the data represents something that looks real, such as the text entered into a form's input field, once the value gets into the program, you can no longer reach out and touch it.

In truth, the data that a program works with is merely a collection of bits (on and off states) in your computer's memory. More specifically, data in a JavaScript-enhanced web page occupies parts of the computer's memory set aside for exclusive use by the browser software. In the olden days, programmers had to know the numeric address in memory (RAM) where a value was stored to retrieve a copy of it for, say, some addition. Although the innards of a program have that level of complexity, programming languages such as JavaScript shield you from it.

The most convenient way to work with data in a script is to first assign the data to a *variable*. It's usually easier to think of a variable as a basket that holds information. How long the variable holds the information depends on a number of factors. But the instant a web page clears the window (or frame), any variables it knows about are discarded.

Creating variables

You have a couple of ways to create a variable in JavaScript, but one way covers all cases properly. Use the `var` keyword, followed by the name you want to give that variable. Therefore, to *declare* a new variable called `myAge`, the JavaScript statement is

```
var myAge;
```

That statement lets the browser know that you can use that variable later to hold information or to modify any of the data in that variable.

To assign a value to a variable, use one of the *assignment operators*. The most common one by far is the equal sign. For example, if you want to assign a value to the `myAge` variable at the same time that you declare it (a combined process known as *initializing the variable*), use that operator in the same statement as the `var` keyword:

```
var myAge = 45;
```

On the other hand, if you declare a variable in one statement and later want to assign a value to it, the sequence of statements is

```
var myAge;
myAge = 45;
```

Use the `var` keyword for declaration or initialization — typically only once for the life of any variable name in a document.

A JavaScript variable can hold any value type. Unlike many other languages, you don't have to tell JavaScript during variable declaration what type of value the variable will hold. In fact, the value type of a variable can change during the execution of a program. (This flexibility drives experienced programmers crazy because they're accustomed to assigning both a data type and a value to a variable.)

Naming variables

Choose the names you assign to variables with care. You'll often find scripts that use vague variable names, such as single letters. Other than a few specific circumstances where using letters is a common practice (for example, using `i` as a counting variable in repeat loops in Chapter 9), you should use names that truly describe a variable's contents. This practice can help you follow the state of your data through a long series of statements or jumps, especially for complex scripts.

A number of restrictions help instill good practice in assigning names. First, you cannot use any reserved keyword as a variable name. That includes all keywords currently used by the language and all others held in reserve for future versions of JavaScript. The designers of JavaScript, however, cannot foresee every keyword that the language may need in the future. By using the kind of single words that currently appear in the list of reserved keywords (see Appendix C, on the CD-ROM), you always run a risk of a future conflict.

To complicate matters, a variable name cannot contain space characters. Therefore, one-word variable names are fine. Should your description really benefit from more than one word, you can use one of

two conventions to join multiple words as one. One convention is to place an underscore character between the words; the other is to start the combination word with a lowercase letter and capitalize the first letter of each subsequent word within the name — referred to as *CamelCase* or *interCap* format. Both of the following examples are valid variable names:

```
my_age
myAge
```

Our preference is for the second version; it is easier to type and easier to read. In fact, because of the potential conflict with future one-word keywords, using multiword combinations for variable names is a good idea. Multiword combinations are less likely to appear in the list of reserved words.

Variable names have a couple of other important restrictions. Avoid all punctuation symbols except for the underscore character. Also, the first character of a variable name cannot be a numeral. If these restrictions sound familiar, it's because they're similar to those for HTML element identifiers described in Chapter 6.

Expressions and Evaluation

Another concept closely related to the value and variable is *expression evaluation* — perhaps the most important concept in learning how to program a computer.

We use expressions in our everyday language. Remember the theme song of "The Beverly Hillbillies"?:

Then one day he was shootin' at some food
And up through the ground came a-bubblin' crude
Oil, that is. Black gold. Texas tea.

At the end of the song, you find four quite different references (crude, oil, black gold, and Texas tea). They all mean oil. They're all *expressions* for oil. Say any one of them, and other people know what you mean. In our minds, we *evaluate* those expressions to mean one thing: oil.

In programming, a variable always evaluates to its contents, or value. For example, after assigning a value to a variable, such as

```
var myAge = 45;
```

any time the variable is used in a statement, its value (`45`) is automatically applied to whatever operation that statement calls. Therefore, if you're 15 years my junior, I can assign a value to a variable representing your age based on the evaluated value of `myAge`:

```
var yourAge = myAge - 15;
```

The variable, `yourAge`, evaluates to 30 the next time the script uses it. If the `myAge` value changes later in the script, the change has no link to the `yourAge` variable because `myAge` evaluated to `45` when it was used to assign a value to `yourAge`.

Expressions in scripts

You probably didn't recognize it at the time, but you have seen how expression evaluation came in handy in several scripts in previous chapters. Let's look at one in particular — from

Listing 5-6 — where a script writes dynamic text to the page as the page loads. Recall the second `document.write()` statement:

```
document.write(" of " + navigator.appName + ".");
```

Testing JavaScript Evaluation

You can begin experimenting with the way JavaScript evaluates expressions with the help of The Evaluator Jr. (see Figure 8-1), an HTML page you can find on the companion CD-ROM. (The Senior version is introduced in Chapter 4, ''JavaScript Essentials.'') Enter any JavaScript expression into the top text box, and either press Enter/Return or click the Evaluate button.

FIGURE 8-1

The Evaluator Jr. for testing expression evaluation.

The Evaluator Jr. has 26 variables (lowercase `a` through `z`) predefined for you. Therefore, you can assign values to variables, test comparison operators, and even do math here. Using the `age` variable examples

continued

continued

from earlier in this chapter, type each of the following statements in the upper text box, and observe how each expression evaluates in the Results field. Be sure to observe case sensitivity in your entries. The trailing semicolons are optional in The Evaluator.

```
a = 45;
a;
b = a - 15;
b;
a - b;
a > b;
```

To start over, click the Reload/Refresh button.

The `document.write()` method (remember, JavaScript uses the term *method* to mean *command*) requires a parameter in the parentheses: the text string to be displayed on the web page. The parameter here consists of one expression that joins three distinct strings:

```
" of "
navigator.appName
"."
```

The plus symbol is one of JavaScript's ways of joining strings. Before JavaScript can display this line, it must perform some quick evaluations. The first evaluation is the value of the `navigator.appName` property. This property evaluates to a string of the name of your browser. With that expression safely evaluated to a string, JavaScript can finish the job of joining the three strings in the final evaluation. The evaluated string expression is what ultimately appears on the web page.

Expressions and variables

As one more demonstration of the flexibility that expression evaluation offers, this section shows you a slightly different route to the `document.write()` statement. Rather than join those strings as the direct parameter to the `document.write()` method, you can gather the strings in a variable and then apply the variable to the `document.write()` method. Here's how that sequence looks, as you simultaneously declare a new variable and assign it a value:

```
var textToWrite = " of " + navigator.appName + ".";
document.write(textToWrite);
```

This method works because the variable, `textToWrite`, evaluates to the combined string. The `document.write()` method accepts that string value and does its display job. As you read a script or try to work through a bug, pay special attention to how each expression (variable, statement, object property) evaluates. As you learn JavaScript (or any language), you will end up scratching your head from time to time because you haven't stopped to examine how expressions evaluate when a particular kind of value is required in a script.

Data Type Conversions

We mentioned earlier that the type of data in an expression can trip up some script operations if the expected components of the operation are not of the right type. JavaScript tries its best to perform internal conversions to head off such problems, but JavaScript cannot read your mind. If your intentions differ from the way JavaScript treats the values, you won't get the results you expect.

A case in point is adding numbers that may be in the form of text strings. In a simple arithmetic statement that adds two numbers, you get the expected result:

```
3 + 3          // result = 6
```

But if one of those numbers is a string, JavaScript leans toward converting the other value to a string — thus turning the plus sign's action from arithmetic addition to joining strings. Therefore, in the statement

```
3 + "3"        // result = "33"
```

the stringness of the second value prevails over the entire operation. The first value is automatically converted to a string, and the result joins the two strings. Try this yourself in The Evaluator Jr.

If you take this progression one step further, look what happens when another number is added to the statement:

```
3 + 3 + "3"    // result = "63"
```

This might seem totally illogical, but there is logic behind this result. The expression is evaluated from left to right. The first plus operation works on two numbers, yielding a value of `6`. But as the `6` is about to be added to the `3`, JavaScript lets the stringness of the `3` rule. The `6` is converted to a string, and two string values are joined to yield `63`.

Most of your concern about data types will focus on performing math operations like the ones here. However, some object methods also require one or more parameters of particular data types. Although JavaScript provides numerous ways to convert data from one type to another, it is appropriate at this stage of the tutorial to introduce you to the two most common data conversions: string to number and number to string.

Converting strings to numbers

As you saw in the preceding section, if a numeric value is stored as a string — as it is when entered into a form text field — your scripts may have difficulty applying that value to a math operation. The JavaScript language provides two built-in functions to convert string representations of numbers to true numbers: `parseInt()` and `parseFloat()`.

There is a difference between integers and floating-point numbers in JavaScript. *Integers* are always whole numbers, with no decimal point or numbers to the right of a decimal. *Floating-point numbers,* on the other hand, have fractional values to the right of the decimal. By and large, JavaScript math operations don't differentiate between integers and floating-point numbers: A number is a number. The only time you need to be cognizant of the difference is when a method parameter requires an integer because it can't handle fractional values. For example, parameters to the `scroll()` method of a window require integer values of the numbers of pixels vertically and horizontally that you want to scroll the window. That's because you can't scroll a window a fraction of a pixel onscreen.

To use either of these conversion functions, insert the string value you wish to convert as a parameter to the function. For example, look at the results of two different string values when passed through the `parseInt()` function:

```
parseInt("42")        // result = 42
parseInt("42.33")     // result = 42
```

Even though the second expression passes the string version of a floating-point number to the function, the value returned by the function is an integer. No rounding of the value occurs here (although other math functions can help with that if necessary). The decimal and everything to its right are simply stripped off.

The `parseFloat()` function returns an integer if it can; otherwise, it returns a floating-point number, as follows:

```
parseFloat("42")       // result = 42
parseFloat("42.33")    // result = 42.33
```

Because these two conversion functions evaluate to their results, you simply insert the entire function wherever you need a string value converted to a number. Therefore, modifying an earlier example in which one of three values was a string, the complete expression can evaluate to the desired result:

```
3 + 3 + parseInt("3")  // result = 9
```

Converting numbers to strings

You'll have less need for converting a number to its string equivalent than the other way around. As you saw in the previous section, JavaScript gravitates toward strings when faced with an expression containing mixed data types. Even so, it is good practice to perform data type conversions explicitly in your code to prevent any potential ambiguity. The simplest way to convert a number to a string is to take advantage of JavaScript's string tendencies in addition operations. By adding an empty string to a number, you convert the number to its string equivalent:

```
("" + 2500)           // result = "2500"
("" + 2500).length    // result = 4
```

In the second example, you can see the power of expression evaluation at work. The parentheses force the conversion of the number to a string. A *string* is a JavaScript object that has properties associated with it. One of those properties is the `length` property, which evaluates to the number of characters in the string. Therefore, the length of the string `"2500"` is `4`. Note that the `length` value is a number, not a string.

Operators

You will use lots of *operators* in expressions. Earlier, you used the equal sign (=) as an assignment operator to assign a value to a variable. In the preceding examples with strings, you used the plus symbol (+) to join two strings. An operator generally performs some kind of calculation (operation) or comparison with two values (the value on each side of an operator is called an *operand*) to reach a third value. This lesson briefly describes two categories of operators: arithmetic and comparison. Chapter 22, "JavaScript Operators," covers many more operators, but after you understand the basics here, the others are easier to grasp.

Arithmetic operators

It may seem odd to talk about text strings in the context of arithmetic operators, but you have already seen the special case of the plus (+) operator when one or more of the operands is a string. The plus operator instructs JavaScript to *concatenate* (pronounced "kon-KAT-en-eight"), or join, two strings together precisely where you place the operator. The string concatenation operator doesn't know about words and spaces, so the programmer must make sure that any two strings to be joined have the proper word spacing as part of the strings, even if that means adding a space:

```
firstName = "John";
lastName = "Doe";
fullName = firstName + " " + lastName;
```

JavaScript uses the same plus operator for arithmetic addition. When both operands are numbers, JavaScript knows to treat the expression as an arithmetic addition rather than a string concatenation. The standard math operators for addition, subtraction, multiplication, and division (+, -, *, /) are built into JavaScript.

Comparison operators

Another category of operator helps you compare values in scripts — whether two values are the same, for example. These kinds of comparisons return a value of the Boolean type: `true` or `false`. Table 8-2 lists the comparison operators. The operator that tests whether two items are equal consists of a pair of equal signs to distinguish it from the single-equal-sign assignment operator.

TABLE 8-2

JavaScript Comparison Operators

Symbol	Description
==	Equals
!=	Does not equal
>	Is greater than
>=	Is greater than or equal to
<	Is less than
<=	Is less than or equal to

Comparison operators come into greatest play in the construction of scripts that make decisions as they run. A cook does this in the kitchen all the time: If the sauce is too watery, add a bit of flour. You see comparison operators in action in Chapter 22.

Exercises

1. Which of the following are valid variable declarations or initializations? Explain why each one is or is not valid. If an item is not valid, how do you fix it so that it is?
   a. `my_name = "Cind";`
   b. `var how many = 25;`
   c. `var zipCode = document.getElementById("zip").value`
   d. `var 1address = document.("address1").value;`
2. Assume that the following statements operate rapidly in sequence, where each statement relies on the result of the one before it. For each of the statements in the sequence, write down how the `someVal` expression evaluates after the statement executes in JavaScript.

```
var someVal = 2;
someVal = someVal + 2;
someVal = someVal * 10;
someVal = someVal + "20";
someVal = "Robert";
```

3. Name the two JavaScript functions that convert strings to numbers. How do you give the function a string value to convert to a number?
4. Type and load the HTML page and script shown in Listing 8-1. Enter a three-digit number in the top two fields, and click the Add button. Examine the code, and explain what is wrong with the script. How do you fix the script so that the proper sum is displayed in the output field?

LISTING 8-1

What's Wrong with This Script?

HTML: jsb-08-01.html

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Making Sums</title>
      <script type="text/javascript" src="../jsb-global.js"></script>
      <script type="text/javascript" src="jsb-08-01.js"></script>
   </head>
   <body>
      <h1>Making Sums</h1>
      <form action="addit.php">
         <p><input type="text" id="inputA" name="inputA" value="0"></p>
         <p><input type="text" id="inputB" name="inputB" value="0"></p>
         <p><input type="button" id="add" name="add" value="Add"></p>
         <p><input type="text" id="output" name="output"></p>
      </form>
   </body>
</html>
```

JavaScript: jsb-08-01.js

```
// initialize when the page has loaded
addEvent(window, 'load', initialize);

function initialize()
{
   // do this only if the browser can handle DOM methods
   if (document.getElementById)
   {
      // point to critical elements
      var oInputA = document.getElementById('inputA');
      var oInputB = document.getElementById('inputB');
      var oButton = document.getElementById('add');
      var oOutput = document.getElementById('output');

         // if they all exist...
         if (oInputA && oInputB && oButton && oOutput)
         {
            // apply behaviors
            addEvent(oButton, 'click', addIt);
         }
   }
}

// add two input numbers & display result
function addIt()
{
   var value1 = document.getElementById("inputA").value;
   var value2 = document.getElementById("inputB").value;

   document.getElementById("output").value = value1 + value2;
}
```

5. What does the term *concatenate* mean in the context of JavaScript programming?

CHAPTER

Programming Fundamentals, Part II

Your tour of programming fundamentals continues in this chapter with subjects that have more intriguing possibilities. For example, we show you how programs make decisions and why a program must sometimes repeat statements over and over. Before you're finished here, you also will learn how to use one of the most powerful information holders in the JavaScript language: the array.

IN THIS CHAPTER

How control structures make decisions

How to define functions

Where to initialize variables efficiently

What those darned curly braces are all about

The basics of data arrays

Decisions and Loops

Every waking hour of every day, you make decisions of some kind; most of the time, you probably don't even realize it. Don't think so? Well, look at the number of decisions you make at the grocery store, from the moment you enter the store to the moment you clear the checkout aisle.

No sooner do you enter the store than you are faced with a decision. Based on the number and size of items you intend to buy, do you pick up a hand-carried basket or attempt to extricate a shopping cart from the metallic conga line near the front of the store? That key decision may have impact later, when you see a special offer on an item that is too heavy to put in the handbasket.

Next, you head for the food aisles. Before entering an aisle, you compare the range of goods stocked in that aisle with items on your shopping list. If an item you need is likely to be found in this aisle, you turn into the aisle and start looking for the item; otherwise, you skip the aisle and move to the head of the next aisle.

Later, you reach the produce section in search of a juicy tomato. Standing in front of the bin of tomatoes, you begin inspecting them one by one — picking one up, feeling its firmness, checking the color, looking for blemishes or signs of pests. You discard one, pick up another, and continue this process until one matches the criteria you set in your mind for an acceptable morsel. Your last stop in the store is the checkout aisle. "Paper or plastic?" the clerk asks. One more decision to make. What you choose affects how you get the groceries from the car to the kitchen, as well as your recycling habits.

During your trip to the store, you go through the same kinds of decisions and repetitions that your JavaScript programs encounter. If you understand these frameworks in real life, you can look into the JavaScript equivalents and the syntax required to make them work.

Control Structures

In the vernacular of programming, the kinds of statements that make decisions and loop around to repeat themselves are called *control structures*. A control structure directs the execution flow through a sequence of script statements based on simple decisions and other factors.

An important part of a control structure is the *condition*. Just as you may travel different routes to work depending on certain conditions (for example, nice weather, nighttime, whether you're attending a soccer game), so, too, does a program sometimes have to branch to an execution route if a certain condition exists. Each condition is an expression that evaluates to `true` or `false` — one of those Boolean data types mentioned in Chapter 8. The kinds of expressions commonly used for conditions are expressions that include a comparison operator. You do the same in real life: If it is true that the outdoor temperature is less than freezing, you put on a coat before going outside. In programming, the comparisons are strictly comparisons of numeric values and character strings.

JavaScript provides several kinds of control structures for different programming situations. Three of the most common control structures you'll use are `if` constructions, `if...else` constructions, and `for` loops.

Chapter 21 covers in great detail other common control structures you should know. For this tutorial, however, you need to learn about the three common ones just mentioned.

if constructions

The simplest program decision is to follow a special branch or path of the program if a certain condition is true. Formal syntax for this construction follows. Items in italics get replaced in a real script with expressions and statements that fit the situation.

```
if (condition)
{
   statement[s] if true
}
```

Don't worry about the curly braces yet. Instead, get a feel for the basic structure. The keyword, `if`, is a must. Between the parentheses goes an expression that evaluates to a Boolean (true/false) value. This is the condition being tested as the program runs past this point. If the condition evaluates to `true`, one or more statements inside the curly braces execute before the program continues with the next statement after the closing brace. If the condition evaluates to `false`, the statements inside the curly braces are ignored, and processing continues with the next statement after the closing brace.

The following example assumes that a variable, `myAge`, has had its value set earlier in the script (exactly how is not important for this example). The conditional expression compares the value `myAge` against a numeric value of 13:

```
if (myAge < 13)
{
   alert("You are not yet a teenager.");
}
```

In this example, the data type of the value inside myAge must be a number so that the proper comparison (via the <, or less than, comparison operator) does the right thing. For all instances of myAge less than 13, the nested statement inside the curly braces runs and displays the alert to the user. After the user closes the alert dialog box, the script continues with whatever statement follows the entire if construction.

if . . . else constructions

Not all program decisions are as simple as the one shown for the if construction. Rather than specifying one detour for a given condition, you might want the program to follow either of two branches depending on that condition. It is a fine but important distinction. In the plain if construction, no special processing is performed when the condition evaluates to false. But if processing must follow one of two special paths, you need the if...else construction. The formal syntax definition for an if...else construction is as follows:

```
if (condition)
{
   statement[s] if true
}
else
{
   statement[s] if false
}
```

Everything you know about the condition for an if construction applies here. The only difference is the else keyword, which provides an alternative path for execution to follow if the condition evaluates to false.

As an example, the following if...else construction determines how many days are in February for a given year. To simplify the demo, the condition simply tests whether the year divides equally by 4. (True testing for this value includes special treatment of end-of-century dates, but we're ignoring that for now.) The % operator symbol is called the *modulus operator* (covered in more detail in Chapter 22). The result of an operation with this operator yields the remainder of division of the two values. If the remainder is zero, the first value divides evenly by the second.

```
var febDays;
var theYear = 2010;
   if (theYear % 4 == 0)
   {
      febDays = 29;
   }
   else
   {
      febDays = 28;
   }
```

The important point to see from this example is that by the end of the if...else construction, the febDays variable is set to either 28 or 29. No other value is possible. For years evenly divisible by 4, the first nested statement runs. For all other cases, the second statement runs. Then processing picks up with the next statement after the if...else construction.

Repeat Loops

Repeat loops in real life generally mean the repetition of a series of steps until some condition is met, thus enabling you to break out of that loop. Such was the case earlier in this chapter, when you looked through a bushel of tomatoes for the one that came closest to your ideal tomato. The same can be said for driving around the block in a crowded neighborhood until a parking space opens up.

A *repeat loop* lets a script cycle through a sequence of statements until some condition is met. For example, a JavaScript data validation routine might inspect every character that you enter in a form text field to make sure that each one is a number. Or, if you have a collection of data stored in a list, the loop can check whether an entered value is in that list. When that condition is met, the script can break out of the loop and continue with the next statement after the loop construction.

The most common repeat loop construction used in JavaScript is called the `for` loop. It gets its name from the keyword that begins the construction. A `for` loop is a powerful device because you can set it up to keep track of the number of times the loop repeats itself. The formal syntax of the `for` loop is as follows:

```
for ([initial expression]; [condition]; [update expression])
{
   statement[s] inside loop
}
```

The square brackets mean that the item is optional. However, until you get to know the `for` loop better, we recommend designing your loops to use all three items inside the parentheses. The *initial expression* portion usually sets the starting value of a counter variable. The *condition* — the same kind of condition you saw for `if` constructions — defines the condition that forces the loop to stop going around and around. Finally, the *update expression* is a statement that executes each time all the statements nested inside the construction complete running.

A common implementation initializes a counting variable, `i`; increments the value of `i` by 1 each time through the loop; and repeats the loop until the value of `i` exceeds some maximum value, as in the following:

```
for (var i = startValue; i <= maxValue; i++)
{
   statement[s] inside loop
}
```

Placeholders `startValue` and `maxValue` represent any numeric values, including explicit numbers or variables holding numbers. In the update expression is an operator we have not presented yet. The ++ operator adds 1 to the value of `i` each time the update expression runs at the end of the loop. If `startValue` is 1, the value of `i` is 1 the first time through the loop, 2 the second time through, and so on. Therefore, if `maxValue` is 10, the loop repeats itself 10 times (in other words, as long as `i` is less than or equal to 10). Generally speaking, the statements inside the loop use the value of the counting variable in their execution. Later in this lesson, we show how the variable can play a key role in the statements inside a loop. At the same time, you will see how to break out of a loop prematurely and why you may need to do this in a script.

Functions

In Chapter 7, you saw a preview of the JavaScript function. A *function* is a definition of a set of deferred actions. Functions are invoked by event handlers or by statements elsewhere in the script.

Whenever possible, good functions are designed for reuse in other documents. They can become building blocks you use over and over again.

If you have programmed before, you can see parallels between JavaScript functions and other languages' subroutines. But unlike some languages that distinguish between procedures (which carry out actions) and functions (which carry out actions and return values), only one classification of routine exists for JavaScript. A function is capable of returning a value to the statement that invoked it, but this is not a requirement. When a function does return a value, the calling statement treats the function call like any expression — plugging in the returned value right where the function call is made. We will show some examples in a moment.

Formal syntax for a function is as follows:

```
function functionName ( [parameter1]...[,parameterN] )
{
   statement[s]
}
```

Names you assign to functions have the same restrictions as names you assign to HTML elements and variables. You should devise a name that succinctly describes what the function does. We tend to use multiword names with the CamelCase or interCap (internally capitalized) format that start with a verb because functions are action items, even if they do nothing more than get or set a value.

Another practice to keep in mind as you start to create functions is to keep the focus of each function as narrow as possible. It is possible to generate functions that are literally hundreds of lines long. Such functions are usually difficult to maintain and debug. Chances are that you can divide the long function into smaller, more tightly focused segments.

Function parameters

In Chapter 4, you saw how an event handler invokes a function by calling the function by name. A typical call to a function, including one that comes from another JavaScript statement, works the same way: A set of parentheses follows the function name.

You also can define functions so that they receive parameter values from the calling statement. Listing 9-1 shows a simple script in which a statement passes text data to a function at the same time that it calls the function.

LISTING 9-1

Calling a Function from an Event Handler

HTML: jsb-09-01.html

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Passing a Parameter to a Function</title>
      <script type="text/javascript" src="jsb-09-01.js"></script>
   </head>
```

continued

LISTING 9-1 *(continued)*

```
   <body>
      <h1>Passing a Parameter to a Function</h1>
   </body>
</html>
```

JavaScript: jsb-09-01.js

```
// display a personalized greeting
function greeting(sName)
{
   alert("Hello, " + sName + "!");
}

greeting("Chris");
```

Parameters (also known as *arguments*) provide a mechanism for handing off a value from one statement to another by way of a function call. If no parameters occur in the function definition, both the function definition and the call to the function have only empty sets of parentheses (as shown in Chapter 4, Listing 4-2).

When a function receives parameters, it assigns the incoming values to the variable names specified in the function definition's parentheses. Consider the following script segment:

```
function sayHelloFirst(a, b, c)
{
   alert("Say hello, " + a);
}
sayHelloFirst("Gracie", "George", "Harry");
sayHelloFirst("Larry", "Moe", "Curly");
```

After the function is defined in the script, the next statement calls that very function, passing three strings as parameters. The function definition automatically assigns the strings to variables `a`, `b`, and `c`. Therefore, before the `alert()` statement inside the function ever runs, `a` evaluates to `"Gracie"`, `b` evaluates to `"George"`, and `c` evaluates to `"Harry"`. In the `alert()` statement, only the `a` value is used, and the alert reads

```
Say hello, Gracie
```

When the user closes the first alert, the next call to the function occurs. This time through, different values are passed to the function and assigned to `a`, `b`, and `c`. The alert dialog box reads

```
Say hello, Larry
```

Unlike other variables that you define in your script, function parameters do not use the `var` keyword to initialize them. They are automatically initialized whenever the function is called.

Variable scope

Speaking of variables, it's time to distinguish between variables that are defined outside and those that are defined inside functions. Variables defined outside of functions are called *global variables;* those defined inside functions with the `var` keyword are called *local variables.*

A global variable has a slightly different connotation in JavaScript than it has in most other languages. For a JavaScript script, the globe of a global variable is the current document loaded in a browser window or frame. Therefore, when you initialize a variable as a global variable, it means that all script statements in the page (including those inside functions) have direct access to that variable's value via the variable's name. Statements can retrieve and modify global variables from anywhere in the script. In programming terminology, this kind of variable is said to have *global scope* because every statement on the page can see it.

It is important to remember that the instant a page unloads itself, all global variables defined in that page disappear from memory forever. If you need a value to persist from one page to another, you must use other techniques to store that value (for example, as a global variable in a framesetting document, as described in Chapter 27, "Window and Frame Objects," or in a cookie, as described in Chapter 29, "The Document and Body Objects." Although the `var` keyword is usually optional for initializing global variables, we strongly recommend that you use it for all variable initializations to guard against future changes to the JavaScript language.

In contrast to the global variable, a local variable is defined inside a function. You already saw how parameter variables are defined inside functions (without `var` keyword initializations). But you can also define other variables with the `var` keyword (absolutely required for local variables; otherwise, they become recognized as global variables). The scope of a local variable is only within the statements of the function. No other functions or statements outside functions have access to a local variable.

Local scope allows for the reuse of variable names within a document. For most variables, we strongly discourage this practice because it leads to confusion and bugs that are difficult to track down. At the same time, it is convenient to reuse certain kinds of variable names, such as `for` loop counters. These are safe because they are always reinitialized with a starting value whenever a `for` loop starts. In order to nest one `for` loop inside another, you need to specify a different loop-counting variable in the nested loop.

To demonstrate the structure and behavior of global and local variables — and show you why it can get confusing to reuse most variable names inside a script — Listing 9-2 defines two global variables and a local one. For the purposes of this illustration, we intentionally use bad form by initializing a local variable that has the same name as a global variable.

LISTING 9-2

Global and Local Variable Scope Demonstration

HTML: jsb-09-02.html

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Global and Local Variable Scope</title>
```

continued

LISTING 9-2 *(continued)*

```
      <script type="text/javascript" src="jsb-09-02.js"></script>
   </head>
   <body>
      <h1>Global and Local Variable Scope</h1>
   </body>
</html>
```

JavaScript: jsb-09-02.js

```
var Boy = "Charlie Brown";   // global
var Dog = "Snoopy";        // global

// using improper design to demonstrate a point
function demo()
{
   // local variable with the same name as a global
   var Dog = "Gromit";

   alert(Dog + " does not belong to " + Boy + ".");
}

// use global variables
alert(Dog + " belongs to " + Boy + ".");

// use global & local variables
demo();
```

When the page loads, the script initializes the two global variables (`Boy` and `Dog`) and defines the `demo()` function in memory. Inside the function, a local variable is initialized with the same name as one of the global variables: `Dog`. In JavaScript, such a local initialization overrides the global variable for all statements inside the function. (But note that if we had omitted the `var` keyword from the local initialization, the variable `Dog` inside the function would have referred to the global variable — i.e., `"Gromit"`.)

The script displays two `alert` dialogs. The first to appear concatenates the two global variables to display:

```
Snoopy belongs to Charlie Brown.
```

The second dialog, invoked by calling the function `demo()` in the last line of the script, occurs inside the `demo()` function and therefore uses the local variable `Dog` instead of the global one:

```
Gromit does not belong to Charlie Brown.
```

Curly Braces

Despite the fact that you probably rarely — if ever — use curly braces (`{ }`) in your writing, there is no mystery to their usage in JavaScript (and many other languages). Curly braces enclose blocks of

statements that belong together. Although they do assist humans who are reading scripts in knowing what's going on, curly braces also help the browser know which statements belong together. You always must use curly braces in matched pairs.

You use curly braces most commonly in function definitions and control structures. In the function definition in Listing 9-2, curly braces enclose three statements (including the comment line) that make up the function definition. The closing brace lets the browser know that whatever statement comes next is a statement outside the function definition.

Physical placement of curly braces is not critical. (Neither is the indentation style you see in the code we provide.) The following function definitions are treated identically by scriptable browsers:

```
function sayHiToFirst(a, b, c)
{
   alert("Say hello, " + a);
}
function sayHiToFirst(a, b, c) {
   alert("Say hello, " + a);
}

function sayHiToFirst(a, b, c) {alert("Say hello, " + a);}
```

Throughout this book, we use the style shown in the first example because we find that it makes lengthy and complex scripts easier to read — especially scripts that have many levels of nested control structures. However, this aspect of scripting style is highly personal and varies from one programmer to another — even among the co-authors of this book!

Arrays

The JavaScript array is one of the most useful data constructions you have available to you. You can visualize the structure of a basic array as though it were a single-column spreadsheet. Each row of the column holds a distinct piece of data, and each row is numbered. Numbers assigned to rows are in strict numerical sequence, starting with zero as the first row. (Computers tend to start counting with zero.) This row number is called an *index*. To access an item in an array, you need to know the name of the array and the index for the row. Because index values start with zero, the total number of items of the array (as determined by the array's `length` property) is always one more than the highest index value of the array. More advanced array concepts enable you to create the equivalent of an array with multiple columns (described in Chapter 18). For this tutorial, we stay with the single-column basic array.

Data elements inside JavaScript arrays can be any data type, including objects. And unlike a lot of other programming languages, JavaScript allows different rows of the same array to contain different data types.

Creating an array

An array is stored in a variable, so when you create an array, you assign the new array object to the variable. (Arrays are objects that belong to the core JavaScript language rather than to the document object model [DOM].) A special keyword — `new` — preceding a call to the JavaScript function that generates arrays creates space in memory for the array. An optional parameter to the `Array()` function enables you to specify at the time of creation how many elements (rows) of data eventually will occupy the array. JavaScript is very forgiving about this because you can change the size of an array

at any time. Therefore, if you omit a parameter when generating a new array, your script incurs no penalty.

To demonstrate the array creation process, we create an array that holds the names of the 50 U.S. states plus the District of Columbia (a total of 51). The first task is to create that array and assign it to a variable of any name that helps us remember what this collection of data is about:

```
var USStates = new Array(51);
```

At this point, the `USStates` array is sitting in memory like a 51-row table with no data in it. To fill the rows, we must assign data to each row. Addressing each row of an array requires a special way of indicating the index value of the row: square brackets after the name of the array. The first row of the `USStates` array is addressed as:

```
USStates[0]
```

To assign the string name of the first state of the alphabet to that row, use a simple assignment operator:

```
USStates[0] = "Alabama";
```

To fill in the rows, include a statement for each row:

```
USStates[0] = "Alabama";
USStates[1] = "Alaska";
USStates[2] = "Arizona";
USStates[3] = "Arkansas";
...
USStates[50] = "Wyoming";
```

(Note that the 51st array element has an index of 50 because we're starting with 0, not 1.)

Therefore, if you want to include a table of information from which a script can look up information without accessing the server, you include the data in the script in the form of an array creation sequence. When the script loads into the browser and the statements run, the data collection array is built and ready to go. Despite what appears to be the potential for a lot of statements in a document for such a data collection, the amount of data that must download for typical array collections is small enough not to affect page loading severely — even for dial-up users. In Chapter 18, you also see some syntax shortcuts for creating arrays that reduce source code character counts.

Accessing array data

The array index is the key to accessing an array element. The name of the array and an index in square brackets evaluates to the content of that array location. For example, after the `USStates` array is built, a script can display an alert with Alaska's name in it with the following statement:

```
alert("The largest state is " + USStates[1] + ".");
```

Just as you can retrieve data from an indexed array element, you can change the element by reassigning a new value to any indexed element in the array.

Parallel arrays

Now we show you why the numeric index methodology works well in JavaScript. To help with the demonstration, we generate another array that is parallel with the `USStates` array. This new array is also 51 elements long, and it contains the year in which the state in the corresponding row of `USStates` entered the Union. That array construction looks like the following:

```
var stateEntered = new Array(51);
stateEntered [0] = 1819;
stateEntered [1] = 1959;
stateEntered [2] = 1912;
stateEntered [3] = 1836;
...
stateEntered [50] = 1890;
```

In the browser's memory, then, are two data tables that you can visualize as looking like the model in Figure 9-1. We can build more arrays that are parallel to these for items such as the postal abbreviation and capital city. The important point is that the zeroth element in each of these tables applies to Alabama, the first state in the `USStates` array.

FIGURE 9-1

Visualization of two related parallel data tables.

USStates		stateEntered
"Alabama"	[0]	1819
"Alaska"	[1]	1959
"Arizona"	[2]	1912
"Arkansas"	[3]	1836
⋮	⋮	⋮
"Wyoming"	[50]	1890

If a web page included these data tables and a way for a user to look up the entry date for a given state, the page would need a way to look through all the `USStates` entries to find the index value of the one that matches the user's entry. Then that index value could be applied to the `stateEntered` array to find the matching year.

For this demo, the page includes a text entry field in which the user types the name of the state to look up. This methodology is fraught with peril unless the script performs some error checking in case the user makes a mistake. Let's not assume that the user always types a valid state name; instead, let's gracefully handle the circumstance in which they don't. (Don't ever assume that user input is valid in your web site's pages.) An event handler from either the text field or a clickable button calls a function that looks up the state name, fetches the corresponding entry year, and displays an alert message with the information. The function is as follows:

```
function getStateDate()
{
   var selectedState = document.getElementById("entry").value;
   for (var i = 0; i < USStates.length; i++)
   {
```

```
         if (USStates[i] == selectedState)
         {
            break;
         }
      }

      if (i < USStates.length)
      {
         alert(selectedState + " entered the Union in " + stateEn-
tered[i] + ".");
      }
      else
      {
         alert("Sorry, '" + selectedState + "' isn't a US state.");
      }
}
```

In the first statement of the function, we grab the value of the text box and assign the value to a variable, `selectedState`. This is mostly for convenience, so that we can use this shorter variable name later in the script. In fact, the usage of that value is inside a `for` loop, so the script is marginally more efficient because the browser doesn't have to evaluate that long reference to the text field each time through the loop.

The key to this function is in the `for` loop. Here is where we combine the natural behavior of incrementing a loop counter with the index values assigned to the two arrays. Specifications for the loop indicate that the counter variable, `i`, is initialized with a value of zero. The loop is directed to continue as long as the value of `i` is less than the length of the `USStates` array. Remember that the length of an array is always one more than the index value of the last item. Therefore, the last time the loop runs is when `i` is 50, which is both less than the length of 51 and equal to the index value of the last element. Each time after the loop runs, the counter increments by 1 (`i++`).

Nested inside the `for` loop is an `if` construction. The condition tests the value of an element of the array against the value typed by the user. Each time through the loop, the condition tests a different row of the array, starting with row zero. In other words, this `if` construction can be performed dozens of times before a match is found, but each time, the value of `i` is 1 larger than in the previous try.

The equality comparison operator (==) is fairly strict when it comes to comparing string values. Such comparisons respect the case of each letter. In our example, the user must type the state name exactly as it is stored in the `USStates` array for the match to be found. In Chapter 12, you learn about some helper methods that eliminate case and sensitivity in string comparisons.

When a match is found, the statement nested inside the `if` construction runs. The `break` statement is designed to help control structures bail out if the program needs it. For this application, it is imperative that the `for` loop stop running when a match for the state name is found. When the `for` loop breaks, the value of the `i` counter is fixed at the row of the `USStates` array containing the entered state. We need that index value to find the corresponding entry in the other array. Even though the counting variable, `i`, is initialized in the `for` loop, it is still alive and in the scope of the function for all statements after the initialization. That's why we can use it to extract the value of the row of the `stateEntered` array in the final statement that displays the results in an alert message.

If the entered state name doesn't match any of the values in the `USStates` array, the counter variable `i` increments to the length of the array (51) and the `for` loop terminates, leaving `i` equal to 51. From

that we can easily tell whether we successfully found a state name match and display an error message if no match was found.

This application of a `for` loop and array indexes is a common one in JavaScript. Study the code carefully, and be sure you understand how it works. This way of cycling through arrays plays a role not only in the kinds of arrays you create in your code, but also in the arrays that browsers generate for the DOM.

Document objects in arrays

If you look at the `document` object portions of the Quick Reference in Appendix A, you can see that the properties of some objects are listed with square brackets after them. These are indeed the same kind of square brackets you just saw for array indexes. That's because when a document loads, the browser creates arrays of like objects in the document. For example, if your page includes two `<form>` tag sets, two forms appear in the document. The browser maintains an array of `form` objects for that document. References to those forms are

```
document.forms[0]
document.forms[1]
```

You can also capture this array using the DOM method `getElementsByTagName()`:

```
var aForms = document.getElementsByTagName('form');
```

and then refer to the form array items with:

```
aForms[0]
aForms[1]
```

Index values for objects are assigned according to the loading order of the objects. In the case of `form` objects, the order is dictated by the order of the `<form>` tags in the document. This indexed array syntax is another way to reference forms in an object reference. You can still use a form's identifier (`id` attribute) if you prefer — and we heartily recommend using object names wherever possible, because even if you change the physical order of the objects in your HTML, references that use names still work without modification. But if your page contains only one form, you can use the reference types interchangeably, as in the following examples of equivalent references to the `length` property of a form's `elements` array (the `elements` array contains all the form controls in the form):

```
document.getElementById("entryForm").elements.length
document.forms[0].elements.length
document.getElementsByTagName('form')[0].elements.length
```

In examples throughout this book, you can see that we often use the array type of reference to simple forms in simple documents. But in our production pages, we almost always use named references.

Exercises

1. With your newly acquired knowledge of functions, event handlers, and control structures, use the script fragments from this chapter to complete the page that has the lookup table for all the states and the years they entered into the union. If you do not have a reference book for the dates, use different year numbers, starting with 1800 for each entry. In the page, create a text entry field for the state and a button that triggers the lookup in the arrays.

2. Examine the following function definition. Can you spot any problems with the definition? If so, how can you fix the problems?

```
function format(ohmage)
{
   var result;
   if ohmage >= 1e6
   {
      ohmage = ohmage / 1e6;
      result = ohmage + " Mohms";
   }
   else
   {
      if (ohmage >= 1e3)
         ohmage = ohmage / 1e3;
         result = ohmage + " Kohms";
      else
         result = ohmage + " ohms";
}
alert(result);
```

3. Devise your own syntax for the scenario of looking for a ripe tomato at the grocery store, and write a `for` loop using that object and property syntax.
4. Modify Listing 9-2 so that it does not reuse the `Dog` variable inside the function.
5. Given the following table of data about several planets of our solar system, create a web page that enables users to enter a planet name and, at the click of a button, have the distance and diameter appear either in an alert box or (as extra credit) in separate fields of the page.

Planet	Distance from the Sun	Diameter
Mercury	36 million miles	3,100 miles
Venus	67 million miles	7,700 miles
Earth	93 million miles	7,920 miles
Mars	141 million miles	4,200 miles

CHAPTER

10

Window and Document Objects

IN THIS CHAPTER

What the window object does

How to access key window object properties and methods

How to trigger script actions after a document loads

The purposes of the location and navigator objects

How the document object is created

How to access key document object properties and methods

Now that you have exposure to programming fundamentals, it is easier to demonstrate how to script objects in documents. Starting with this lesson, the tutorial turns back to the Document Object Model (DOM), diving more deeply into the objects you will place in many of your documents.

Top-Level Objects

As a refresher, study the hierarchy of top-level objects in Figure 10-1. This chapter focuses on objects of this level that you'll frequently encounter in your scripting: `window`, `location`, `navigator`, and `document`. The goal is not only to equip you with the basics so you can script simple tasks, but also to prepare you for the in-depth examinations of each object and its properties, methods, and event handlers, that you will find in Part IV, "Document Objects Reference." We introduce only the basic properties, methods, and events for objects in this tutorial. Examples in Part IV of the book assume that you know the programming fundamentals covered here in Part II.

The window Object

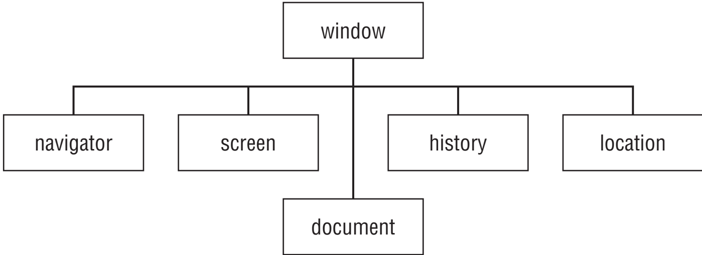
At the top of the object hierarchy is the `window` object. This object gains that exalted spot in the object food chain because it is the master container for all content you view in the web browser. As long as a browser window is open — even if no document is loaded in the window — the `window` object is defined in the current model in memory.

In addition to the content part of the window where documents go, a window's sphere of influence includes the dimensions of the window and all the stuff that surrounds the content area. The area where scrollbars, toolbars, the status bar, and (non-Macintosh) menu bar live is known as a window's *chrome*. Not every browser has full scripted control over the chrome of the main browser window,

but you can easily script the creation of additional windows sized the way you want and with only the chrome elements you wish to display.

FIGURE 10-1

The top-level browser object model for all scriptable browsers.



Although the discussion of frames comes in Chapter 13, "Scripting Frames and Multiple Windows," we can now safely say that each frame is also considered a `window` object. If you think about it, that makes sense, because each frame can hold a different document. When a script runs in one of those documents, it regards the frame that holds the document as the `window` object in its view of the object hierarchy.

As you will learn in this chapter, the `window` object is a convenient place for the DOM to attach methods that display modal dialog boxes and adjust the text that displays in the status bar at the bottom of the browser window. A `window` object method enables you to create a separate window that appears onscreen. When you look at all of the properties, methods, and events defined for the `window` object (see Chapter 27, "Window and Frame Objects"), it should be clear why they are attached to window objects: Visualize their scope and the scope of a browser window.

Note

In the modern crop of browsers, the user is given the choice of opening web pages either in new *tabs* within one browser or in new *browser windows*, which are separate copies of the browser program on the desktop. JavaScript does not exert any control whatsoever over this choice, which remains the user's prerogative. Browser windows and browser tabs are both identically `window` objects to JavaScript. Efforts to control features such as window size and the appearance of toolbars will either fail or will frustrate the user's attempts to make their browser conform to their own personal usability standards. In other words, if you don't want to irritate web site visitors, don't try to mess with their settings. Trust users to manage their browsers. Focus your creative energies on the content and style of the page, not the window chrome. ■

Accessing window properties and methods

You can word script references to properties and methods of the `window` object in several ways, depending more on whim and style than on specific syntactical requirements. The most logical and common way to compose such references includes the `window` object in the reference:

```
window.propertyName
window.methodName([parameters])
```

A window object also has a synonym when the script doing the referencing points to the window that houses the document. The synonym is self. Then the reference syntax becomes

```
self.propertyName
self.methodName([parameters])
```

You can use these initial reference object names interchangeably, but we tend to reserve the use of self for more complex scripts that involve multiple frames and windows. The self moniker more clearly denotes the current window holding the script's document, and can make the script more readable — by us and by others.

Back in Chapter 6, "Browser and Document Objects," we indicated that because the window object is always there when a script runs, you can omit it from references to any objects inside that window. Therefore, the following syntax models assume properties and methods of the current window:

```
propertyName
methodName([parameters])
```

In fact, as you will see in a few moments, some methods may be more understandable if you omit the window object reference. The methods run just fine either way.

Creating a window

A script does not create the main browser window. A user does that by virtue of launching the browser or by opening a URL or file from the browser's menus (if the window is not already open). But a script can generate a large number of subwindows when the main window is open (and that window contains a document whose script needs to open subwindows).

The method that generates a new window is window.open(). This method contains up to three parameters that define window characteristics: the URL of the document to load, its name for target attribute reference purposes in HTML tags, and physical appearance (size and chrome contingent). We don't go into the details of the parameters here (they're covered in great depth in Chapter 27), but we do want to expose you to an important concept involved with the window.open() method.

Consider the following statement, which opens a new window to a specific size and with an HTML document from the same server directory that holds the current page:

```
var subWindow = window.open("define.html","def","height=200,width=300");
```

The important thing to note about this statement is that it is an assignment statement. Something gets assigned to that variable subWindow. What is it? It turns out that when the window.open() method runs, it not only opens that new window according to specifications set as parameters, but also evaluates to a reference to that new window. In programming parlance, the method is said to *return a value* — in this case, a genuine object reference. The value returned by the method is assigned to the variable.

Now your script can use that variable as a valid reference to the second window. If you need to access one of its properties or methods, you must use that reference as part of the complete reference. For example, to close the subwindow from a script in the main window, use this reference to the close() method for that subwindow:

```
subWindow.close();
```

If you issue `window.close()`, `self.close()`, or just `close()` in the main window's script, the method closes the main window (after confirming with the user), but not the subwindow. To address another window, then, you must include a reference to that window as part of the complete reference. This has an impact on your code because you probably want the variable holding the reference to the subwindow to be valid as long as the main document is loaded into the browser. For that to happen, the variable has to be initialized as a global variable, rather than inside a function (although you can set its value inside a function). That way, one function can open the window while another function closes it.

Listing 10-1 is a page that has a button for opening a blank new window and a button for closing that window from the main window. To view this demonstration, shrink your main browser window to less than full screen. Then, when the new window is generated, reposition the windows so you can see the smaller, new window when the main window is in front. (If you lose a window behind another, use the browser's Window menu to choose the hidden window.) The key point of Listing 10-1 is that the `newWindow` variable is defined as a global variable so that both the `makeNewWindow()` and `closeNewWindow()` functions have access to it. When a variable is declared with no value assignment, its initial value is `null`. A `null` value is interpreted to be the same as `false` in a condition, whereas the presence of any nonzero value is the same as `true` in a condition. Therefore, in the `closeNewWindow()` function, the condition tests whether the window has been created before issuing the subwindow's `close()` method. Then, to clean up, the function sets the `newWindow` variable to `null` so that another click of the Close button doesn't try to close a nonexistent window.

Note

The property assignment event-handling technique employed throughout the code in this chapter, and much of the book, is `addEvent()`, a cross-browser event handler explained in detail in Chapter 32, "Event Objects."

The `addEvent()` function is part of the script file `jsb-global.js`, located on the accompanying CD-ROM in the `Content/` folder where it is accessible to all chapters' scripts. ■

LISTING 10-1

References to Window Objects

HTML: jsb-10-01.html

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Window Open &amp; Close</title>
      <script type="text/javascript" src="../jsb-global.js"></script>
      <script type="text/javascript" src="jsb-10-01.js"></script>
   </head>
   <body>
      <h1>Window Open &amp; Close</h1>
      <form>
         <p>
            <input type="button" id="create-window" value="Create New Window">
            <input type="button" id="close-window" value="Close New Window">
         </p>
      </form>
```

```
   </body>
</html>
```

JavaScript: jsb-10-01.js

```
// initialize when the page has loaded
addEvent(window, 'load', initialize);

function initialize()
{
   // do this only if the browser can handle DOM methods
   if (document.getElementById)
   {
      // point to critical elements
      var oButtonCreate = document.getElementById('create-window');
      var oButtonClose = document.getElementById('close-window');

         // if they all exist...
         if (oButtonCreate && oButtonClose)
         {
            // apply behaviors
            oButtonCreate.onclick = makeNewWindow;
            oButtonClose.onclick = closeNewWindow;
         }
   }
}

var newWindow;

function makeNewWindow()
{
   newWindow = window.open("","","height=300,width=300");
}

function closeNewWindow()
{
   if (newWindow)
   {
      newWindow.close();
      newWindow = null;
   }
}
```

window Properties and Methods

The three methods for the `window` object described in this section have an immediate impact on user interaction by displaying dialog boxes of various types. They work with all scriptable browsers. You can find extensive code examples in Part IV for each property and method. You can also experiment with the one-statement script examples by entering them in the top text box of The Evaluator Jr. (from Chapter 8, "Programming Fundamentals, Part I").

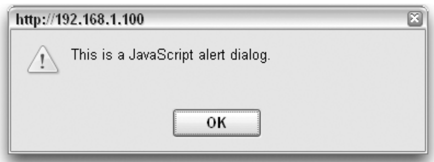
One of the first questions that new scripters ask is how to customize the title bars, sizes, and button labels of these dialog boxes. Each browser maker dictates how these dialogs are labeled. Because tricksters have tried to use these dialog boxes for nefarious purposes over the years, browser makers now go to great lengths to let users know that the dialog boxes emanate from web page scripts. Scripters cannot alter the user interfaces of these dialog boxes.

window.alert() method

We use the `alert()` method many times in this tutorial. This window method generates a dialog box that displays whatever text you pass as a parameter (see Figure 10-2). A single OK button (whose label you cannot change) enables the user to dismiss the alert.

FIGURE 10-2

A JavaScript alert dialog box.



All three dialog-box methods are good cases for using a `window` object's methods without the reference to the window. Even though the `alert()` method technically is a `window` object method, no special relationship exists between the dialog box and the window that generates it. In production scripts, we usually use the shortcut reference:

```
alert("This is a JavaScript alert dialog.");
```

window.confirm() method

The second style of dialog box presents two buttons (Cancel and OK in most versions on most platforms) and is called a confirm dialog box (see Figure 10-3). More important, this is one of those methods that returns a value: `true` if the user clicks OK or `false` if the user clicks Cancel. You can use this dialog box and its returned value as a way to have a user make a decision about how a script progresses.

FIGURE 10-3

A JavaScript confirm dialog box (IE7/WinXP style).

Because the method always returns a Boolean value, you can use the evaluated value of the entire method as a condition statement in an `if` or `if...else` construction. For example, in the following

code fragment, the user is asked about starting the application over. Doing so causes the `index.html` page to load into the browser.

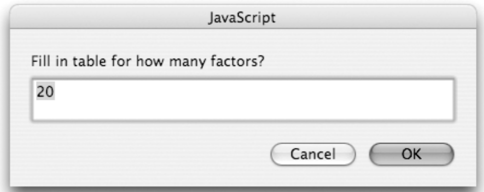
```
if (confirm("Are you sure you want to start over?"))
{
    location.href = "index.html";
}
```

window.prompt() method

The final dialog box of the `window` object, the prompt dialog box (see Figure 10-4), displays a message that you set and provides a text field for the user to enter a response. Two buttons, Cancel and OK, enable the user to dismiss the dialog box with two opposite expectations: canceling the entire operation or accepting the input typed in the dialog box.

FIGURE 10-4

A JavaScript prompt dialog box (Safari 2 style).



The `window.prompt()` method has two parameters. The first is the message that acts as a prompt to the user. You can suggest a default answer in the text field by including a string as the second parameter. If you don't want any default answer to appear, include an empty string (two double quotes without any space between them).

This method returns one value when the user clicks either button. A click of the Cancel button returns a value of `null`, regardless of what the user types in the field. A click of the OK button returns a string value of the typed entry. Your scripts can use this information in conditions for `if` and `if...else` constructions. A value of `null` is treated as `false` in a condition. It turns out that an empty string is also treated as `false`. Therefore, a condition can easily test for the presence of real characters typed in the field to simplify a conditional test, as shown in the following fragment:

```
var answer = prompt("What is your name?","");
if (answer)
{
   alert("Hello, " + answer + "!");
}
```

In this example, the only time the `alert()` method is called is when the user enters something in the prompt dialog box and clicks the OK button.

load event

The `window` object reacts to several system and user events, but the one you will probably use most often is the event that fires as soon as everything in a page finishes loading. This event waits for

images, Java applets, and data files for plug-ins to download fully to the browser. It can be dangerous to script access to elements of a document object while the page loads because if the object has not loaded yet (perhaps due to a slow network connection or server), a script error results. The advantage of using the `load` event to invoke functions is that you are assured that all document objects are in the browser's DOM.

The `load` event handler can be applied to the `window` object in several ways, depending on the browser and the circumstances:

```
window.addEventListener('load', functionName, false);
window.attachEvent('onload', functionName);
```

where `functionName` is the function you've written that you want to run as soon as the page has downloaded. (For a cross-browser event-adding function, see Chapter 32.) `addEventListener` and `attachEvent` can be called multiple times to add more than one function to the list to be executed when the page has loaded.

You can also apply the behavior directly to the element:

```
window['onload'] = functionName;
window.onload = functionName;
```

However, this usage dictates that there will be only one function to run when the page is loaded, replacing any event handler already assigned to the `window` object.

In old-school legacy web pages, you'll sometimes find the window event handler applied to the `body` element right in the HTML:

```
<body onload="functionName()">
```

(Even though you will come to associate the `<body>` tag's attributes with the `document` object's properties, it is the `window` object's event handlers that go inside the tag.) This embedding of JavaScript into the fabric of the HTML is considered poor usage today for several reasons: it doesn't let browsers that can't handle the scripting fail gracefully, it makes the HTML file heavier than it has to be, and it makes both the script and the markup messier and more time-consuming to modify. Separation of layers is the way to go.

Cross-Reference

For more on the `window.onload` event, see Chapter 27, "Window and Frame Objects." ■

The location Object

Sometimes an object in the hierarchy represents something that doesn't seem to have the kind of physical presence that a window or a button does. That's the case with the `location` object. This object represents the URL loaded into the window. This differs from the `document` object (discussed later in this lesson): the document is the content of the page, while the location is its URL (Uniform Resource Locator or address).

A URL consists of many components that define the address and method of data transfer for a file. Pieces of a URL include the protocol (such as `http:`) and the hostname (such as `www.example.com`). You can access all these items as properties of the `location` object. For the

most part, though, your scripts will be interested in only one property: the `href` property, which defines the complete URL.

Setting the `location.href` property is the primary way your scripts navigate to other pages:

```
location.href = "http://www.example.com/contact.html";
```

For pages outside the domain of the current page, you need to specify the complete URL. You can generally navigate to a page in your own web site by specifying a relative URL (that is, relative to the currently loaded page) rather than the complete URL with protocol and host information.

```
location.href = "contact.html";   // relative URL
```

If the page to be loaded is in another window or frame, the window reference must be part of the statement. For example, if your script opens a new window and assigns its reference to a variable named `newWindow`, the statement that loads a page into the subwindow is

```
newWindow.location.href = "http://www.example.com";
```

The navigator Object

Despite a name reminiscent of the Netscape Navigator-branded browser, the `navigator` object is implemented in all scriptable browsers. All browsers also implement a handful of properties that reveal the same kind of information that browsers send to servers with each page request. Thus, the `navigator.userAgent` property returns a string with numerous details about the browser and operating system. For example, a script running in Internet Explorer 8 in Windows XP receives the following value for the `navigator.userAgent` property:

```
Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 5.1)
```

The same script running in Firefox 3.5.2 on a Macintosh reveals the following `userAgent` details:

```
Mozilla/5.0 (Macintosh; U; PPC Mac OS X 10.4; en-US; rv:1.9.1.2)
    Gecko/20090729 Firefox/3.5.2
```

Cross-Reference

See Chapter 42, "Navigator and Other Environment Objects" (on the CD), for more details about the `navigator` object and the meaning of the values returned by its properties. Unfortunately, it cannot be trusted to accurately report the actual user agent make, model, and version. It once was used extensively to branch script execution according to various browser versions. Chapter 25, "Document Object Model Essentials," describes more modern ways to detect the capabilities of the browser. ■

The document Object

The `document` object holds the real content of the page. Properties and methods of the `document` object generally affect the look and content of the document that occupies the window. As you saw in Chapter 5, "Your First JavaScript Script," all W3C DOM-compatible browsers allow script access

to the text contents of a page when the document has loaded. You've also seen that DOM methods let a script create content dynamically after the page has loaded. Many `document` object properties are arrays of other objects in the document, which provide additional ways to reference these objects (over and above the `document.getElementById()` method).

Accessing a `document` object's properties and methods is straightforward, as shown in the following syntax examples:

```
[window.]document.propertyName
[window.]document.methodName([parameters])
```

The `window` reference is optional when the script is accessing the `document` object that contains the script. If you want a preview of the long list of `document` object properties of IE or a Mozilla-based browser, enter `document` in the object text box of The Evaluator Jr. and press Enter/Return. The object's properties, current values, and value types appear in the Results box (as well as methods in Mozilla). Following are some of the most commonly used properties and methods of the `document` object.

document.getElementById() method

You met the `document.getElementById()` method in Chapter 5 when learning about the syntax for referencing element objects. This W3C DOM method is one you will use a lot. Get to know its finger-twisting name well. Be sure to honor the upper- and lowercase spelling of this all-important method.

The sole parameter of this method is a quoted string containing the ID of the element you wish to reference. The Evaluator Jr. page from Chapter 8 (and in the listings on the CD-ROM) has several element objects with IDs, including `input`, `output`, and `inspector`. Type this method in the top text box with each ID, as in the following example:

```
document.getElementById("output")
```

The method returns a value, which you typically preserve in a variable for use by subsequent script statements — for example:

```
var oneTable = document.getElementById("salesResults");
```

After the assignment statement, the variable represents the element object, allowing you to get and set its properties or invoke whatever methods belong to that type of object.

document.getElementsByTagName() method

`getElementsByTagName()` is a handy method for collecting an array of page elements that share the same tag name. For example, to get a list of all of the images on the page, we could call:

```
var aImages = document.getElementsByTagName('img');
```

Referring to an image by its position in an array of all images can be challenging — for example, if we want to work with an image gallery on a page that also contains miscellaneous images before and after the gallery. One of the advantages of using `getElementsByTagName()` is that it can gather a collection within any container element on the page:

```
HTML:

<div id="gallery">
   <h2>My Gallery</h2>
   <img src="bat.jpg" alt="bat">
   <img src="cat.jpg" alt="cat">
   <img src="hat.jpg" alt="hat">
</div>

JavaScript:

var oGallery = document.getElementById('gallery');
var aCollection = oGallery.getElementsByTagName('img');
```

The array `aCollection()` in this example then contains the three image objects (but not, of course, the `h2` element). As you'll recall from the discussion of arrays in Chapter 9, "Programming Fundamentals, Part II," an index number inside an array's square brackets points to one of the elements in the array. To find out how many objects are in the collection, use

```
aCollection.length
```

Each object can be accessed by its offset into the array:

```
oImage = aImages[0];   // first image
oImage = aImages[1];   // second image
```

Other applications of this method might be to collect all the items in a list, all the hyperlinks (anchors) in a text block, or all the multimedia objects on a page.

document.forms[] property

Implemented back in pre-DOM days, the `document.forms` property of the document object contains an array of all `form` element objects in the document. Compare these two expressions:

```
var aForms = document.forms;
var aForms = document.getElementsByTagName('form');
```

One important difference is that the `document.forms` collection enables us to reference a particular form directly by its name, not just by its offset in the array. It isn't always practical to refer to a form by its index number. A dynamic web page might contain a varying number of forms depending on context, so the position of any one form on the page might change with circumstances. For example, a Search form that sits in the header chrome of every page of a web site might be suppressed on the Advanced Search page; a Join List form might be suppressed on the Contact page; and a page listing workshops might blossom with registration forms for selected workshops.

Scriptable browsers let you refer to a form more directly by its name or ID (that is, the identifier assigned to either the `name` or `id` attribute of the `<form>` tag):

```
<form id="formId" name="formName" ...>
```

The first way is by using the `getElementById()` method:

```
document.getElementById("formId")
```

The second way uses array syntax, applying the form's name or ID as a string index value of the array:

```
document.forms["formId"]
document.forms["formName"]
```

A third, even shorter way to refer to a `form` object by name is to append the name as a property of the `document` object, as in:

```
document.formName
```

However, this last technique works only if the `name` attribute omits several characters that are legal in HTML element names but illegal in JavaScript object names such as hyphen (`-`), period (`.`), and colon (`:`). (For more on this point, see Chapter 6.)

Any of these methodologies reaches the same object. We will primarily be using the DOM methods `getElementById()` and `getElementsByTagName()` throughout this book; however, the `document.forms` collection syntax, which dates back to the earliest scriptable browsers, is still valid in the most modern versions.

document.images[] property

Just as a document keeps track of forms in a special array property, the `document` object maintains a collection (array) of images inserted into the document by way of `<img>` tags. Images referenced through the `document.images` array may be reached by either numeric or string index of the `img` element's name. Just as with forms, the `name` attribute value is the identifier you use for a string index.

The presence of the `document.images` property indicates that the browser supports image objects and therefore scripting techniques such as image-swapping. Thus, you can use the existence of the property as a test condition to make sure the browser supports images as objects before attempting to perform any script action on an image. To do so, surround statements that deal with images with an `if` construction that verifies the property's existence, as follows:

```
if (document.images)
{
    // statements dealing with img objects
}
```

Older browsers skip the nested statements, preventing them from displaying error messages to their users.

In a typical, complex web page today, there are images in different sections used for very different purposes, and it doesn't often make sense to collect all the images on a page in a single array. In cases like this, it might be more convenient to use the DOM method `getElementsByTagName()` instead (see above).

document.createElement() and document.createTextNode() methods

Adding a new element to an HTML document consists of, at minimum, two steps:

1. Create the new element for the document.
2. Insert it exactly where you want it within the tree structure of the page.

The `document.createElement()` method lets you create a brand-new element object in the browser's memory. To specify the element you wish to create, pass the tag name of the element as a string parameter of the method:

```
var newElem = document.createElement("p");
```

You may also want to add some attribute values to the element, which you may do by assigning values to the newly created object's properties, even before the element becomes part of the document.

```
newElem.setAttribute("class", "intro");
```

This element can be inserted wherever you want in the document — for example, at the end:

```
document.body.appendChild(newElem);
```

These three lines of code generate a paragraph at the end of the document body:

```
<body>
...
<p class="intro"></p></body>
```

As you saw in the object hierarchy illustrations in Chapter 6, "Browser and Document Objects," an element object frequently needs text content between its start and end tags. The W3C DOM way to create that text is to generate a brand new text node via the `document.createTextNode()` method and populate the node with the desired text. For example:

```
var newText = document.createTextNode("Greetings to all.");
newElem.appendChild(newText);
```

resulting in:

```
<body>
...
<p class="intro">Greetings to all.</p></body>
```

As you can see, the act of creating an element or text node does not by itself influence the document node tree. You must invoke one of the various insertion or replacement methods to place the new text node in its element and place the element in the document, as you learned how to do in Chapter 5.

document.write() method

The `document.write()` method is another way of writing content to a document. It has the advantage of being "quick and dirty," but it also has several disadvantages:

- `document.write()` can add a piece of a web page only while the page is loading into the browser the first time. Any subsequent invocation of the method replaces the entire page.
- Because `document.write()` will insert any old block of text into the document, it enables sloppy programming habits and malformed markup, encourages the mixture of markup with content in scripting, and does not encourage the separation of development layers (structure from content).
- `document.write()` does not work with XHTML, a document type that doesn't permit its content to be modified during the initial parsing.

For these reasons we do not recommend that you use `document.write()` in your own code, though it's still useful to understand how it works in order to be able to read legacy code.

The `document.write()` method operates both in immediate scripts to create content in a page as it loads and in deferred scripts that create new content in the same window or in a different window. The method requires one string parameter, the HTML content to be written to the window or frame. Such string parameters can be variables or any other expressions that evaluate to a string. Very often, the written content includes HTML tags.

Bear in mind that after a page loads, the browser's *output stream* automatically closes. After that, any `document.write()` method issued to the current page opens a new stream that immediately erases the current page (along with any variables or other values in the original document). Therefore, if you wish to replace the current page with script-generated HTML, you need to accumulate that HTML in a variable and perform the writing with just one `document.write()` method. You don't have to clear a document explicitly and open a new data stream; one `document.write()` call does it all.

One last piece of housekeeping advice about the `document.write()` method involves its companion method, `document.close()`. Your script must close the output stream when it finishes writing its content to the window (either the same window or another). After the last `document.write()` method in a deferred script, be sure to include a `document.close()` method. Failure to do this may cause images and forms not to appear. Also, any `document.write()` method invoked later will only append to the page, rather than clear the existing content to write anew.

To demonstrate the `document.write()` method, we show two versions of the same application. One writes to the same document that contains the script; the other writes to a separate window. Type the code for each document in a new text editor file, save them, and open the `.html` file in your browser.

Listing 10-2 creates a button that assembles new HTML content for a document, including HTML tags for new document title and color attributes for the `<body>` tag. An operator in the listing that may be unfamiliar to you is +=. It appends a string on its right side to whatever string is stored in the variable on its left side. This operator is a convenient way to accumulate a long string across several separate statements. With the content gathered in the `newContent` variable, one `document.write()` statement blasts the entire new content to the same document, obliterating all vestiges of the content of Listing 10-2. The `document.close()` statement, however, is required to close the output stream properly. When you load this document and click the button, notice that the document title in the browser's title bar changes accordingly. As you click back to the original and try the button again, notice that the dynamically written second page loads much faster than even a reload of the original document.

LISTING 10-2

Using document.write() on the Current Window

HTML: jsb-10-02.html

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Writing to Same Doc</title>
```

```
      <script type="text/javascript" src="../jsb-global.js"></script>
      <script type="text/javascript" src="jsb-10-02.js"></script>
   </head>
   <body>
      <h1>Writing to Same Doc</h1>
      <form>
         <p>
            <input type="button" id="rewritePage" value="Replace Content">
         </p>
      </form>
   </body>
</html>
```

JavaScript: jsb-10-02.js

```
// replace the page with new markup
function reWrite()
{
 // assemble content for new window
 var newContent = '<!DOCTYPE html>';
 newContent += '<html>';
 newContent += '<head>';
 newContent += '<meta http-equiv="content-type"
    content="text/html;charset=utf-8">';
 newContent += '<title>A New Doc</title>';
 newContent += '<style type="text/css">';
 newContent += 'body { background-color: aqua; }';
 newContent += '</style>';
 newContent += '</head>';
 newContent += '<body>';
 newContent += '<h1>This document is brand new.</h1>';
 newContent += '<p>Click the Back button to see the original document.</p>';
 newContent += '</body>';
 newContent += '</html>';

 // write HTML to new window document
 document.write(newContent);
 document.close(); // close layout stream
}

// apply behaviors when document has loaded
function initialize()
{
 // do this only if the browser can handle DOM methods
 if (document.getElementById)
 {
    // point to the button
    var oButtonRewrite = document.getElementById('rewritePage');

       // if it exists...
       if (oButtonRewrite)
       {
```

continued

LISTING 10-2 *(continued)*

```
            // apply event handler
            addEvent(oButtonRewrite, 'click', reWrite);
        }
  }
}

// initialize when the page has loaded
addEvent(window, 'load', initialize);
```

In Listing 10-3, the situation is a bit more complex because the script generates a subwindow to which an entirely script-generated document is written.

Note

You will have to turn off pop-up window blocking temporarily to run this script. ■

To keep the reference to the new window alive across both functions, the `newWindow` variable is declared as a global variable. As soon as the page loads, the `onload` event handler invokes the `makeNewWindow()` function. This function generates a blank subwindow. We added a property to the third parameter of the `window.open()` method that instructs the status bar of the subwindow to appear.

A button in the page invokes the `subWrite()` method. The first task it performs is to check the `closed` property of the subwindow. This property returns `true` if the referenced window is closed. If that's the case (if the user closed the window manually), the function invokes the `makeNewWindow()` function again to reopen that window.

With the window open, new content is assembled as a string variable. As with Listing 10-2, the content is written in one blast (although this isn't necessary for a separate window), followed by a `close()` method. But notice an important difference: Both the `write()` and `close()` methods explicitly specify the subwindow.

LISTING 10-3

Using document.write() on Another Window

HTML: jsb-10-03.html

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Writing to Subwindow</title>
      <script type="text/javascript" src="../jsb-global.js"></script>
      <script type="text/javascript" src="jsb-10-03.js"></script>
   </head>
   <body>
      <h1>Writing to Subwindow</h1>
      <form>
```

```
         <p>
            <input type="button" id="writeSubwindow" value="Write to Subwindow">
         </p>
      </form>
   </body>
</html>
```

JavaScript: jsb-10-03.js

```
var newWindow;

function makeNewWindow()
{
   newWindow = window.open("","","status,height=200,width=300");
}

function subWrite()
{
    // make new window if someone has closed it
    if (!newWindow || newWindow.closed)
    {
       makeNewWindow();
    }

 // bring subwindow to front
 newWindow.focus();

 // assemble content for new window
 var newContent = '<!DOCTYPE html>';
 newContent += '<html>';
 newContent += '<head>';
 newContent += '<meta http-equiv="content-type"
     content="text/html;charset=utf-8">';
 newContent += '<title>A New Doc</title>';
 newContent += '<style type="text/css">';
 newContent += 'body { background-color: aqua; }';
 newContent += '</style>';
 newContent += '</head>';
 newContent += '<body>';
 newContent += '<h1>This document is brand new.</h1>';
 newContent += '</body>';
 newContent += '</html>';

 // write HTML to new window document
 newWindow.document.write(newContent);

 // close layout stream
 newWindow.document.close();
}

// apply behaviors when document has loaded
function initialize()
{
```

continued

LISTING 10-3 *(continued)*

```
  // do this only if the browser can handle DOM methods
  if (document.getElementById)
  {
     // point to the button
     var oButtonRewrite = document.getElementById('writeSubwindow');

        // if it exists...

        if (oButtonRewrite)
        {
           // apply event handler
           addEvent(oButtonRewrite, 'click', subWrite);
        }
  }
}

// initialize when the page has loaded
addEvent(window, 'load', initialize);
addEvent(window, 'load', makeNewWindow);
```

The next logical step after the document level in the object hierarchy is the form. That's where you will spend the next lesson.

Exercises

1. Which of the following references are valid, and which are not? Explain what is wrong with the invalid references.
   a. `window.document.form[0]`
   b. `self.entryForm.submit()`
   c. `document.forms[2].name`
   d. `document.getElementByID("firstParagraph")`
   e. `newWindow.document.write("Howdy")`
2. Write the JavaScript statement that displays an (annoying) dialog box welcoming visitors to your web page.
3. Write the JavaScript statement that executes while the page loads to display the same message from Question 2 to the document as an `<h1>`-level headline on the page.
4. Create a page that prompts the user for his or her name as the page loads (via a dialog box) and then welcomes the user by name in the body of the page.
5. Create a page with any content you like, but one that automatically displays a dialog box after the page loads to show the user the URL of the current page.

CHAPTER 11

Forms and Form Elements

IN THIS CHAPTER

What the form object represents

How to access key form object properties and methods

How text, button, and select objects work

How to submit forms from a script

How to pass information from form elements to functions

Most interactivity between a web page and the user takes place inside a form. That's where a lot of the interactive HTML stuff lives for every browser: text fields, buttons, checkboxes, option lists, and so on.

In this chapter, we discuss how to locate forms and their controls in the document tree, how to change them, how to examine the user's input, and how to submit a form or suppress submission if the input doesn't validate.

The Form object

A form and the input controls that go inside it are DOM objects with unique properties that other objects in the document don't have. For example, a form object has an `action` property that tells the browser where to send input values when a form is submitted. A `select` control (drop-down list) has a `selectedIndex` property that tells us which option has been selected by the user.

Our first step is to point to the form on the page. Here are three ways of doing this, all referring to the following abbreviated snippet of a page that contains three forms:

```
<div id="header">
   <form id="search" action="...">...</form>
   <form id="join-list" action="...">...</form>
</div>
   ...
   <form id="contact" action="...">...</form>
   ...
```

1. The DOM method `getElementById()`, as described in previous chapters, gives us a handle on a single, specific element on the page if we know its `id` attribute:

   ```
   var oForm = document.getElementById('search');
   ```

In practice, using `getElementById()` suffices most of the time because web pages generally contain one or a very small number of unique forms which accomplish very different tasks; assigning them unique IDs in the markup is a natural.

2. The DOM method `getElementsByTagName()` delivers an array or collection of form objects with a given tag name:

```
var aForms = document.getElementsByTagName('form');
var oForm = aForms[0];  // get the first form on the page
```

As always, JavaScript arrays begin with the first item having an index of zero, so our collection of three forms can be addressed as array elements 0, 1, and 2.

It's worth pondering the usefulness of working with a collection of all the forms on a page. In order to locate a specific form from the collection, we would have to know its position among all the forms on the page (which could change in a rich, dynamic site) or we'd have to cycle through them looking for an identifier (which suggests using `getElementById()` in the first place).

One of the cool things about `getElementsByTagName()` is that it can give us a collection of objects within a particular parent other than the `document` element itself:

```
var oHeader = document.getElementById('header');
var aForms = oHeader.getElementsByTagName('form');
var oForm = aForms[0];
```

In this example, we're collecting all the forms in the "header" section of the document only.

Or imagine a page listing a series of workshops, each of which has its own separate registration form. We could collect an array of all those registration forms by first pointing to their parent `div` without worrying about other miscellaneous forms on the page confusing the mix.

3. The syntax `document.forms` is another way of collecting all the forms on the page that pre-dates today's DOM methods but is still supported by modern browsers, so as not to break legacy web sites.

```
var aForms = document.forms;
// then point to one form:
var oForm = aForms[0];
// or:
var oForm = aForms["search"];
// or:
var oForm = aForms.search;
```

Using this original "DOM Level 0" syntax, we can reference a `form` object either by its position in the array of forms contained in a document or by name (if you assign an identifier to the `id` or `name` attribute inside the `<form>` tag). If only one form appears in the document, it is still a member of an array (a one-element array) and is referenced as follows:

```
document.forms[0]
```

Or use the string of the element's name as the array index:

```
document.forms[formName]
```

Notice that the array reference uses the plural version of the word, followed by a set of square brackets containing the index number (zero is always first) or name of the element. Alternatively, you can use the form's name (not as a quoted string) as though it were a property of the `document` object:

```
document.formName
```

However, this last syntax works only if the form `id` or `name` doesn't include characters such as hyphens, periods, or colons that would confuse JavaScript. For example, the second form in our sample HTML above has the `id` of `join-list`:

```
var oForm = document.forms.join-list;
```

This looks to JavaScript like we're trying to subtract the value of a variable named `list` from a `form` object with the `id` of `join`. This statement would either stop JavaScript cold or produce a result of `NaN` (Not a Number). As a general rule, we recommend that you use syntax that puts element names in quotes.

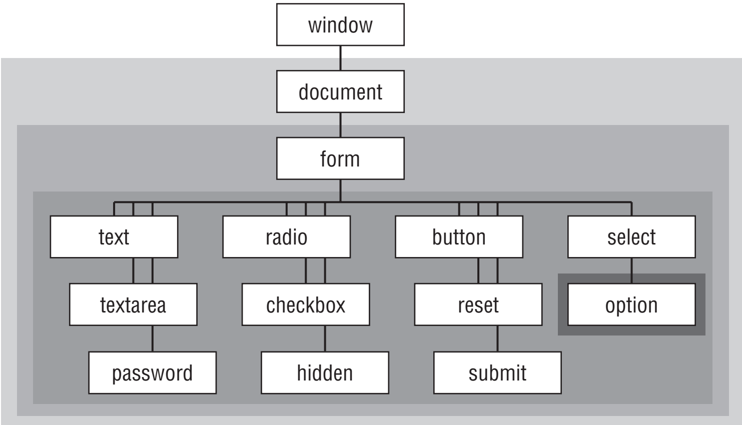
Because there are so many ways to arrive at a `form` object, we'll often represent them in further examples with the label `formObject`.

Form as object and container

Because the modern DOM is backward-compatible with many of yesterday's conventions, the form object finds itself the matriarch of two different family trees at the same time. The modern DOM Level 2 specifies that the form is the parent of all of its child nodes, both element and text nodes, whereas the older DOM Level 0 makes the form the container for all of its form control objects only (`input`, `select`, `button`, and `textarea` elements). Figure 11-1 shows the structure of this DOM 0 hierarchy and its place relative to the `document` object. You'll see the effect this structure has on the way you reference form control elements in a moment.

FIGURE 11-1

DOM Level 0 hierarchy for forms and controls.



Let's illustrate the difference by diagramming the following snippet of HTML:

LISTING 11-1

Sample form markup

```
<form action="search.php" method="post">
   <p>
      <label for="inputSearch">Search for:</label>
      <input id="inputSearch" name="inputSearch" type="text" value="">
      <input id="submit" type="submit" value="Search">
   </p>
</form>
```

Figure 11-2 shows the DOM Level 2 tree for this markup. Note that it represents the entire content of that segment of the document. The carriage returns, tabs, and spaces between elements are text nodes (represented here by the shorthand `"[whitespace]"`).

FIGURE 11-2

DOM Level 2 tree for a typical form.

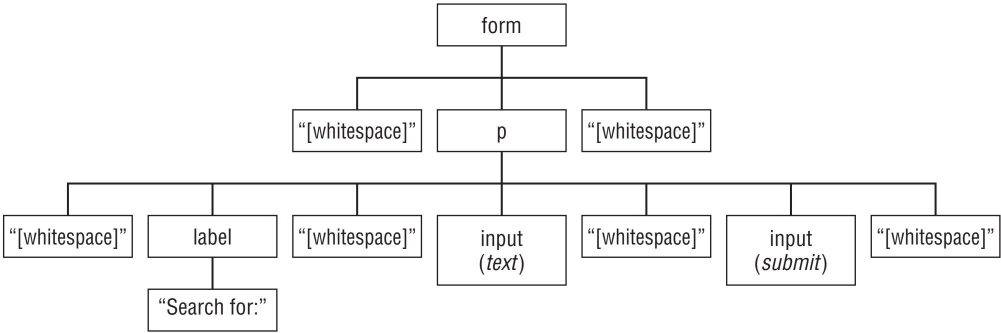
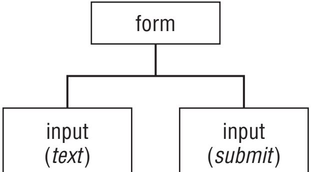


Figure 11-3 shows the DOM Level 0 tree for this same form. It includes only the input controls and omits the paragraph, the label, and the text nodes.

FIGURE 11-3

DOM Level 0 tree for the same form.



These two object trees are clearly useful for very different purposes. The DOM Level 2 tree can be used to read and write the entire document content with fine granularity. The DOM Level 0 tree makes it quick and easy to read and write the form controls only.

Please note that whether you're using DOM 0 or DOM 2 techniques in any given statement, the objects are the same. For example,

```
var oForm = document.getElementById('myForm');
var oControl = oForm.fritzy;
```

is perfectly valid JavaScript and will work fine if `fritzy` is the ID of an input control inside the form `myForm`.

In addition to a large collection of properties and methods it has in common with all HTML element objects, the form object features a number of items that are unique to this object. Almost all of these unique properties are scripted representations of the `form` element's attributes (`action`, `target`, and so on). Scriptable browsers allow scripts to change these properties under script control, which gives your scripts potentially significant power to direct the behavior of a form submission in response to user selections on the page.

Accessing form properties

Forms can be created from standard markup tags in the HTML page or by using DOM methods in JavaScript. Either way, you can set attributes such as `name`, `target`, `action`, `method`, and `enctype`. Each of these is a property of a form object, accessed by all lowercase versions of those words, as in:

```
var sURL = formObject.action;
```

To change any of these properties, simply assign new values to them:

```
formObject.action = "http://www.example.com/cgi/login.pl";
```

These last two JavaScript statements could be re-stated using compound object references:

```
var sURL = document.getElementById('formName').action;
document.forms[0].action = "http://www.example.com/cgi/login.pl";
```

However, combining multiple operations into a single expression doesn't allow the script to test for validity along the way. Separating steps and testing as we go is a more bulletproof way to code:

```
var oForm = document.getElementById ('formName');
   if (!oForm)
   {
      // do something if the named form isn't found
   }
var sURL = oForm.action;
```

form.elements[] property

The `elements[]` property is a collection of all the input controls within a form. This is another array with items listed in the order their HTML tags appear in the source code. It is generally more efficient to refer to a single element directly using its ID, but sometimes a script needs to look through all

of the elements in a form. For example, a form-validation routine might loop through every element checking to make sure that its value has been entered correctly by the user. A loop like that might not need to look at the text nodes and other elements that aren't form controls, and in fact it would have to perform a series of tests on each element it encountered to determine whether it were a control. In a case like this, using the `elements[]` collection is vastly more efficient.

The following code fragment shows the `form.elements[]` property at work in a `for` repeat loop that looks at every control element in a form to set the contents of text fields to an empty string. The script cannot simply barge through the form and set every element's content to an empty string because some elements may be types (for example, a button) whose `value` properties have different purposes.

```
var oForm = document.getElementById('registration-form');
   if (!oForm) return false;

for (var i = 0; i < oForm.elements.length; i++)
{
   if (oForm.elements[i].type == "text")
   {
      oForm.elements[i].value = "";
   }
}
```

In the first statement, we create the variable `oForm` that holds a reference to the desired form. We do this so that when we make many references to form elements later in the script, the typical length of each reference is much shorter (and marginally faster). We can use the `oForm` variable as a shortcut to building references to items more deeply nested in the form.

Next, we start looping through the items in the `elements` array for the form. Each form control element has a `type` property, which reveals what kind of form control it is: text, button, radio, checkbox, textarea, and so on. We're interested in finding elements whose type is `text`. For each of those, we set the `value` property to an empty string.

We're permitting compound expressions such as `oForm.elements[i].type` in this case because, having already tested to make sure the form exists, we can trust the DOM to deliver us a valid form control object with each iteration of the `elements[]` collection. Even if the form were empty of controls, the script wouldn't fail; if the `elements[]` collection were an empty array, the `for` loop above would simply terminate before its first iteration with no error because `oForm.elements.length` would be zero.

We'll return to forms later in this chapter to show you how to submit a form without a Submit button and how client-side form validation works.

Form Controls as Objects

Three kinds of HTML elements nested inside a `<form>` tag become scriptable objects in all browser DOMs. Most of the objects owe their existence to the `<input>` tag in the page's source code. Only the value assigned to the `type` attribute of an `<input>` tag determines whether the element is a text box, password entry field, hidden field, button, checkbox, or radio button. The other two kinds of form controls, `textarea` and `select`, have their own tags.

To reference a particular form control as an object, you can point directly to it using its `id` or `tagName` with DOM Level 2 methods, or, using DOM Level 0 syntax, build a reference as a hierarchy starting with the `document`, through the form, and then to the control. You've already seen how many ways you can reference merely the form part — all of which are valid for building form control references. But if you are using only the identifiers assigned to the form and form control elements (rather than the associated arrays of elements), the syntax is as follows:

```
document.getElementById(controlName)
```

or

```
document.formName.controlName
```

For example, consider the following simple form:

```
<form id="searchForm" action="cgi-bin/search.pl">
   <p>
      <input type="text" id="entry" name="entry">
      <input type="submit" id="sender" name="sender" value="Search">
   </p>
</form>
```

The following sample references to the text input control are all valid:

```
document.getElementById("entry")
document.searchForm.entry
document.searchForm.elements[0]
document.forms["searchForm"].elements["entry"]
document.forms["searchForm"].entry
```

Although form controls have several properties in common, some properties are unique to a particular control type or to related types. For example, only a `select` object offers a property that reveals which item in its list is currently selected. Checkboxes and radio buttons both have a property that indicates whether the control is currently set to on. Similarly, all text-oriented controls operate the same way for reading and modifying their content.

Having a good grasp of the scriptable features of form control objects is important to your success with JavaScript. In the next sections, you meet the most important form control objects and see how scripts interact with them.

Text-related input objects

Each of the four text-related HTML form elements — input elements of the `text`, `password`, and `hidden` types, plus the `textarea` element — is an element in the document object hierarchy. All but the hidden types are normally displayed on the page, enabling users to enter text and select options.

To make these form control objects scriptable in a page, you don't need to do anything special to their normal HTML tags — with the possible exception of assigning an `id` attribute. We strongly recommend assigning both unique IDs and names to every text-related form control element if your scripts will be getting or setting properties, or invoking their methods. IDs are handy for DOM manipulation and for associating labels with controls, and names are necessary for a form to function normally.

When a form is actually submitted to a server-side program, it is the control elements' `name` attributes that are sent to the server along with the elements' values.

For the visible objects in this category, event handlers are triggered from many user actions, such as giving a field focus (getting the text insertion pointer in the field) and changing text (entering new text and then leaving the field). Most of your text-field actions are triggered by the change of text (the `onchange` event handler). In current browsers, events fire in response to individual keystrokes as well.

Without a doubt, the single most-used property of a text-related element is the `value` property. This property represents the current contents of the text element. A script can retrieve and set its content at any time. Content of the `value` property is always a string. This may require conversion to numbers (see Chapter 8, "Programming Fundamentals, Part I") if text fields are used for entering values for math operations.

Text Object Behavior

Many scripters look to JavaScript to solve what are perceived as shortcomings or behavioral anomalies with text-related objects in forms. We want to single these out early in your scripting experience so that they do not confuse you later.

Most browser forms practice a behavior that was recommended long ago as an informal standard by web pioneers. When a form contains only one text `input` object, a press of the Enter/Return key while the text object has focus automatically submits the form. For two or more fields in browsers other than IE5/Mac and Safari, you need another way to submit the form (for example, a Submit button). This one-field submission scheme works well in many cases, such as the search page of most web search sites. But if you are experimenting with simple forms containing only one field, you can submit the form with a press of the Enter/Return key. Submitting a form that has no other action or target specified means the page performs an unconditional reload, wiping out any information entered into the form. You can, however, cancel the submission through an `onsubmit` event handler in the form, as shown later in this chapter. You can also script the press of the Enter/Return key in any text field to submit a form (see Chapter 32, ''Event Objects'').

To demonstrate how a text field's `value` property can be read and written, Listing 11-2 provides a complete HTML page with a single-entry field. When you enter text into the field and press Tab or Enter, the input text is made all uppercase.

LISTING 11-2

Getting and Setting a Text Object's value Property

HTML: jsb-11-02.html

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Text Object value Property</title>
      <script type="text/javascript" src="../jsb-global.js"></script>
      <script type="text/javascript" src="jsb-11-02.js"></script>
```

```
   </head>
   <body>
      <h1>Text Object value Property</h1>
      <form id="UCform" action="make-uppercase.php">
         <p>
            <input type="text" id="converter" name="converter" value="sample">
         </p>
      </form>
   </body>
</html>
```

JavaScript: jsb-11-02.js

```
// initialize when the page has loaded
addEvent(window, 'load', initialize);

var oInput;   // (global) input field to make uppercase

// apply behaviors when document has loaded
function initialize()
{
   // do this only if the browser can handle DOM methods
   if (document.getElementById)
   {
      // apply event handler to the button
      oInput = document.getElementById('converter');
         if (oInput)
         {
            addEvent(oInput, 'change', upperMe);
         }

      // apply event handler to the form
      var oForm = document.getElementById('UCform');
         if (oForm)
         {
            addEvent(oForm, 'submit', upperMe);
         }
   }
}

// make the text UPPERCASE
function upperMe(evt)
{
      // consolidate event handling
      if (!evt) evt = window.event;

   // set input field value to the uppercase version of itself
   var sUpperCaseValue = oInput.value.toUpperCase();
   oInput.value = sUpperCaseValue;

   // cancel default behavior (esp. form submission)
      // W3C DOM method (hide from IE)
```

continued

LISTING 11-2 *(continued)*

```
        if (evt.preventDefault) evt.preventDefault();

    // IE method
    return false;
}
```

Here's how it works: When the page loads into the browser, the `initialize()` function applies event handlers to the form and to the input field it contains. The `onchange` event handler of both the form and the input field invokes the `upperMe()` function, which converts the text to uppercase.

Notice that the `oInput` variable, used to refer to the input field `converter`, is declared outside of all the functions and is therefore a global variable. This means that it's accessible from anywhere in the script. We've done this so that we can use it from two separate functions — `initialize()` and `upperMe()`. (In contrast to global variables, *local variables* are declared with `var` inside of a function and cannot be seen outside of the function in which they were declared.)

The core of the `upperMe()` function consists of two statements:

```
var sUpperCaseValue = oInput.value.toUpperCase();
oInput.value = sUpperCaseValue;
```

A lot goes on in the first statement of the function. The right side of the assignment statement performs a couple of key tasks. The reference to the `value` property of the object (`oInput.value`) evaluates to whatever content is in the text field at that instant. (Remember that `oInput` is that global variable that points to the input field `converter`.) Then that string is handed over to one of JavaScript's string functions, `toUpperCase()`, which converts the value to uppercase. The evaluated result of the right-side statement is then assigned to the second variable: `sUpperCaseValue`. Nothing has changed yet in the text box. That comes in the second statement, where the `value` property of the text box is assigned whatever the `sUpperCaseValue` variable holds. We've divided this logic into two statements for learning purposes so that you can see the process. In practice, you can combine the actions of steps 1 and 2 into one power-packed statement:

```
oInput.value = oInput.value.toUpperCase();
```

The `onchange` event for the input field is what makes the text uppercase when you press Tab to navigate away from that field. Because this is a one-input-field form, the Enter key acts to submit the form, so we've set the form's `onsubmit` event to run the `upperMe()` function as well. In addition to making the text uppercase, that function also cancels the normal submit behavior. (More on event capture later in this chapter.)

If JavaScript is disabled or not supported by the user agent rendering this page, the entered text is submitted to a server-side program called `make-uppercase.php`, which presumably performs the uppercasing of the text as well.

Note

The function to assign event handlers throughout the code in this chapter and much of the book is `addEvent()`**, a cross-browser event handler explained in detail in Chapter 32, "Event Objects."**

The `addEvent()` **function is located in the file** `jsb-global.js` **on the accompanying CD-ROM and is included in each HTML example with a** `script` **tag in the document** `head`**.** ■

The button input object

We use the button-type `input` element in many examples in this book. The button is one of the simplest objects to script. In the simplified object model of this tutorial, the button object has only a few properties that are rarely accessed or modified in day-to-day scripts. Like the text object, the visual aspects of the button are governed not by HTML or scripts but by the operating system and browser that the page visitor uses. By far the most useful event of the button object is the `click` event. It fires whenever the user clicks the button. Simple enough. No magic here.

The checkbox input object

A checkbox is also a simple element of the `form` object, but some of the properties may not be entirely intuitive. Unlike the `value` property of a plain button object (the text of the button label), the `value` property of a checkbox is any other text you want associated with the object. This text does not appear on the page in any fashion, but the property (initially set via the `value` attribute) might be important to a script that wants to know more about the purpose of the checkbox within the form.

For example, consider this:

```
<input type="checkbox" id="memory" name="remember-me" value="yup">
<label for="memory">Remember me on this computer</label>
```

If we check this checkbox and submit the form, the browser sends the server the name/value pair “remember-me” and “yup”; the label text “Remember me on this computer” appears on the screen but is not sent to the server.

The key property of a checkbox object is whether the box is checked. The `checked` property is a Boolean value: `true` if the box is checked, `false` if not. When you see that a property is a Boolean value, it’s a clue that the value will be easy to use in an `if` or `if...else` conditional expression. In Listing 11-3, the value of the `checked` property determines which alert box the user sees.

LISTING 11-3

The Checkbox Object’s checked Property

HTML: jsb-11-03.html

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Checkbox Inspector</title>
      <script type="text/javascript" src="../jsb-global.js"></script>
      <script type="text/javascript" src="jsb-11-03.js"></script>
   </head>
   <body>
      <h1>Checkbox Inspector</h1>
      <form action="">
         <p>
            <input type="checkbox" id="checkThis" name="checkThis">
```

continued

LISTING 11-3 *(continued)*

```
            <label for="checkThis">Check here</label>
            <input type="button" id="inspectIt" value="Inspect Box">
         </p>
      </form>
   </body>
</html>
```

JavaScript: jsb-11-03.js

```
// initialize when the page has loaded
addEvent(window, 'load', initialize);

var oCheckbox;   // checkbox object (global)

// apply behaviors when document has loaded
function initialize()
{
   // do this only if the browser can handle DOM methods
   if (document.getElementById)
   {
      // point to crucial elements
      oCheckbox = document.getElementById('checkThis');
      var oButton = document.getElementById('inspectIt');

         // if they exist, apply event handler
         if (oCheckbox && oButton)
         {
            addEvent(oButton, 'click', inspectBox);
         }
   }
}

// report the checked state of the checkbox
function inspectBox()
{
   if (oCheckbox.checked)
   {
      alert("The box is checked.");
   }
   else
   {
      alert("The box is not checked at the moment.");
   }
}
```

Checkboxes are generally used as preference setters rather than as action inducers. Although a checkbox object has an `onclick` event handler, a click of a checkbox should never do anything drastic, such as navigate to another page.

The radio input object

Setting up a group of radio objects for scripting requires a bit more work. To let the browser manage the highlighting and unhighlighting of a related group of buttons, you must assign the same `name` attribute to each of the buttons in the group. You can have multiple radio groups within a form, but each member of the same group must have the same name.

Assigning the same name to a form element forces the browser to manage the elements differently than if they each had a unique name. Instead, the browser maintains an array list of objects with the same name. The name assigned to the group becomes the name of the array. Some properties apply to the group as a whole; other properties apply to individual buttons within the group and must be addressed via array index references. For example, you can find out how many buttons are in a radio group by reading the `length` property of the group:

```
formObject.groupName.length
```

If you want to find out whether a particular button is currently highlighted via the same `checked` property used for the checkbox, you can access the button element by its position in the collection of same-named input fields:

```
formObject.groupName[0].checked
```

Listing 11-4 demonstrates several aspects of the radio-button object, including how to look through a group of buttons to find out which one is checked, and how to use the `value` attribute and corresponding property for meaningful work.

The page includes three radio buttons and a plain button. Each radio button's `value` attribute contains the full name of one of the Three Stooges. When the user clicks the button, the `onclick` event handler invokes the `fullName()` function. In that function, the first statement creates a shortcut reference to the form. Next, a `for` repeat loop looks through all the buttons in the `stooges` radio-button group. An `if` construction looks at the `checked` property of each button. When a button is checked, the `break` statement bails out of the `for` loop, leaving the value of the `i` loop counter at the number where the loop broke ranks. Then the alert dialog box uses a reference to the `value` property of the `i`th button so that the full name can be displayed in the alert.

LISTING 11-4

Scripting a Group of Radio Objects

HTML: jsb-11-04.html

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Extracting Highlighted Radio Button</title>
      <script type="text/javascript" src="../jsb-global.js"></script>
      <script type="text/javascript" src="jsb-11-04.js"></script>
   </head>
   <body>
      <h1>Extracting Highlighted Radio Button</h1>
      <form action="stooges.php">
```

continued

LISTING 11-4 *(continued)*

```
         <fieldset>
            <legend>Select your favorite Stooge:</legend>
            <p>
               <input type="radio" name="stooges" id="stooges-1"
               value="Moe Howard" checked>
               <label for="stooges-1">Moe</label>
            </p>
            <p>
               <input type="radio" name="stooges" id="stooges-2"
               value="Larry Fine">
               <label for="stooges-2">Larry</label>
            </p>
            <p>
               <input type="radio" name="stooges" id="stooges-3"
               value="Curly Howard">
               <label for="stooges-3">Curly</label>
            </p>
            <p>
               <input type="submit" id="Viewer" name="Viewer"
               value="View Full Name...">
            </p>
         </fieldset>
      </form>
   </body>
</html>
```

JavaScript: jsb-11-04.js

```
// initialize when the page has loaded
addEvent(window, 'load', initialize);

var aStooges;   // radio button array (global)

// apply behaviors when document has loaded
function initialize()
{
   // do this only if the browser can handle DOM methods
   if (document.getElementById)
   {
      // point to crucial elements
      var oButton = document.getElementById('Viewer');
      var aForms = document.forms;
         if (aForms) aStooges = aForms[0].stooges;   // global variable

         // if they exist, apply event handler
         if (oButton && aStooges)
         {
            addEvent(oButton, 'click', showFullName);
         }
```

```
    }
}

// display the full name of the selected stooge
function showFullName()
{
    for (var i = 0; i < aStooges.length; i++)
    {
        if (aStooges[i].checked)
        {
            break;
        }
    }
    alert("You chose " + aStooges[i].value + ".");
}
```

The select object

The most complex form control to script is the `select` element object. As you can see from the DOM Level 0 form object hierarchy diagram (see Figure 11-1), the select object is really a compound object that contains an array of option objects. Moreover, you can establish this object in HTML to display itself as either a drop-down list or a scrolling list — the latter configurable to accept multiple selections by users. For the sake of simplicity at this stage, this lesson focuses on deploying it as a drop-down list that allows only single selections.

Some properties belong to the entire select object; others belong to individual options inside the select object. If your goal is to determine which item the user selects, and you want the code to work on the widest range of browsers, you must use properties of both the select and option objects.

The most important property of the select object itself is the `selectedIndex` property, accessed as follows:

```
formObject.selectName.selectedIndex
```

This value is the index number of the currently selected item. As with most index counting schemes in JavaScript, the first item (the one at the top of the list) has an index of zero. The `selectedIndex` value is critical for enabling you to access properties of the selected option. Two important properties of an option item are `text` and `value`, accessed as follows:

```
formObject.selectName.options[n].text
formObject.selectName.options[n].value
```

The `text` property is the string that appears onscreen in the select object's list. It is unusual for this information to be exposed as a form object property because the HTML that generates a select object defines the text as an `<option>` tag's nested text. But inside the `<option>` tag, you can set a `value` attribute, which, like the radio buttons shown earlier, enables you to associate some hidden string information with each visible entry in the list.

To read the `value` or `text` property of a selected option most efficiently for all browsers, you can use the select object's `selectedIndex` property as an index value to the option. References for this kind of operation get pretty long, so take the time to understand what's happening here. In the following function, the first statement creates a shortcut reference to the select object. Then the

selectedIndex property of the select object is substituted for the index value of the options array of that same object:

```
function inspect()
{
   var oList = document.getElementById('choices');
      if (oList)
      {
         var sChosenItemValue = oList.options[oList.selectedIndex].value;
      }
}
```

To bring a select object to life, use the onchange event handler. As soon as a user makes a new selection in the list, the script associated with the onchange event is run. Listing 11-5 shows a common application for a select object. Its text entries describe places to go in and out of a web site, and the value attributes hold the URLs for those locations. When a user makes a selection in the list, the onchange event handler triggers a script that extracts the value property of the selected option and assigns that value to the location.href object property to effect the navigation.

Of course, most of these new locations (web pages) won't already exist on your computer in the folder where the HTML page resides. When you select a location and your page attempts to relocate there, you'll see a "File not found" error page. Notice the intended page in the address bar, and then click the Back button to return to the sample form.

The form includes a submit button so that, when JavaScript is turned off or is non-existent in the user agent, the page still works by submitting the form to a server-side program to perform the redirection. Under JavaScript control, it can be argued that this kind of navigation doesn't need a separate Go button on the page, and in this example we've hidden the button from view using JavaScript to set its display style to "none." Note, however, that some people criticize the usability and accessibility of a select list that submits a form as soon as a selection is made. The argument is that since form submission isn't a normal function of a select list, adding that behavior with JavaScript will take some users by surprise and will switch them to a new page without their deliberate consent. Also, the list's behavior assumes that the user's first selection from the list is where they really want to go and doesn't allow for accidental selections, which anyone could make, but particularly those with motor-skill disabilities. Try this form with JavaScript turned off to see the difference for yourself.

LISTING 11-5

Navigating with a select Object

HTML: jsb-11-05.html

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Select Navigation</title>
      <script type="text/javascript" src="../jsb-global.js"></script>
      <script type="text/javascript" src="jsb-11-05.js"></script>
   </head>
   <body>
```

```
      <h1>Select Navigation</h1>
      <form action="redirect.php" method="get">
         <p>
            <label for="urlList">Choose a place to go:</label>
            <select id="urlList" name="urlList">
               <option selected value="index.html">Home Page</option>
               <option value="store.html">Shop Our Store</option>
               <option value="policies.html">Shipping Policies</option>
               <option value="http://www.google.com">Search the Web</option>
            </select>

            <input type="submit" id="submit-button" value="Go">
         </p>
      </form>
   </body>
</html>
```

JavaScript: jsb-11-05.js

```
// initialize when the page has loaded
addEvent(window, 'load', initialize);

var oList;   // select list (global)

// apply behaviors when document has loaded
function initialize()
{
   // do this only if the browser can handle DOM methods
   if (document.getElementById)
   {
      // point to crucial elements
      oList = document.getElementById('urlList');
      oButton = document.getElementById('submit-button');

         // if they exist...
         if (oList && oButton)
         {
            // make the list dynamic
            addEvent(oList, 'change', goThere);

            // make the submit button disappear when JavaScript is running
            oButton.style.display = 'none';
         }
   }
}

// direct the browser to the selected URL
function goThere()
{
   location.href = oList.options[oList.selectedIndex].value;
}
```

Note
Recent browsers also expose the `value` property of the selected option item by way of the `value` property of the `select` object. This is certainly a logical and convenient shortcut, and you can use it safely if your target browsers include IE, Mozilla-based browsers, and Safari. ■

There is much more to the select object, including the ability to change the contents of a list in newer browsers. Chapter 37 covers the select object in depth.

Passing Elements to Functions with this

In all the examples so far in this lesson, when an event handler invokes a function that works with form elements, the form or form control is explicitly referenced by a global variable. But valuable shortcuts exist for transferring information about the form or form control directly to the function without having to declare global variables.

JavaScript features a keyword — `this` — that always refers to whatever object contains the script in which the keyword is used. For instance, if you attach a function to a button by its `click` event, then you click the button, the keyword `this` inside the function refers to the button itself. Using `this` is cool: it means having to use fewer global variables, lessening the chance that two independent scripts will someday use the same global variable name and mess up each other's logic, and it means that you can use the same generic function for multiple elements. Here's a simple example:

```
function identify()
{
   if (this.tagName)
   {
      alert('My tagName is ' + this.tagName);
   }
}
```

If you apply the `identify()` function to multiple elements on the page and then click them, each reports its `tagName`: `P`, `LABEL`, `INPUT`, `FORM`, `BODY`, etc.

Take a look at Listing 11-6. There are two functions that use the `this` keyword: `processData()`, which is attached to the submit button, and `verifySong()`, which is attached to the song name input field.

LISTING 11-6

Passing Elements to Functions with this

HTML: jsb-11-06.html

```
<!DOCTYPE html>
<html>
   <head>
     <meta http-equiv="content-type" content="text/html;charset=utf-8">
     <title>Beatle Picker</title>
     <script type="text/javascript" src="../jsb-global.js"></script>
```

```
    <script type="text/javascript" src="jsb-11-06.js"></script>
  </head>
  <body>
    <h1>Beatle Picker</h1>
    <form id="beatles-form" action="beatles.php" method="get">
      <p>
        <label>Choose your favorite Beatle:</label>
        <input type="radio" name="Beatles" id="radio1"
                     value="John Lennon" checked>
        <label for="radio1">John</label>
        <input type="radio" name="Beatles" id="radio2" value="Paul McCartney">
        <label for="radio2">Paul</label>
        <input type="radio" name="Beatles" id="radio3" value="George Harrison">
        <label for="radio3">George</label>
        <input type="radio" name="Beatles" id="radio4" value="Ringo Starr">
        <label for="radio4">Ringo</label>
      </p>
      <p>
        <label for="song">Enter the name of your favorite Beatles song:</label>
        <input type="text" id="song" name="song" value="Eleanor Rigby">
        <input type="submit" id="submit" value="Process Request...">
      </p>
    </form>
  </body>
</html>
```

JavaScript: jsb-11-06.js

```
// initialize when the page has loaded
addEvent(window, 'load', initialize);

// apply behaviors when document has loaded
function initialize()
{
   // do this only if the browser can handle DOM methods
   if (document.getElementById)
   {
      // point to crucial elements
      var oForm = document.getElementById('beatles-form');
      var oSong = document.getElementById('song');
      var oButtonSubmit = document.getElementById('submit');

         // if they all exist...
         if (oForm && oSong && oButtonSubmit)
         {
            // apply behavior to input field & button
            addEvent(oSong, 'change', verifySong);
            addEvent(oButtonSubmit, 'click', processData);
         }
   }
}
```

continued

LISTING 11-6 *(continued)*

```
// verify all input & suppress form submission
function processData(evt)
{
      // consolidate event handling
      if (!evt) evt = window.event;

   // point to the activated control's form ancestor
   var oForm = this.form;

   // see which radio button was selected
   for (var i = 0; i < oForm.Beatles.length; i++)
   {
      if (oForm.Beatles[i].checked)
      {
         break;
      }
   }

   // assign values to variables for convenience
   var sBeatle = oForm.Beatles[i].value;
   var sSong = oForm.song.value;

   // this is where a data lookup would go...
   alert("Checking whether " + sSong + " features " + sBeatle + "...");

      // cancel form submission
      if (evt.preventDefault) evt.preventDefault();
   return false;
}

// verify the song name when it's changed
function verifySong()
{
   // get the input song
   // ('this' is the object whose event handler called this function)
   var sSong = this.value;

   // this is where a data lookup would go...
   alert("Checking whether " + sSong + " is a Beatles tune...");
}
```

The function `verifySong()` wants access to the value entered into the song input field, so it uses `this` in the statement:

```
var sSong = this.value;
```

In other words, get the value of the current object (the input field) and assign it to the variable `sSong`. We're using `this` as a shortcut replacement for

```
var oInput = document.getElementById('song');
var sSong = oInput.value;
```

The function `processData()` needs to refer to several form elements, so it first takes advantage of the fact that every form element points to the form it belongs to with the `form` property:

```
var oForm = this.form;
```

Regardless of which input control `this` refers to, `this.form` points to the parent form. Then it can proceed to use object references such as `oForm.Beatles` and `oForm.song` to point to various controls within the form by name.

If you're a bit puzzled by the behavior of this script when you run it in your browser, here's an explanation of the programming logic behind what you experience. When you enter a new song title in the text box and exit the input field by pressing Tab or clicking elsewhere, the `onchange` event handler calls the `verifySong()` function, which displays an alert saying that it's checking the song. (It doesn't actually validate it against a database in this example.)

When you click the Process Request button, its `onclick` event handler calls the function `processData()`, which announces that it's checking to see if the entered song features the Beatle selected in the radio buttons above.

Now let's throw it two balls at once. Type a new song name in the input field, and then immediately click the Process Request button. You see only one alert — the `verifySong()` message that it's checking the song. Why don't you see both alerts? Because the button `click` action is interrupted by the `onchange` event handler of the text box. In other words, the button doesn't really get clicked, because the `onchange` alert dialog box comes up first. That's why you have to click the button for what seems to be a second time to get the song/Beatle verification. If you don't change the text in the field, your click of the button occurs without interruption, and the `processData()` verification takes place.

Note

Discrepancies between the ways that IE and other browsers handle event assignments and event processing require explanations beyond the scope of this tutorial. You'll meet them soon enough, however, beginning in Chapter 25, "Document Object Model Essentials," and again in Chapter 32, "Event Objects." ■

Submitting and Prevalidating Forms

In an ordinary HTML document, we submit a form by clicking on a `submit` button; the browser then gathers up the values we entered or selected with the input controls and sends them to the specified URI. JavaScript lets us intercept that submit request, validate the input or do whatever else we need to do, and then either permit the form submission to proceed or cancel it so that we can guide the user to improve the input.

You can perform this sort of last-second validation of data or other scripting (for example, changing the form's `action` property based on user choices) in a function invoked by the form's `onsubmit` event handler. We'll go into substantial detail about validation routines in Chapter 46, "Data-Entry Validation," on the CD-ROM. For now, we just want to show you how the `onsubmit` event handler works.

To juggle the ways different browsers handle events, we need to use two techniques to cancel the event, corresponding exactly to the two techniques our `addEvent()` function uses to apply the event handlers in the first place:

```
// add an event to an element
function addEvent(elem, evtType, func)
{
   // first try the W3C DOM method
   if (elem.addEventListener)
   {
      elem.addEventListener(evtType, func, false);
   }
   // otherwise use the 'traditional' technique
   else
   {
      elem["on" + evtType] = func;
   }
}
```

Compare this to how the script in Listing 11-7 cancels the submit event:

```
function checkForm(evt)
{
      // consolidate event handling
      if (!evt) evt = window.event;
   ...
   // cancel form submission
      // W3C DOM method (hide from IE)
      if (evt.preventDefault) evt.preventDefault();
   // IE method
   return false;
   ...
}
```

For browsers compatible with the W3C DOM, the event is added with `addEventListener()`, and the form submission is cancelled with `preventDefault()`. Attempting to invoke a DOM method that the browser thinks doesn't exist causes a runtime error, so we protect the older and less capable browsers by testing first, to make sure the method exists for the event object.

For those less capable browsers, notably Internet Explorer, the event is added to the object by setting the event property (in this case, `onsubmit`), and form submission is cancelled simply by returning a `false` value from the function called by the event handler.

We'll get into all the whys and wherefores later in this book, but for now let's use this dual technique for handling events.

Listing 11-7 shows a page with a simple validation routine that ensures that all fields have something in them before allowing form submission to take place. The HTML marks up a form with four input fields and a submit button. Note that the `action` of the `form` element is the fictitious server-side program `validate.php`. If this page is delivered to a user agent not running JavaScript, the form will submit the input values to the server-side program for validation. Fundamental form validation

must always happen server-side; we're duplicating that validation in client-side JavaScript to give the user a more immediate response to their actions. (Incidentally, that's one indicator of whether there's a typo in the script: if you submit this form with any of the input fields blank and the browser window attempts to bring up `validate.php`, you'll know that you probably made a typo while entering the HTML or JavaScript code because the script isn't blocking form submission.)

After the page loads, the script attaches the `checkForm()` function to the `onsubmit` event of the form. Note that we're not applying `onclick` behavior to the `submit` button; instead, we're letting the button do its normal job and we're intercepting the submission process at the form object itself.

When `submit` is clicked, the next normal step is for the browser to submit the form, but first it honors the `onsubmit` event handler and calls our function `checkForm()`. This function's principal job is to loop through all the form controls, looking for a blank input field. If it finds one, it displays an error message and cancels form submission.

Inside the loop, the `if` statement performs two tests. The first test is to make sure that we're examining form controls whose `type` properties are text (so as not to bother with, say, buttons). Next, it checks to see whether the value of the text field is empty. The `&&` operator (called a Boolean `AND` operator) forces both sides to evaluate to `true` before the entire condition expression inside the parentheses evaluates to `true`. If either subtest fails, the whole condition fails.

LISTING 11-7

Last-Minute Checking Before Form Submission

HTML: jsb-11-07.html

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Form Field Validator</title>
      <script type="text/javascript" src="../jsb-global.js"></script>
      <script type="text/javascript" src="jsb-11-07.js"></script>
   </head>
   <body>
      <h1>Form Field Validator</h1>
      <form id="theForm" action="validate.php" method="get">
         <p>Please enter all requested information:</p>
         <p>
            <label for="firstName">First Name:</label>
            <input type="text" id="firstName" name="firstName">
         </p>
         <p>
            <label for="lastName">Last Name:</label>
            <input type="text" id="lastName" name="lastName">
         </p>
         <p>
            <label for="age">Age:</label>
            <input type="text" id="age" name="age">
         </p>
```

continued

LISTING 11-7 *(continued)*

```
         <p>
            <label for="favoriteColor">Favorite Color:</label>
            <input type="text" id="favoriteColor" name="favoriteColor">
         </p>
         <p>
            <input type="submit" id="submit">
         </p>
      </form>
   </body>
</html>
```

JavaScript: jsb-11-07.js

```
// initialize when the page has loaded
addEvent(window, 'load', initialize);

// apply behaviors when document has loaded
function initialize()
{
   // do this only if the browser can handle DOM methods
   if (document.getElementById)
   {
      // point to the form
      var oForm = document.getElementById('theForm');

         // if it exists, apply the behavior
         if (oForm)
         {
            addEvent(oForm, 'submit', checkForm);
         }
   }
}

// when the form is submitted,
// check to make sure all input fields have been filled in
function checkForm(evt)
{
      // consolidate event handling
      if (!evt) evt = window.event;

   // 'this' is the current object, in this case the form
   for (var i = 0; i < this.elements.length; i++)
   {
      if (this.elements[i].type == "text" && this.elements[i].value == "")
      {
         alert("Fill out ALL fields.");

         // cancel form submission
            // W3C DOM method (hide from IE)
```

```
            if (evt.preventDefault) evt.preventDefault();
         // IE method
         return false;
      }
   }
   // allow form submission
   return true;
}
```

A word about submit()

The scripted equivalent of submitting a form is the form object's `submit()` method. All you need in the statement is a reference to the form and this method:

```
formObject.submit();
```

One quirky bit of behavior involving the `submit()` method and the `onsubmit` event handler needs explanation. Although you might think (and logically so, in our opinion) that the `submit()` method would be the exact scripted equivalent of the click of the real Submit button, it's not. The `submit()` method does not cause the form's `submit` event to fire at all. If you want to perform validation on a form submitted via the `submit()` method, invoke the validation in the script function that ultimately calls the `submit()` method.

So much for the basics of forms and form controls. In Chapter 12, you step away from HTML for a moment to look at more advanced JavaScript core language items: strings, math, and dates.

Exercises

1. Rework Listings 11-2, 11-3, 11-4, and 11-5 to use the `this` keyword instead of global variables.
2. For the following form (assume that it's the only form on the page), write at least 10 ways to reference the text input field as an object in all modern scriptable browsers.

```
<form name="subscription" action="cgi-bin/maillist.pl" method="post">
 <p>
   <input type="text" id="email" name="email">
   <input type="submit">
 </p>
</form>
```

3. Write a function that displays the value of an input field in an alert dialog box, plus the script that causes that function to be called when the input field is changed.
4. A document contains two forms, `specifications` and `accessories`. In the `accessories` form is a field named `acc1`. Write at least two different statements that set the contents of that field to `Leather Carrying Case`.
5. Create a page that includes a select object to change the background color of the current page. The property that you need to set is `document.body.style.backgroundColor`, and the three values you should offer as options are `red`, `yellow`, and `green`. In the `select` list, the colors should display as `Stop`, `Caution`, and `Go`.

CHAPTER 12

Strings, Math, and Dates

For most of the lessons in the tutorial so far, the objects at the center of attention belong to the document object model (DOM). But, as indicated in Chapter 2, "Developing a Scripting Strategy," a clear dividing line exists between the DOM and the JavaScript language. The language has some of its own objects that are independent of the DOM. These objects are defined such that if a vendor wished to implement JavaScript as the programming language for an entirely different kind of product, the language would still use these core facilities for handling text, advanced math (beyond simple arithmetic), and dates. You can find formal specifications of these objects in the ECMA-262 recommendation.

IN THIS CHAPTER

How to modify strings with common string methods

When and how to use the Math object

How to use the Date object

Core Language Objects

It is often difficult for newcomers to programming — or even experienced programmers who have not worked with object-oriented languages before — to think about objects, especially when objects are attributed to things that don't seem to have a physical presence. For example, it doesn't require lengthy study to grasp the notion that a button on a page is an object. It has several physical properties that make perfect sense. But what about a string of characters? As you learn in this chapter, in an object-based environment such as JavaScript, "everything that moves" is treated as an object — each piece of data from a Boolean value to a date. Each such object probably has one or more properties that help define the content; such an object may also have methods associated with it to define what the object can do or what you can do to the object.

JavaScript objects that are not part of the DOM are called *core language objects*. You can see the full complement of them in the Quick Reference in Appendix A. This chapter focuses on the `String`, `Math`, and `Date` objects.

String Objects

You have used `String` objects many times in earlier lessons. A *string* is any text inside a quote pair. A quote pair consists of either double quotes (`" "`) or single quotes (`' '`). That JavaScript includes two types of quotes makes it easy to nest one string inside another. In the following example, the `alert()` method requires a quoted string as a parameter:

```
alert('You cannot lose.');
```

If the quoted expression includes an apostrophe, it's easy enough to switch the outer quotes to double:

```
alert("You can't lose.");
```

When the solution isn't so simple, as when both apostrophe and quotation mark appear in the quoted string, *escape sequences* come to the rescue. More on them in Chapter 15, "The String Object."

JavaScript imposes no practical limit on the number of characters that a string can hold. However, most older browsers have a limit of 255 characters for a script statement. This limit is sometimes exceeded when a script includes a lengthy string that is to become scripted content in a page. You need to divide such lines into smaller chunks, using techniques described in a moment.

You have two ways to assign a string value to a variable. The simplest is a basic assignment statement:

```
var myString = "Howdy";
```

This works perfectly well except in some exceedingly rare instances. You can also create a string object using the more formal syntax that involves the `new` keyword and a constructor function (that is, it constructs a new object):

```
var myString = new String("Howdy");
```

Whichever way you initialize a variable with a string, the variable receiving the assignment can respond to all `String` object methods.

Joining strings

Bringing two strings together as a single string is called *concatenation*, a term you learned in Chapter 8, "Programming Fundamentals, Part I." String concatenation requires one of two JavaScript operators. Even in your first close look at script, in Chapter 3, "Selecting and Using Your Tools," you saw how the addition operator (+) combines multiple strings into one:

```
var today = new Date();
var msg = "This is JavaScript saying it's now " + today.
    toLocaleString();
```

As valuable as the + operator is, another related operator can be even more scripter-friendly: +=. This operator is helpful when you are assembling large strings in a single variable. The strings may be so long or cumbersome that you need to divide the building process into multiple statements. Or you might decide to split the concatenation into several statements to help it make more sense to the human reader. The pieces may be combinations of *string literals* (strings inside quotes) and variable values. The clumsy way to do it (perfectly doable in JavaScript) is to use the addition operator to append more text to the existing chunk:

```
var msg = "Four score";
msg = msg + " and seven";
msg = msg + " years ago,";
```

But the += operator, called the *add-by-value operator*, offers a handy shortcut. The symbol for the operator is a plus and equal sign together. This operator means *append the stuff on the right of me to the end of the stuff on the left of me*. Therefore, the preceding sequence is shortened as follows:

```
var msg = "Four score";
msg += " and seven";
msg += " years ago,";
```

You can also combine the operators if the need arises:

```
var msg = "Four score";
msg += " and seven" + " years ago";
```

String methods

Of all the core JavaScript objects, the `String` object has the most diverse collection of methods associated with it. Many methods are designed to help scripts extract segments of a string. Another group, rarely used and now obsolete in favor of Cascading Style Sheets (CSS), wraps a string with one of several style-oriented tags (a scripted equivalent of tags for font size, font style, and the like).

In a string method, the string being acted upon becomes part of the reference followed by the method name:

```
myString.methodName();
```

All methods return a value of some kind. Most of the time, the returned value is a converted version of the string object referred to in the method call — but the original string is still intact. To capture the modified version, you need to assign the results of the method to a variable:

```
var result = myString.methodName();
```

The following sections introduce you to several important string methods available to all browser brands and versions.

Changing string case

Two methods convert a string to all uppercase or all lowercase letters:

```
var result = string.toUpperCase();
var result = string.toLowerCase();
```

As always, you must strictly observe the case of each letter of the method names if you want them to work. These methods come in handy when your scripts need to compare strings that may not have the same case (for example, a string in a lookup table compared with a string typed by a user). Because the methods don't change the original strings attached to the expressions, you can simply compare the evaluated results of the methods:

```
var foundMatch = false;
   if (stringA.toUpperCase() == stringB.toUpperCase())
   {
      foundMatch = true;
   }
```

String searches

You can use the `string.indexOf()` method to determine whether one string is contained by another. For example, say you've built a contact form on a web site that sends visitors' messages to the public relations department, but you want to carbon copy the ombudsman any messages containing the word "complaint." That word could occur anywhere within the long message string. In this case, you probably don't need to know exactly where the word occurs, just whether it's there.

The `string.indexOf()` method returns a number indicating the index value (zero-based) of the character in the larger string where the smaller string begins. The key point about this method is that, if no match occurs, the returned value is `-1`. To find out whether the smaller string is inside, all you need to test is whether the returned value is something other than `-1`.

Two strings are involved with this method: the shorter one and the longer one. The longer string is the one that appears in the reference to the left of the method name; the shorter string is inserted as a parameter to the `indexOf()` method. To demonstrate the method in action, the following fragment looks to see whether the input message contains the word 'complaint':

```
var sMsg = '';
var oInput = document.getElementById('message');
   if (oInput)
   {
      sMsg = oInput.value;
   }
var isComplaint = false;
   if (sMsg.indexOf("complaint") != -1)
   {
      isComplaint = true;
   }
```

The operator in the `if` construction's condition (`!=`) is the inequality operator. You can read it as meaning *is not equal to.*

Extracting copies of characters and substrings

To extract a single character at a known position within a string, use the `charAt()` method. The parameter of the method is an index number (zero-based) of the character to extract. When we say *extract,* we don't mean delete, but grab a snapshot of the character. The original string is not modified in any way.

For example, consider a script in a main window that is capable of inspecting a variable, `stringA`, in another window that displays map images of different corporate buildings. When the window has a map of Building C in it, the `stringA` variable contains `"Building C"`. In English the building letter is always at the 10th character position of the string (or number 9 in a zero-based counting world), so the script can examine that one character to identify the map currently in that other window:

```
var stringA = "Building C";
var bldgLetter = stringA.charAt(9);
   // result: bldgLetter = "C"
```

There are two similar methods — `string.substr()` and `string.substring()` — that enable you to extract a contiguous sequence of characters, provided that you know the starting position of the substring you want to grab. The difference between the two methods is that `string.substr()`

wants to know the length of the substring, whereas `string.substring()` wants to know the ending position of the substring:

```
string.substr(startingPosition [, length])
string.substring(startingPosition [, endingPosition])
```

Either method will extract everything up to the end of the original string if the second parameter (length or ending position) is omitted. The original string from which the extraction is made appears to the left of the method name in the reference. The starting and ending position parameters are index values (zero-based).

```
var stringA = "banana daiquiri";

var excerpt = stringA.substr(2,4);     // result: "nana"
var excerpt = stringA.substr(2);       // result: "nana daiquiri"

var excerpt = stringA.substring(2,6);  // result: "nana"
var excerpt = stringA.substring(2);    // result: "nana daiquiri"
```

String manipulation in JavaScript is fairly cumbersome compared with that in some other scripting languages. Higher-level notions of words, sentences, or paragraphs are absent. Therefore, sometimes it takes a bit of scripting with string methods to accomplish what seems like a simple goal. Yet you can put your knowledge of expression evaluation to the test as you assemble expressions that utilize heavily nested constructions.

For example, let's say you want to grab everything from a multiword expression except the first word:

```
var stringA = "The Painted Bird";
var firstSpace = stringA.indexOf(" ");
var excerpt = stringA.substring(firstSpace + 1);
   // result: excerpt = "Painted Bird"
```

Assuming that the first word can be of any length, the second statement uses the `string.indexOf()` method to look for the first space character. We add 1 to that value to serve as the starting index value for the `string.substring()` method. We omit the second `length` parameter in order to extract everything up to the end of `stringA`.

Creating statements like this one is not something you are likely to enjoy over and over again, so in Chapter 23, "Function Objects and Custom Objects," we show you how to create your own library of string functions that you can reuse in all of your scripts that need their string-handling facilities. More powerful string-matching facilities are also built into today's browsers by way of *regular expressions* (see Chapter 15, "The String Object," and Chapter 45, "The Regular Expression and RegExp Objects").

The Math Object

JavaScript provides ample facilities for math — far more than most scripters who don't have a background in computer science and math will likely use. But every genuine programming language needs these powers to accommodate clever programmers who can make windows fly in circles onscreen.

The `Math` object contains all these powers. This object is unlike most of the other objects in JavaScript in that you don't generate copies of the object to use. Instead, your scripts use the properties and

methods of a single `Math` object. (Technically, one `Math` object actually occurs per window or frame, but this fact has no impact whatsoever on your scripts.) Programmers call this kind of fixed object a *static object*. That `Math` object (with an uppercase *M*) is part of the reference to the property or method. Properties of the `Math` object are constant values, such as pi and the square root of 2:

```
var piValue = Math.PI;
var rootOfTwo = Math.SQRT2;
```

`Math` object methods cover a wide range of trigonometric functions and other math functions that work on numeric values already defined in your script. For example, you can find which of two numbers is the larger:

```
var larger = Math.max(value1, value2);
```

Or you can raise one number to a power of 10:

```
var result = Math.pow(value1, 10);
```

More common, perhaps, is the method that rounds a value to the nearest integer value:

```
var result = Math.round(value1);
```

Another common request of the `Math` object is a random number. The `Math.random()` method returns a floating-point number between 0 and 1. If you design a script to act like a card game, you need random integers between 1 and 52; for dice, the range is 1 to 6 per die. To generate a random integer between 0 and any top value, use the following formula

```
Math.floor(Math.random() * (n + 1))
```

where `n` is the top number. (`Math.floor` returns the integer part — the digits to the left of the decimal point — of any floating-point number.) To generate random numbers between 1 and any higher number, use this formula

```
Math.floor(Math.random() * n) + 1
```

where `n` equals the top number of the range. For the dice game, the formula for each die is

```
newDieValue = Math.floor(Math.random() * 6) + 1;
```

To see this, enter the right part of the preceding statement in the top text box of The Evaluator Jr. and repeatedly click the Evaluate button.

The Date Object

Working with dates beyond simple tasks can be difficult business in JavaScript. A lot of the difficulty comes from the fact that dates and times are calculated internally according to *Greenwich Mean Time (GMT)* — provided that the visitor's own internal PC clock and control panel are set accurately. As a result of this complexity, better left for Chapter 17, "The Date Object," this section of the tutorial touches on only the basics of the JavaScript `Date` object.

A scriptable browser contains one global `Date` object (in truth, one `Date` object per window) that is always present, ready to be called upon at any moment. The `Date` object is another one of those static objects. When you wish to work with a date, such as displaying today's date, you need to invoke the `Date` object constructor function to obtain an instance of a `Date` object tied to a specific time and date. For example, when you invoke the constructor without any parameters, as in

```
var today = new Date();
```

the `Date` object takes a snapshot of the PC's internal clock and returns a Date object for that instant. Notice the distinction between the static `Date` object and a `Date` object instance, which contains an actual date value. The variable `today` contains not a ticking clock but a value that you can examine, tear apart, and reassemble as needed for your script.

Internally, the value of a `Date` object instance is the time, in milliseconds, from zero o'clock on January 1, 1970, in the GMT zone — the world standard reference point for all time conversions. That's how a `Date` object contains both date and time information.

You can also grab a snapshot of the `Date` object for a particular date and time in the past or future by specifying that information as parameters to the `Date` object constructor function:

```
var someDate = new Date("Month dd, yyyy hh:mm:ss");
var someDate = new Date("Month dd, yyyy");
var someDate = new Date(yyyy,mm,dd,hh,mm,ss);
var someDate = new Date(yyyy,mm,dd);
var someDate = new Date(GMT milliseconds from 1/1/1970);
```

If you attempt to view the contents of a raw `Date` object, JavaScript converts the value to the local time-zone string, as indicated by your PC's control panel setting. To see this in action, use The Evaluator Jr.'s top text box to enter the following:

```
new Date();
```

Your PC's clock supplies the current date and time as the clock calculates them (even though JavaScript still stores the date object's millisecond count in the GMT zone). You can, however, extract components of the `Date` object via a series of methods that you apply to a `Date` object instance. Table 12-1 shows an abbreviated listing of these properties and information about their values.

Caution

Be careful about values whose ranges start with zero, especially the months. The `getMonth()` and `setMonth()` method values are zero based, so the numbers are 1 less than the month numbers you are accustomed to working with (for example, January is 0 and December is 11). ■

You may notice one difference about the methods that set values of a `Date` object. Rather than returning some new value, these methods actually modify the value of the instance of the `Date` object referenced in the call to the method.

TABLE 12-1

Some Date Object Methods

Method	Value Range	Description
dateObj.getTime()	0-...	Milliseconds since 1/1/70 00:00:00 GMT
dateObj.getYear()	70-...	Specified year minus 1900; four-digit year for 2000+
dateObj.getFullYear()	1970-...	Four-digit year (Y2K-compliant); version 4+ browsers
dateObj.getMonth()	0-11	Month within the year (January = 0)
dateObj.getDate()	1-31	Date within the month
dateObj.getDay()	0-6	Day of week (Sunday = 0)
dateObj.getHours()	0-23	Hour of the day in 24-hour time
dateObj.getMinutes()	0-59	Minute of the specified hour
dateObj.getSeconds()	0-59	Second within the specified minute
dateObj.setTime(*val*)	0-...	Milliseconds since 1/1/70 00:00:00 GMT
dateObj.setYear(*val*)	70-...	Specified year minus 1900; four-digit year for 2000+
dateObj.setMonth(*val*)	0-11	Month within the year (January = 0)
dateObj.setDate(*val*)	1-31	Date within the month
dateObj.setDay(*val*)	0-6	Day of week (Sunday = 0)
dateObj.setHours(*val*)	0-23	Hour of the day in 24-hour time
dateObj.setMinutes(*val*)	0-59	Minute of the specified hour
dateObj.setSeconds(*val*)	0-59	Second within the specified minute

Date Calculations

Performing calculations with dates frequently requires working with the millisecond values of the Date objects. This is the surest way to compare, add, and subtract date values. To demonstrate a few Date object machinations, Listing 12-1 displays the current date and time and one way to calculate the date and time seven days from now.

LISTING 12-1

Date Object Calculations

HTML: jsb-12-01.html

```
<!DOCTYPE html>
<html>
   <head>
```

```
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Date Calculation</title>
      <script type="text/javascript" src="../jsb-global.js"></script>
      <script type="text/javascript" src="jsb-12-01.js"></script>
   </head>
   <body>
      <h1>Date Calculation</h1>
      <form id="date-form" action="date-calc.php">
         <p>
            <label for="today">Today is:</label>
            <input type="text" id="today" name="today" size="80">
         </p>
         <p>
            <label for="nextWeek">Next week will be:</label>
            <input type="text" id="nextWeek" name="nextWeek" size="80">
         </p>
         <p>
            <input type="submit" id="submit">
         </p>
      </form>
   </body>
</html>
```

JavaScript: jsb-12-01.js

```
// run script when the page has loaded
addEvent(window, 'load', calcDates);

// calculate dates and plug into document
function calcDates()
{
   var dTodaysDate = new Date();

   // plug in today
   var oToday = document.getElementById('today');
      if (oToday)
      {
         oToday.value = dTodaysDate;
      }

   // plug in next week
   var oNextWeek = document.getElementById('nextWeek');
      if (oNextWeek)
      {
         // today in milliseconds
         var msToday = dTodaysDate.getTime();

         // one week = 1000 milliseconds * 60 seconds * 60 minutes
         //            * 24 hours * 7 days
         var msOneWeek = 1000 * 60 * 60 * 24 * 7;
```

continued

LISTING 12-1 *(continued)*

```
          // next week in milliseconds
          var msNextWeek = msToday + msOneWeek;

          // next week as date object
          var dAWeekFromNow = new Date(msNextWeek);

          // plug in string value
          oNextWeek.value = dAWeekFromNow;
       }
}
```

The script first creates a new `Date` object and assigns it to the variable `dTodaysDate`. Then it displays the current date and time by setting the value of the input field `today` equal to `dTodaysDate`. Although you might expect that this would insert a copy of the `Date` object into the input field, what really happens is that it uses the `toString()` method of the `Date` object to output a string such as

```
Sun Dec 05 2010 16:47:20 GMT-0800 (Pacific Standard Time)
```

Next, we display the date and time one week from now. To work with milliseconds, we start by assigning the current date and time in milliseconds to variable `msToday`. To add one week to this, we need to know how many milliseconds there are in a week: 1000 ms. per second, 60 seconds per minute, 60 minutes per hour, 24 hours per day, and seven days per week. The product of all these we store in variable `msOneWeek`. We add that week of milliseconds to the current date and time, and then use the result to create a new `Date` object, which we store in variable `dAWeekFromNow`. Displaying the resultant date and time is done the same way as before — simply by assigning the new `Date` object to an input field's `value` attribute.

To add time intervals to or subtract time intervals from a `Date` object, you can use a shortcut that doesn't require the millisecond conversions. By combining the date object's `set` and `get` methods, you can let the `Date` object work out the details. For example, in Listing 12-1, you could replace the "next week" logic with:

```
dTodaysDate.setDate(dTodaysDate.getDate() + 7);
oNextWeek.value = dTodaysDate;
```

Because JavaScript tracks the date and time internally as milliseconds, the accurate date appears in the end, even if the new date is into the next month. JavaScript automatically takes care of figuring out how many days there are in a month, as well as in leap years.

Many other-quirks and complicated behavior await you if you script dates in your page. As later chapters demonstrate, however, the results may be worth the effort.

Exercises

1. Create a web page that has one form field for entering the user's email address, and a Submit button. In the accompanying JavaScript script, write a presubmission validation routine that verifies that the text field contains the @ symbol used in all email addresses before you allow submission of the form.
2. Given the string `"Internet Explorer"`, fill in the blanks of the `string.substring()` method parameters here that yield the results shown to the right of each method call.

```
var myString = "Internet Explorer";
myString.substring(___,___)     // result = "Int"
myString.substring(___,___)     // result = "plorer"
myString.substring(___,___)     // result = "net Exp"
```

3. Consider this logic for extracting the part of a string that comes after the first word:

```
var stringA = "The Painted Bird";
var firstSpace = stringA.indexOf(" ");
var excerpt = stringA.substring(firstSpace + 1);
```

What will happen in this logic if `stringA` does not contain a space? Work out what you think should occur, then run the script to see if you're right.

4. Fill in the rest of the function in the listing that follows so that it looks through every character of the entry field and counts how many times the letter *e* (either upper- or lowercase) appears in the field. (Hint: All that is missing is a `for` repeat loop.)

```
var inputString = 'Elephantine';
var count = 0;

MISSING CODE

var msg = "The number of e's in '" + this.mainstring.value + "'";
msg += " is " + count;
alert(msg);
```

5. Create a page that has two fields and one button. The button should trigger a function that generates two random numbers between 1 and 6, placing each number in one of the fields. (Think of using this page as a substitute for rolling a pair of dice in a board game.)
6. Create a script that displays the number of days between today and next Christmas.

CHAPTER

13

Scripting Frames and Multiple Windows

IN THIS CHAPTER

Relationships among frames in the browser window

How to access objects and values in other frames

How to control navigation of multiple frames

Communication skills between separate windows

One of the attractive aspects of JavaScript for some applications on the client is that it allows user actions in one frame or window to influence what happens in other frames and windows. In this section of the tutorial, you extend your existing knowledge of object references to the realm of multiple frames and windows.

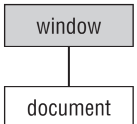
Frames: Parents and Children

You've seen in earlier top-level hierarchy illustrations (such as Figure 6-2) that the `window` object is at the top of the chart. The `window` object also has several synonyms, which stand in for the `window` object in special cases. For instance, in Chapter 10, "Window and Document Objects," you learned that `self` is synonymous with `window` when the reference applies to the same window that contains the script's document. In this lesson, you learn the roles of three other references that point to objects behaving as windows: `frame`, `top`, and `parent`.

Loading an ordinary HTML document into the browser creates a model in the browser that starts out with one `window` object and the document it contains. The top rungs of the hierarchy model are as simple as can be, as shown in Figure 13-1. This is where references begin with `window` or `self` (or with `document` because the current window is assumed).

FIGURE 13-1

Single-frame window and document hierarchy.



To frame or not to frame?

Frames and iFrames are valid HTML and their use might be seen as desirable for accomplishing some tasks. However, frame-based web sites pose serious usability and accessibility issues that can make them less than ideal for general web site development.

The fundamental problem with framesets is that any given combination of framed pages is not directly addressable. That is, if you navigate through a frameset to bring up a particular combination of content in the frames, the URL in the browser's address bar will remain that of the parent frameset document, not that of the actual content you're viewing. This lack of unique addressing of frame-paged content prevents search engines from indexing content and prevents other people from publishing links to the framed content, bookmarking them on their computers, or sharing them in email. Framesets can also be disorienting or completely inaccessible for the users of mobile devices and assistive technology such as magnifiers and screen-readers.

Web usability expert Jakob Nielsen wrote ''Why Frames Suck (Most of the Time)'' back in 1996. The situation hasn't improved in all the years since.

Alternatives to frames that avoid these problems include using server-side scripting to combine multiple source pages, and using the CSS properties `overflow: auto`, to create fields of scrolling content, and `position: fixed`, to establish sections that don't move when the rest of the page scrolls.

In this book, we assume that you've considered the pros and cons of using frames and may have a web development problem — or mere curiosity — that requires their use. We want you to know how to manipulate them with JavaScript if you have to, and they make for great exercise in navigating the DOM, but we don't recommend their use in most circumstances.

The instant a framesetting document loads into a browser, the browser starts building a slightly different hierarchy model. The precise structure of that model depends entirely on the structure of the frameset defined in that framesetting document. Consider the following skeletal frameset definition:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Frameset//EN"
        "http://www.w3.org/TR/html4/frameset.dtd">
<html>
   <head>
      <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
      <title>Title of all pages in this frameset</title>
   </head>
   <frameset cols="50%,50%">
      <frame name="leftFrame" src="somedoc1.html" title="Frame 1">
      <frame name="rightFrame" src="somedoc2.html" title="Frame 2">
      <noframes>
```

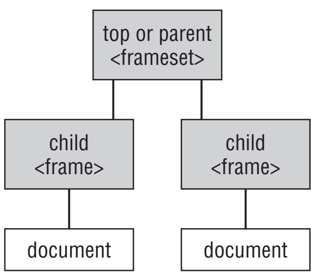
```
            ...
         </noframes>
      </frameset>
   </html>
```

This markup splits the browser window into two frames side by side, with a different document loaded into each frame. The model conveys both the structure (parent with children) and the layout (the relative sizes of the frames and whether they're set up in columns or rows). Because frames markup is not backward-compatible — it displays absolutely nothing in browsers that don't support frames — we also provide a no-frames alternative, which might be actual content or a hyperlink to a frameless page.

Framesets establish relationships among the frames in the collection. Borrowing terminology from the object-oriented programming world, the framesetting document loads into a *parent window*. Each of the frames defined in that parent window document is a *child frame*. Figure 13-2 shows the hierarchical model of a two-frame environment. This illustration reveals a lot of subtleties about the relationships among framesets and their frames.

FIGURE 13-2

Two-frame window and document hierarchy.



It is often difficult at first to visualize the frameset as a window object in the hierarchy. After all, with the exception of the URL showing in the browser's Location/Address field, you don't see anything about the frameset in the browser. But that window object exists in the object model. Notice, too, that in the diagram the framesetting parent window has no document object showing. This may also seem odd, because the window obviously requires an HTML file containing the specifications for the frameset. In truth, the parent window has a `document` object associated with it, but it is omitted from the diagram to better portray the relationships among parent and child windows. A frameset parent's document cannot itself contain most of the typical HTML objects such as forms and controls, so references to the parent's document are rarely, if ever, used.

If you add a script to the framesetting document that needs to access a property or method of that window object, references are like any single-frame situation. Think about the point of view of a script located in that window. Its immediate universe is the very same window.

Things get more interesting when you start looking at the child frames. Each of these frames contains a `document` object whose content you see in the browser window, and the structure is such that each frame's document is entirely independent of the other. It is as though each document lived in its own browser window. Indeed, that's why each child frame is also a window type of object. A frame has the same kinds of properties and methods as the `window` object that occupies the entire browser.

From the point of view of either child window in Figure 13-2, its immediate container is the `parent` window. When a `parent` window is at the top of the hierarchical model loaded in the browser, that window is also referred to as the `top` object.

References Among Family Members

Given the frame structure of Figure 13-2, it's time to look at how a script in any one of those windows can access objects, functions, or variables in the others. An important point to remember about this facility is that if a script has access to an object, function, or global variable in its own window, that same item can be reached by a script from another frame in the hierarchy (provided that both documents come from the same web server).

A script reference may need to take one of three possible routes in the two-generation hierarchy described so far: parent to child; child to parent; or child to child. Each of the paths between these windows requires a different reference style.

Parent-to-child references

Probably the least common direction taken by references is when a script in the parent document needs to access some element of one of its frames. Since the parent can contain one or more frames, the parent maintains an array of the child frame objects. You can address a frame by array syntax or by the name you assign to it with the `id` or `name` attribute inside the `<frame>` tag. In the following examples of reference syntax, we substitute a placeholder named `ObjFuncVarName` for whatever object, function, or global variable you intend to access in the distant window or frame. Remember that each visible frame contains a `document` object, which generally is the container of elements you script; be sure that references to the elements include `document`. With that in mind, a reference from a parent to one of its child frames can follow any one of these models:

```
[window.]frames[n].ObjFuncVarName
[window.]frames["frameName"].ObjFuncVarName
[window.]frameName.ObjFuncVarName
```

Numeric index values for frames are based on the order in which their `<frame>` tags appear in the framesetting document. You will make your life easier, however, if you assign recognizable names to each frame and use the frame's name in the reference.

Child-to-parent references

It is not uncommon to link scripts to the parent frameset document (in the `head` portion) that multiple child frames or multiple documents in a frame use as a common script library. Because they load in the frameset, these scripts load only once while the frameset is visible. If other documents from the same server load into the frames over time, they can take advantage of the parent's scripts without having to load their own copies into the browser.

From the child's point of view, the next level up the hierarchy is called the `parent`. Therefore, a reference from a child frame to items at the parent level is simply:

```
parent.ObjFuncVarName
```

If the item accessed in the parent is a function that returns a value, the returned value transcends the parent/child borders down to the child without hesitation.

```
var sValue = parent.FuncName();
```

When the parent window is also at the top of the object hierarchy currently loaded into the browser, you can optionally refer to it as the *top window,* as in:

```
top.ObjFuncVarName
```

Using the `top` reference can be hazardous if for some reason your web page gets displayed in some other web site's frameset. What if your top window is not the master frameset's top window? Therefore, we recommend using the `parent` reference whenever possible (unless you want your script to fail when nested inside an unwanted framer of your web site).

Child-to-child references

The browser needs a bit more assistance when it comes to getting a child window to communicate with one of its siblings. One of the properties of any window or frame is its `parent` (whose value is `null` for a single window). A reference must use the `parent` property to work its way out of the current frame to a point that both child frames have in common — the parent, in this case. When the reference is at the parent level, the rest of the reference can carry on as though it were starting at the parent. Thus, from one child to one of its siblings, you can use any of the following reference formats:

```
parent.frames[n].ObjFuncVarName
parent.frames["frameName"].ObjFuncVarName
parent.frameName.ObjFuncVarName
```

A reference from the other sibling back to the first looks the same, but the `frames[]` array index or `frameName` part of the reference differs. Of course, much more complex frame hierarchies exist in HTML. Even so, the object model and referencing scheme provide a solution for the most deeply nested and gnarled frame arrangement you can think of — following the same precepts you just learned.

Frame-Scripting Tips

One of the first mistakes that frame-scripting newcomers make is writing immediate script statements that call on other frames while the pages load. The problem here is that you cannot rely on the document loading sequence to follow the frameset source-code order. All you know for sure is that the parent document *begins* loading first. Regardless of the order of `<frame>` tags, child frames can begin loading in any sequence. Moreover, a frame's loading time depends on other elements in the document, such as images or Java applets.

Fortunately, you can use a certain technique to initiate a script when all the documents in the frameset are completely loaded. Just as the `load` event for a window fires when that window's document is fully loaded, a parent's `load` event fires after the `load` events in its child frames have fired. When you specify a `window.onload` event handler in a script linked to the frameset document, it's the equivalent of applying the event handler to the `frameset` element. That handler might invoke a function in the framesetting document that then has the freedom to tap the objects, functions, or variables of all frames throughout the object hierarchy.

Make special note that a reference to a frame as a type of window object is quite separate from a reference to the `frame` element object. An element object is one of those DOM element nodes in the document node tree (see Chapter 6, "Browser and Document Objects"). The properties and methods of this node differ from the properties and methods that accrue to a window-type object. It may be a difficult distinction to grasp, but it's an important one. The way you reference a frame — as a window object or element node — determines which set of properties and methods are available to your scripts.

Cross-Reference

See Chapter 26, "Generic HTML Element Objects," for a more detailed introduction to element node scripting. ■

If you start with a reference to the `frame` element object, you can still reach a reference to the `document` object loaded into that frame, but the syntax is different, depending on the browser. IE4+ and Safari let you use the same `document` reference as for a window; Mozilla-based browsers follow the W3C DOM standard more closely, using the `contentDocument` property of the frame element. To accommodate both syntaxes, you can build a reference as follows:

```
var docObj;
var frameObj = document.getElementById("myFrame");
   if (frameObj.contentDocument)
   {
      docObj = frameObj.contentDocument;
   }
   else
   {
      docObj = frameObj.document;
   }
```

About iframe Elements

The `iframe` element is supported as a scriptable object in IE4+, Mozilla-based browsers, and Safari (among other modern browsers). It is often used as a way to fetch and load HTML from a server without disturbing the current HTML page. It's not uncommon for an `iframe` to be hidden from view while scripts handle all the processing between it and the main document. (Today, the trick of asynchronously loading data from the server into an already-downloaded page is often accomplished using `XMLHttpRequest()` — see Chapter 39, "Ajax, E4X, and XML," on the CD.)

An `iframe` element becomes another member of the current window's `frames` collection, but you may also reference the `iframe` as an element object through W3C DOM `document.getElementById()` terminology. As with the distinction between the traditional frame-as-window object and DOM element object, a script reference to the `document` object within an `iframe` element object needs special handling. See Chapter 27, "Window and Frame Objects," for additional details.

Highlighting Footnotes: A Frameset Scripting Example

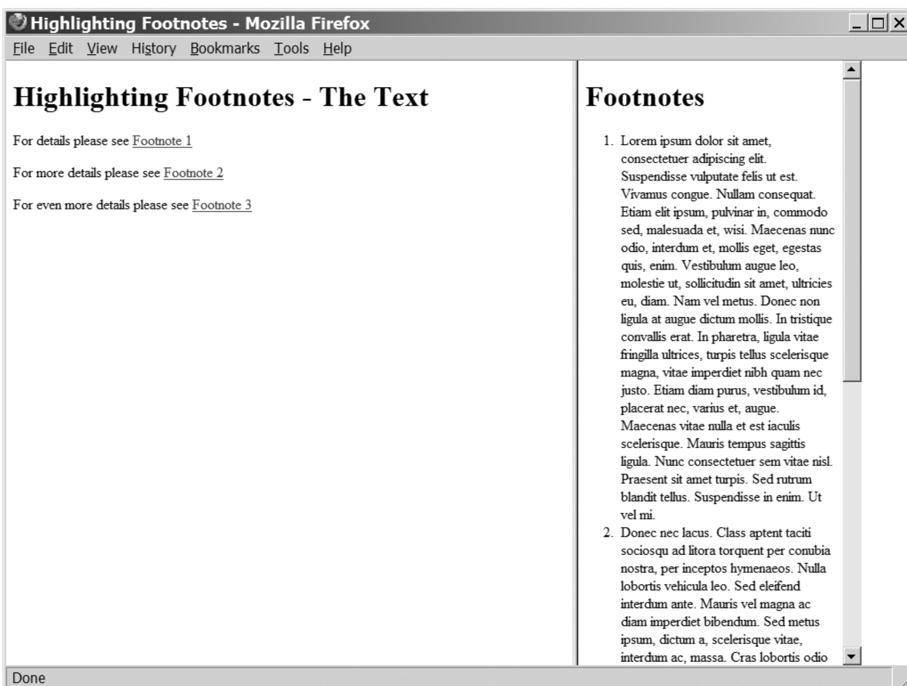
For a taste of inter-window scripting, consider a frameset that highlights a footnote in one frame when the corresponding link is clicked in the document text. Figure 13-3 shows what this frameset looks like, and Listing 13-1 provides the markup and scripting.

The three HTML files establish a frameset with two child frames, one for the main text and one for the footnotes. Clicking on a footnote link in the text brings the selected footnote to the top of the footnote frame. It does this by adding the element ID of the selected footnote to the 'hash' of the URL:

```
<a target="frameFootnotes" ↩
href="jsb-13-01-frame-footnotes.html#footnote-01">Footnote 1</a>
```

FIGURE 13-3

Highlighting Footnotes frameset.



The `target` attribute identifies the child frame while the `href` specifies which document will appear in that frame. The final part of the `href` URL `#footnote-01` matches up with the footnote markup:

```
<li id="footnote-01">Lorem ipsum dolor ...
```

It is basic browser functionality that, when the ID of an element is set as the hash of a URL, the page scrolls up (if it can) to position that element at the top of the window.

JavaScript and CSS work together to highlight the selected footnote: when the user clicks on a footnote link in the text window, JavaScript assigns a "selected" class to the footnote element. CSS styles that class with a background color.

LISTING 13-1

Highlighting Footnotes

HTML: jsb-13-01-frameset.html

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Frameset//EN"
   "http://www.w3.org/TR/html4/frameset.dtd">
<html>
   <head>
```

continued

LISTING 13-1 *(continued)*

```
      <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
      <title>Highlighting Footnotes</title>
      <link href="jsb-13-01-frameset.css" rel="stylesheet"
         type="text/css" media="all">
   </head>
   <frameset cols="66%,33%">
      <frame name="frameText" src="jsb-13-01-frame-text.html" title="Frame 1">
      <frame name="frameFootnotes" src="jsb-13-01-frame-footnotes.html"
         title="Frame 2">
      <noframes>
         <body>
            <h1>Text with Footnotes</h1>
            <p>Proceed to <a href="jsb-13-01-frame-text-footnotes.html">
                  combined text and footnotes</a>.</p>
         </body>
      </noframes>
   </frameset>
</html>
```

Stylesheet: jsb-13-01-frameset.css

```
/* limit the width of the page */
html
{
   width: 60em;
   max-width: 100%;
}
```

HTML: jsb-13-01-frame-text.html

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
   "http://www.w3.org/TR/html4/loose.dtd">
<html>
   <head>
      <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
      <title>Highlighting Footnotes - The Text</title>
      <script type="text/javascript" src="../jsb-global.js"></script>
      <script type="text/javascript" src="jsb-13-01-frame-text.js"></script>
   </head>
   <body>
      <h1>Highlighting Footnotes - The Text</h1>

      <p>For details please see <a target="frameFootnotes"
            href="jsb-13-01-frame-footnotes.html#footnote-01">Footnote 1</a></p>

      <p>For more details please see <a target="frameFootnotes"
            href="jsb-13-01-frame-footnotes.html#footnote-02">Footnote 2</a></p>
```

```
      <p>For even more details please see <a target="frameFootnotes"
            href="jsb-13-01-frame-footnotes.html#footnote-03">Footnote 3</a></p>

   </body>
</html>
```

JavaScript: jsb-13-01-frame-text.js

```
// run script when the page has loaded
addEvent(window, 'load', initialize);

// global memory of the last footnote highlighted
var oFootnote = null;
var sFootnoteClass = '';

// apply behaviors when document has loaded
function initialize()
{
   // do this only if the browser can handle DOM methods
   if (document.getElementById)
   {
      // collect anchors
      var aAnchors = document.getElementsByTagName('a');

      // for each anchor...
      for (var i = 0; i < aAnchors.length; i++)
      {
         // get the target of the link
         var sTarget = aAnchors[i].getAttribute('target');

            // if there is a target and it's the footnotes frame
            if (sTarget && sTarget == 'frameFootnotes')
            {
               addEvent(aAnchors[i], 'click', highlightFootnote);
            }
      }
   }
}

// highlight the selected footnote
function highlightFootnote()
{
      // restore previously highlighted footnote (if any)
      if (oFootnote)
      {
         oFootnote.className = sFootnoteClass;
      }

   // get the HREF of the clicked anchor
   var sHref = this.getAttribute('href');
```

continued

LISTING 13-1 *(continued)*

```
   // get the 'hash' of the URL
   var iHash = sHref.indexOf('#');
   // everything after the hash mark is the element Id
   var sFootnoteId = sHref.substr(iHash + 1);

   // the target of the link is the frame Id
   var sFrameId = this.getAttribute('target');

      // if the frame exists...
      if (parent[sFrameId])
      {
         // point to the indicated footnote
         oFootnote = parent[sFrameId].document.getElementById(sFootnoteId);
            // if it exists...
            if (oFootnote)
            {
               // remember its original class to restore it later
               sFootnoteClass = oFootnote.className;

               // set its new class
               oFootnote.className = 'selected';
            }
      }
}
```

HTML: jsb-13-01-frame-footnotes.html

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
   "http://www.w3.org/TR/html4/loose.dtd">
<html>
   <head>
      <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
      <title>Footnotes</title>
      <link href="jsb-13-01-frame-footnotes.css" rel="stylesheet"
         type="text/css" media="all">
   </head>
   <body>
      <h1>Footnotes</h1>

      <ol class="footnotes">
         <li id="footnote-01">Lorem ipsum dolor sit amet, consectetuer
adipiscing elit. Suspendisse vulputate felis ut est. Vivamus congue.
Nullam consequat. Etiam elit ipsum, pulvinar in, commodo sed, malesuada
et, wisi. Maecenas nunc odio, interdum et, mollis eget, egestas quis, enim.
Vestibulum augue leo, molestie ut, sollicitudin sit amet, ultricies eu, diam.
Nam vel metus. Donec non ligula at augue dictum mollis. In tristique convallis
erat. In pharetra, ligula vitae fringilla ultrices, turpis tellus scelerisque
magna, vitae imperdiet nibh quam nec justo. Etiam diam purus, vestibulum id,
```

```
placerat nec, varius et, augue. Maecenas vitae nulla et est iaculis scelerisque.
Mauris tempus sagittis ligula. Nunc consectetuer sem vitae nisl. Praesent sit
amet turpis. Sed rutrum blandit tellus. Suspendisse in enim. Ut vel mi.</li>

         <li id="footnote-02">Donec nec lacus. Class aptent taciti sociosqu ad
litora torquent per conubia nostra, per inceptos hymenaeos. Nulla lobortis
vehicula leo. Sed eleifend interdum ante. Mauris vel magna ac diam imperdiet
bibendum. Sed metus ipsum, dictum a, scelerisque vitae, interdum ac, massa.
Cras lobortis odio at augue. Aenean fermentum bibendum purus. Ut congue
interdum turpis. Ut quis mi a ligula viverra hendrerit. Nulla facilisi.
Nullam lacinia, ligula ac congue feugiat, risus diam fringilla tellus,
condimentum volutpat eros risus sit amet dolor.</li>

         <li id="footnote-03">Proin et urna. Morbi in odio. In mauris tellus,
tincidunt vitae, ultrices a, sollicitudin sit amet, nulla. Praesent consequat,
lectus auctor imperdiet dapibus, risus turpis feugiat orci, sit amet tempus quam
erat a purus. Nullam neque nulla, fringilla ut, dictum id, luctus placerat,
urna. Sed vel velit. Fusce eget erat a quam auctor pulvinar. Phasellus vel
neque eu risus feugiat sagittis. Nulla est. Duis sed massa. Duis ac leo quis
sapien fringilla posuere. Nam facilisis lorem. Nullam bibendum, wisi quis
ultrices bibendum, sem pede volutpat sem, ut ultrices odio nisl ut ligula.
Cras ligula est, tempus et, tempus vitae, pretium eget, ipsum. Integer
Blandit mattis wisi. Curabitur eget tellus at neque vulputate dignissim.
Phasellus leo nibh, euismod ac, congue vel, interdum et, turpis. Fusce
viverra aliquet wisi.</li>
      </ol>
   </body>
</html>
```

Stylesheet: jsb-13-01-frame-footnotes.css

```
/* style the selected footnote */
ol.footnotes li.selected
{
   background-color: aqua;
}
```

Let's walk through the JavaScript to see how the highlighting works. The script is linked to the `frame-text.html` document because it's the act of clicking the links in that document that triggers the highlighting event. The `addEvent()` function call tells JavaScript to run the `initialize()` function when the document has loaded. That function loops through all the anchor elements in the document. If it finds a `target` attribute set to `frameFootnotes`, it applies the `onclick` event to run the function `highlightFootnote()`.

When the user clicks on one of these links in the document text, the `highlightFootnote()` function gives the anchor element the `selected` class. To make it easy to restore a link to its previous class name later, the script stores the current anchor element pointer and its class attribute in global variables `oFootnote` and `sFootnoteClass`. So, the first thing it does when a link is clicked is to restore the previously-clicked link, if any.

To set the `class` attribute of the footnote in the other child frame, the script needs to determine the names of the frame and the footnote element. The frame name is exactly the same as the `target` attribute of the link that's clicked. The footnote element is whatever element in the footnote frame has the ID equal to the hash at the end of the link's URL.

If the named frame exists and the named element exists, the script can safely save the element's existing class (if any) and set the class to `selected`. Finally, the style sheet linked to the footnotes document applies a background color to any footnote with the `selected` class.

There are two important scripting principles illustrated with Listing 13-1. First, the fundamental operation of this page — bringing the selected footnote to the top of the footnote frame — is handled entirely by HTML without any help from JavaScript. This ensures that the page will be usable when viewed with user agents that don't support JavaScript, such as mobile devices, or with browsers in which the user has disabled JavaScript. This is an example of *progressive enhancement*, which stipulates that our scripting efforts should enhance already functional pages rather than supplying mission-critical functionality. We want to improve the experience for those who are running JavaScript and not break the page for those who aren't.

Second, we're using JavaScript to indicate that the footnotes are "selected" but not how they should be styled. We could have scripted:

```
oFootnote.style.backgroundColor = 'aqua';
```

Instead, the styling is left to CSS. This is an example of *separation of development layers*, whereby HTML structure and content are kept separate from JavaScript behavior and CSS presentation. This mode of development makes it quick and easy to create new code, tweak existing code, and apply old code to new projects. For example, when the web site designer changes the look of the site — and rest assured nearly every site *will* be changed either during development or afterward — cosmetic changes should be within the grasp of someone who knows CSS, without requiring the services of a JavaScript programmer.

References for Multiple Windows

In Chapter 10 you saw how to create a new window and communicate with it by way of the `window` object reference returned from the `window.open()` method. In this section, we show you how one of those subwindows can communicate with objects, functions, and variables in the window or frame that creates the subwindow.

Every `window` object has a property called `opener`. This property contains a reference to the window or frame that held the script whose `window.open()` statement generated the subwindow. For the main browser window and frames therein, this value is `null`. Because the `opener` property is a valid window reference (when its value is not `null`), you can use it to begin the reference to items in the original window — just like a script in a child frame uses `parent` to access items in the parent document. (The `parent` keyword isn't used for a subwindow's opener, however.)

Listing 13-2 contains two documents that work together in separate windows. `jsb-13-02-main.html` displays a button that opens a smaller window and loads `jsb-13-02-sub.html` into it, and another button that closes the subwindow. The main window document also contains a text field that gets filled in when you enter text in a corresponding field in the subwindow.

Note

You may have to turn off pop-up blocking temporarily to experiment with these examples.

Because so many people have set their browsers to block pop-up windows, and because pop-ups can be so disorienting to folks using assistive technology such as screen readers, and simply irritating to everyone else, we don't recommend that you create subwindows with JavaScript in real-world applications — and especially not for any critical function. Instead, use other techniques, such as CSS pop-ups, that gracefully degrade to simple HTML navigation in the absence of styling and scripting. ■

In the main window's script, the `openSubWindow()` function generates the new window. Because the `closeSubWindow()` function needs to know which window to close, we store the pointer to the new window in a global variable.

LISTING 13-2

A Main Window Document

HTML: jsb-13-02-main.html

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Opener</title>
      <script type="text/javascript" src="../jsb-global.js"></script>
      <script type="text/javascript" src="jsb-13-02-main.js"></script>
   </head>
   <body>
      <h1>Opener</h1>
      <form action="">
         <p>
            <input type="button" value="Open Sub Window" id="open-sub-window">
            <input type="button" value="Close Sub Window" id="close-sub-window">
         </p>
         <p>
            <label>Text incoming from subwindow:</label>
            <input type="text" id="output">
         </p>
      </form>
   </body>
</html>
```

JavaScript: jsb-13-02-main.js

```
// run script when the page has loaded
addEvent(window, 'load', initialize);

// global pointer to sub-window
var oSubWindow;
```

continued

LISTING 13-2 *(continued)*

```
// apply behaviors when document has loaded
function initialize()
{
   // do this only if the browser can handle DOM methods
   if (document.getElementById)
   {
      // point to button
      var oButton = document.getElementById('open-sub-window');

         // if it exists, apply behavior
         if (oButton)
         {
            addEvent(oButton, 'click', openSubWindow);
         }

      // point to button
      oButton = document.getElementById('close-sub-window');

         // if it exists, apply behavior
         if (oButton)
         {
            addEvent(oButton, 'click', closeSubWindow);
         }
   }
}

// open (create) sub window
function openSubWindow()
{
   // store sub-window pointer in global variable
   oSubWindow = window.open("jsb-13-02-sub.html", "sub", "height=200,width=300");
}

// close (destroy) sub window
function closeSubWindow()
{
      // if the sub-window has been opened, close it
      if (oSubWindow)
      {
         oSubWindow.close();
      }
}
```

HTML: jsb-13-02-sub.html

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Sub-Document</title>
      <script type="text/javascript" src="../jsb-global.js"></script>
```

```
      <script type="text/javascript" src="jsb-13-02-sub.js"></script>
   </head>
   <body>
      <h1>Sub-Document</h1>
      <form id="sendForm" action="">
         <p>
            <label for="input">Enter text to be copied to the main window:</label>
            <input type="text" id="input">
         </p>
         <p>
            <input type="button" value="Send Text to Opener" id="send-text">
         </p>
      </form>
   </body>
</html>
```

JavaScript: jsb-13-02- sub.js

```
// run script when the page has loaded
addEvent(window, 'load', initialize);

// apply behaviors when document has loaded
function initialize()
{
   // do this only if the browser can handle DOM methods
   if (document.getElementById)
   {
      // point to button
      var oButton = document.getElementById('send-text');

         // if it exists, apply behavior
         if (oButton)
         {
            addEvent(oButton, 'click', copyToOpener);
         }
   }
}

// copy input text to output in parent
function copyToOpener()
{
   // find the input field in the current window
   var oInput = document.getElementById('input');

   // find the output field in the parent window
   var oOutput = opener.document.getElementById('output');

      // if they both exist, copy input text
      if (oInput && oOutput)
      {
         oOutput.value = oInput.value;
      }
}
```

All the action in the subwindow document stems from the `onclick` event handler assigned to the Send Text button. It triggers the `copyToOpener()` function, which assigns the subwindow input field's value to the value of the output field in the opener window's document. Remember, the contents of each window and frame belong to a document. Even after your reference targets a specific window or frame, the reference must continue helping the browser find the ultimate destination, which generally is some element of the document.

Just one more lesson to go before we let you explore all the details elsewhere in the book. We'll use the final tutorial chapter to show you some fun things you can do with your web pages, such as changing images when the user rolls the mouse over a picture.

Exercises

Before answering the first three questions, study the structure of the following frameset for a web site that lists college courses:

```
<html>
   <head>
      ...
   </head>
   <frameset rows="85%,15%">
      <frameset cols="20%,80%">
         <frame name="mechanics" src="history101M.html">
         <frame name="description" src="history101D.html">
      </frameset>
      <frameset cols="100%">
         <frame name="navigation" src="navigator.html">
      </frameset>
   </frameset>
   <noframes>
      ...
   </noframes>
</html>
```

1. Each document that loads into the description frame links to a JavaScript script that uses the `onload` event handler to store a course identifier value in the framesetting document's global variable `currCourse`. Write the function that sets this value to `history101`.
2. Draw a block diagram that describes the hierarchy of the windows and frames represented in the frameset definition.
3. Write the HTML markup and JavaScript statements located in the navigation frame that together perform two actions: load the file `french201M.html` into the mechanics frame, and load the file `french201D.html` into the description frame.
4. While a frameset is still loading, a JavaScript error message suddenly appears, saying, "window.document.navigation.form.selector is undefined." What do you think is happening in the application's scripts, and how can you solve the problem?
5. A script in a child frame of the main window uses `window.open()` to generate a second window. How can a script in the second window access the `location` object (URL) of the top (framesetting) window in the main browser window?

CHAPTER

14

Images and Dynamic HTML

The previous eight lessons have been intensive, covering a lot of ground for both programming concepts and JavaScript. Now it's time to apply those fundamentals to learning more advanced techniques. We cover two areas here. First, we show you how to implement the ever-popular *mouse rollover,* in which images swap when the user rolls the cursor around the screen. Then we introduce you to techniques for modifying a page's style and content after the page has loaded.

Although image swapping on mouse rollover can be accomplished more easily using Cascading Style Sheets (CSS), we'll demonstrate how to do it with JavaScript because you'll probably see a lot of it in legacy scripts, and because it's a handy way to illustrate how to manipulate image objects.

IN THIS CHAPTER

How to pre-cache images

How to swap images for mouse rollovers

Changing style sheet settings

Modifying body content dynamically

The Image Object

One of the objects contained by the document is the Image object — supported in all scriptable browsers since the days of NN3 and IE4. Image object references for a document are stored in the object model as an array belonging to the `document` object. Therefore, you can reference an image by array index or image name. Moreover, the array index can be a string version of the image's name. All the following are valid references to an image object:

```
document.getElementById("imageName")
document.getElementsByTagName("img")[n]
document.images[n]
document.images["imageName"]
document.imageName
```

(The last reference will work only if `imageName` doesn't include any valid HTML ID characters that aren't permitted in JavaScript names, such as a hyphen or period.)

We are no longer constrained by ancient scriptable browser limitations that required an image be encased within an `a` element to receive mouse events. You

may still want to do so if a click is intended to navigate to a new URL, but to use a visitor's mouse click to trigger local JavaScript execution, it's better to let the `img` element's event handlers do all the work.

Interchangeable images

The advantage of having a scriptable image object is that a script can change the image occupying the rectangular space already occupied by an image. In current browsers, the images can even change size, with surrounding content automatically reflowing around the resized image.

The script behind this kind of image change is simple enough. All it entails is assigning a new URL to the `img` element object's `src` property. The size of the image on the page is governed by the `height` and `width` attributes set in the `<img>` tag, and the `height` and `width` properties set in the style sheet. The most common image rollovers use the same size of image for each of the rollover states.

Pre-caching images

Images take extra time to download from a web server until they are stored in the browser's cache. If you design your page so that an image changes in response to user action, you usually want the same fast response that users are accustomed to in other programs. Making the user wait seconds for an image to change can detract from enjoyment of the page.

JavaScript comes to the rescue by enabling scripts to load images into the browser's memory cache without displaying the image, a technique called *pre-caching images*. The tactic that works best is to preload the image into the browser's image cache while the page initially loads. Users are less impatient for those few extra seconds when the main page loads, than they are waiting for an image to download in response to some mouse action. However, keep in mind that there is a balance. Today's audience tends to give a page less than 10 seconds to load before moving on. Although pre-caching images means a fast response to a mouse action, if you pre-cache too many images, you risk losing web site visitors before they can see your amazing image swapping in response to mouse actions.

Pre-caching an image requires constructing an image object in memory. An image object created in memory differs in some respects from the document `img` element object that you create with the `<img>` tag. Memory-only objects are created by script, and you don't see them on the page at all. But their presence in the document code forces the browser to load the images as the page loads. The object model provides an `Image` object constructor function to create the memory type of image object as follows:

```
var myImage = new Image(width, height);
```

Parameters to the constructor function are the pixel `width` and `height` of the image. These dimensions should match the `width` and `height` attributes of the `<img>` tag. Once the image object exists in memory, you can then assign a filename or URL to the `src` property of that image object:

```
myImage.src = "someArt.gif";
```

When the browser encounters a statement assigning a URL to an image object's `src` property, the browser fetches and loads that image into the image cache. All the user sees is some extra loading information in the status bar, as though another image were in the page. By the time the entire page loads, all images generated in this manner are tucked away in the image cache. You can then assign your cached image's `src` property, or the actual image URL, to the `src` property of the document image created with the `<img>` tag:

```
document.images[0].src = myImage.src;
```

or

```
document.getElementById("myImg").src = myImage.src ;
```

The change to the image in the document is instantaneous.

Listing 14-1 demonstrates a page that has one <img> tag and a select list that enables you to replace the image in the document with any of four pre-cached images (including the original image specified for the tag). If you type this listing, you can obtain copies of the four image files from the companion CD-ROM in the Chapter 14 directory of listings. (You still must type the HTML and code, however.)

LISTING 14-1

Pre-Caching Images

HTML: jsb-14-01.html

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Image Object</title>
      <style type="text/css">
         img
         {
            height:90px; width:120px;
         }
      </style>
      <script type="text/javascript" src="../jsb-global.js"></script>
      <script type="text/javascript" src="jsb-14-01.js"></script>
   </head>
   <body >
      <h2>Image Object</h2>
      <img src="desk1.gif" id="thumbnail">
      <form>
         <select id="imageChooser">
            <option value="image1">Bands</option>
            <option value="image2">Clips</option>
            <option value="image3">Lamp</option>
            <option value="image4">Erasers</option>
         </select>
      </form>
   </body>
</html>
```

JavaScript: jsb-14-01.js

```
// initialize when the page has loaded
var oImageChooser;
addEvent(window, "load", initialize);
```

continued

LISTING 14-1 *(continued)*

```
function initialize()
{
   // get W3C DOM or IE4+ reference for an ID
   // imageChooser is the select element
   if (document.getElementById)
   {
      oImageChooser = document.getElementById("imageChooser");
   }
   else
   {
      oImageChooser = document.imageChooser;
   }

   addEvent(oImageChooser, 'change', loadCached);

}

// initialize empty array
var imageLibrary = new Array();
// pre-cache four images
imageLibrary["image1"] = new Image(120,90);
imageLibrary["image1"].src = "desk1.gif";
imageLibrary["image2"] = new Image(120,90);
imageLibrary["image2"].src = "desk2.gif";
imageLibrary["image3"] = new Image(120,90);
imageLibrary["image3"].src = "desk3.gif";
imageLibrary["image4"] = new Image(120,90);
imageLibrary["image4"].src = "desk4.gif";

// load an image chosen from select list
function loadCached()
{
   var imgChoice = oImageChooser.options[oImageChooser.selectedIndex].value;
   document.getElementById("thumbnail").src = imageLibrary[imgChoice].src;
}
```

As the page loads, it executes several statements immediately. These statements create an array that is populated with four new memory image objects. Each image object has a filename assigned to its `src` property. These images are loaded into the image cache as the page loads. Down in the `body` portion of the document, an `<img>` tag stakes its turf on the page and loads one of the images as a starting image.

A `select` element lists user-friendly names for the pictures while housing (in the option values) the names of image objects already pre-cached in memory. When the user makes a selection from the list, the `loadCached()` function extracts the selected item's value — which is a string index of the image within the `imageLibrary` array. The `src` property of the chosen image object is assigned to the `src` property of the visible `img` element object on the page, and the pre-cached image appears instantaneously.

Creating image rollovers

A favorite technique to add some pseudo-excitement to a page is to swap button images as the user rolls the cursor over them. The degree of change to the image is largely a matter of taste. The effect can be subtle (a slight highlight or glow around the edge of the original image) or drastic (a radical change of color). Whatever your approach, the scripting is the same.

When several of these graphical buttons occur in a group, we tend to organize the memory image objects as arrays, and create naming and numbering schemes that facilitate working with the arrays. Listing 14-2 shows such an arrangement for four buttons that control a slide show. The code in the listing is confined to the image-swapping portion of the application. This is the most complex and lengthiest listing of the tutorial, so it requires a bit of explanation as it goes along. It begins with a style sheet rule for each of the `img` elements located in a `controller` container.

LISTING 14-2

Image Rollovers

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Slide Show/Image Rollovers</title>
      <style type="text/css">
         div#controller img {
            height: 70px; width: 136px; padding: 5px;
            }
      </style>
      <script type="text/javascript" src="../jsb-global.js"></script>
      <script type="text/javascript">
```

Only browsers capable of handling image objects should execute statements that pre-cache images. Therefore, the entire sequence is nested inside an `if` construction that tests for the presence of the `document.images` array. In older browsers, the condition evaluates to *undefined,* which an `if` condition treats as `false`:

```
if (document.images)
{
```

Image pre-caching starts by building two arrays of image objects. One array stores information about the images depicting the graphical buttons' off position; the other is for images depicting their on position. These arrays use strings (instead of integers) as index values. The string names correspond to the names given to the visible `img` element objects whose tags come later in the source code. The code is clearer to read (for example, you know that the `offImgArray["first"]` entry has to do with the First button image). Also, as you see later in this listing, rollover images don't conflict with other visible images on the page (a possibility if you rely exclusively on numeric index values when referring to the visible images for the swapping).

After creating the array and assigning new blank image objects to the first four elements of the array, we go through the array again, this time assigning file pathnames to the `src` property of each object

stored in the array. These lines of code execute as the page loads, forcing the images to load into the image cache along the way:

```
// pre-cache all 'off' button images
var offImgArray = new Array();
offImgArray["first"] = new Image(136,70);
offImgArray["prev"] = new Image(136,70);
offImgArray["next"] = new Image(136,70);
offImgArray["last"] = new Image(136,70);

// off image array -- set 'off' image path for each button
offImgArray["first"].src = "firstoff.png";
offImgArray["prev"].src = "prevoff.png";
offImgArray["next"].src = "nextoff.png";
offImgArray["last"].src = "lastoff.png";

// pre-cache all 'on' button images
var onImgArray = new Array();
onImgArray["first"] = new Image(136,70);
onImgArray["prev"] = new Image(136,70);
onImgArray["next"] = new Image(136,70);
onImgArray["last"] = new Image(136,70);

// on image array -- set 'on' image path for each button
onImgArray["first"].src = "firston.png";
onImgArray["prev"].src = "prevon.png";
onImgArray["next"].src = "nexton.png";
onImgArray["last"].src = "laston.png";
```

Do you see the repeating pattern in the statements above? Instead of those statements, here's a way you could take advantage of the repeating pattern:

```
var offImgArray = new Array();
var onImgArray = new Array();
var ButtonArray = ["first", "prev", "next", "last"];

for (var i=0; i < ButtonArray.length; i++)
{
   var Button = ButtonArray[i];

   // pre-cache 'off' button image
   offImgArray[Button] = new Image(136,70);
   offImgArray[Button].src = Button + "off.png";

   // pre-cache 'on' button image
   onImgArray[Button] = new Image(136,70);
   onImgArray[Button].src = Button + "on.png";
}
```

As you can see in the following code, when the user rolls the mouse atop any of the visible document image objects, the `onmouseover` event handler invokes the `imageOn()` function,

passing the name of the particular image. The `imageOn()` function uses that name to synchronize the `document.images` array entry (the visible image) with the entry of the in-memory array of on images from the `onImgArray` array. The `src` property of the array entry is assigned to the corresponding document image `src` property. At the same time, the cursor changes to look like it does over active links.

```
}
// functions that swap images & status bar
function imageOn(imgName)
{
   if (document.images)
   {
      document.images[imgName].style.cursor = "pointer";
      document.images[imgName].src = onImgArray[imgName].src;
   }
}
```

The same goes for the `onmouseout` event handler, which needs to turn the image off by invoking the `imageOff()` function with the same index value.

```
function imageOff(imgName)
{
   if (document.images)
   {
      document.images[imgName].style.cursor = "default";
      document.images[imgName].src = offImgArray[imgName].src;
   }
}
```

Both the `onmouseover` and `onmouseout` event handlers set the status bar text to a friendly descriptor — at least in those browsers that still support displaying custom text in the status bar. The `onmouseout` event handler sets the status bar message to an empty string.

```
function setMsg(msg)
{
   window.status = msg;
   return true;
}
```

For this demonstration, we disable the functions that control the slide show. But we leave the empty function definitions here so they catch the calls made by the clicks of the links associated with the images.

```
// controller functions (disabled)
function goFirst()
{
}
function goPrev()
{
}
function goNext()
{
}
```

```
function goLast()
{
}
// event handler assignments
function init()
{
   if (document.getElementById)
   {
      oImageFirst = document.getElementById("first");
      oImagePrev = document.getElementById("prev");
      oImageNext = document.getElementById("next");
      oImageLast = document.getElementById("last");
   }
   else
   {
      oImageFirst = document.first;
      oImagePrev = document.prev;
      oImageNext = document.next;
      oImageLast = document.last;
   }

   addEvent(oImageFirst, 'click', goFirst);
   addEvent(oImageFirst, 'mouseover', function()
      {
         imageOn("first");
         return setMsg("Go to first picture");
      });
   addEvent(oImageFirst, 'mouseout', function()
      {
         imageOff("first");
         return setMsg("");
      });
   addEvent(oImagePrev, 'click', goPrev);
   addEvent(oImagePrev, 'mouseover', function()
      {
         imageOn("prev");
         return setMsg("Go to previous picture");
      });
   addEvent(oImagePrev, 'mouseout', function()
      {
         imageOff("prev");
         return setMsg("");
      });
   addEvent(oImageNext, 'click', goNext);
   addEvent(oImageNext, 'mouseover', function()
      {
         imageOn("next");
         return setMsg("Go to next picture");
      });
```

```
            addEvent(oImageNext, 'mouseout', function()
               {
                  imageOff("next");
                  return setMsg("");
               });
            addEvent(oImageLast, 'click', goLast);
            addEvent(oImageLast, 'mouseover', function()
               {
                  imageOn("last");
                  return setMsg("Go to last picture");
               });
            addEvent(oImageLast, 'mouseout', function()
               {
                  imageOff("last");
                  return setMsg("");
               });
         }

         // initialize when the page has loaded
         addEvent(window, "load", init);
      </script>
   </head>

   <body>
      <h1>Slide Show Controls</h1>
```

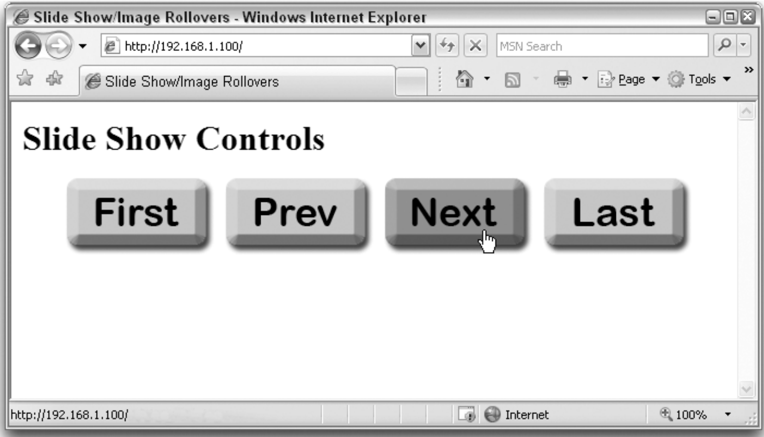
We elected to place the controller images inside a `div` element so that the images could be positioned or styled as a group. Each `img` element's `onmouseover` event handler (defined above) calls the `imageOn()` function, passing the name of the image to be swapped. Because both the `onmouseover` and `onmouseout` event handlers require a `return true` statement to work in older browsers, we combine the second function call (`to setMsg()`) with the `return true` requirement. The `setMsg()` function always returns `true` and is combined with the `return` keyword before the call to the `setMsg()` function. It's just a trick to reduce the amount of code in these event handlers.

```
      <div id="controller">
         <img src="images/firstoff.png" name="first" id="first" >
         <img src="images/prevoff.png" name="prev" id="prev" >
         <img src="images/nextoff.png" name="next" id="next" >
         <img src="images/lastoff.png" name="last" id="last" >
      </div>
   </body>
</html>
```

You can see the results of this lengthy script in Figure 14-1. As the user rolls the mouse atop one of the images, it changes from a light to dark color by swapping the entire image. You can access the image files on the CD-ROM, and we encourage you to enter this lengthy listing and see the magic for yourself.

FIGURE 14-1

Typical mouse rollover image swapping.



Rollovers Without Scripts

As cool as the rollover effect is, thanks to CSS technology, you don't always need JavaScript to accomplish rollover dynamism. You can blend CSS with JavaScript to achieve the same effect. Listing 14-3 demonstrates a version of Listing 14-2, but using CSS for the rollover effect, while JavaScript still handles the control of the slide show.

The HTML for the buttons consists of `li` elements that are sized and assigned background images of the off versions of the buttons. The text of each `li` element is surrounded by an `a` element so that CSS `:hover` pseudo-elements can be assigned. (Internet Explorer through version 7 requires this, whereas W3C DOM browsers recognize `:hover` for all elements.) When the cursor hovers atop an `a` element, the background image changes to the on version. The `onclick` event handler assignments remain in the script portion of the page, where they are performed after the page loads (to make sure the elements exist).

LISTING 14-3

CSS Image Rollovers

HTML: jsb-14-03.html

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Slide Show/Image Rollovers</title>
      <link rel="stylesheet" href="jsb-14-03.css" type="text/css">
      <script type="text/javascript" src="../jsb-global.js"></script>
      <script type="text/javascript" src="jsb-14-03.js"></script>
```

```
        <script type="text/javascript">
        </script>
    </head>
    <body>
        <h1>Slide Show Controls</h1>
        <ul id="controller">
            <li id="first"><a href="#">First</a></li>
            <li id="prev"><a href="#">Previous</a></li>
            <li id="next"><a href="#">Next</a></li>
            <li id="last"><a href="#">Last</a></li>
        </ul>
    </body>
</html>
```

CSS: jsb-14-03.css

```
#controller {
    position: relative;
    }
#controller li {
    position: absolute; list-style: none; display: block;
    height: 70px; width: 136px;
    }
#controller a {
    display: block; text-indent: -999px; height: 70px;
    }

#first {
    left: 0px;
    }
#prev {
    left: 146px;
    }
#next {
    left: 292px;
    }
#last {
    left: 438px;
    }

#first a {
    background-image: url("firstoff.png");
    }
#first a:hover {
    background-image: url("firston.png");
    }
#prev a {
    background-image: url("prevoff.png");
    }
#prev a:hover {
    background-image: url("prevon.png");
    }
```

continued

LISTING 14-3 *(continued)*

```
#next a {
   background-image: url("nextoff.png");
   }
#next a:hover {
   background-image: url("nexton.png");
   }
#last a {
   background-image: url("lastoff.png");
   }
#last a:hover {
   background-image: url("laston.png");
   }
```

JavaScript: jsb-14-03.js

```
// controller functions (disabled)
function goFirst()
{
   alert("in gofirst");
}
function goPrev()
{
   alert("in goprev");
}
function goNext()
{
   alert("in gonext");
}
function goLast()
{
   alert("in golast");
}

// event handler assignments
function init()
{
   if (document.getElementById)
   {
      oImageFirst = document.getElementById("first");
      oImagePrev = document.getElementById("prev");
      oImageNext = document.getElementById("next");
      oImageLast = document.getElementById("last");
   }
   else
   {
      oImageFirst = document.first;
      oImagePrev = document.prev;
      oImageNext = document.next;
      oImageLast = document.last;
   }
```

```
    addEvent(oImageFirst, 'click', goFirst);
    addEvent(oImagePrev, 'click', goPrev);
    addEvent(oImageNext, 'click', goNext);
    addEvent(oImageLast, 'click', goLast);
}

// initialize when the page has loaded
addEvent(window, "load", init);
```

The need to wrap the `li` element text for Internet Explorer (which the CSS shifts completely off-screen, because we don't need the text) forces scripters to address further considerations. In this application, a click of an `li` element is intended to run a local script, not to load an external URL. But the `a` element's default behavior is to load another URL. The # placeholder shown in Listing 14-3 causes the current page to reload, which will wipe away any activity of the `onclick` event handler function. It is necessary, therefore, to equip each of the slide-show navigation functions with some extra code lines that prevent the `a` element from executing its default behavior. You'll learn how to do that in Chapter 32, "Event Objects." (It requires different syntax for the incompatible W3C DOM and IE event models.)

One other note about the CSS approach in Listing 14-3 is that there is no image pre-caching taking place. You could add the pre-caching code for the "on" images from Listing 14-2 to get the alternative background images ready for the browser to swap. That's a case of CSS and JavaScript really working together.

The javascript: Pseudo-URL

You have seen instances in previous chapters of applying what is called the `javascript:` pseudo-URL to the `href` attributes of `<a>` and `<area>` tags. This technique should not be used for public web sites that may be accessed by browsers with scripting turned off (for which the links will be inactive).

The technique was implemented to supplement the `onclick` event handler of objects that act as hyperlinks. Especially in the early scripting days, when elements such as images had no event handlers of their own, hyperlinked elements surrounding those inactive elements allowed users to appear to interact directly with elements such as images. When the intended action was to invoke a script function (rather than navigate to another URL, as is usually the case with a hyperlink), the language designers invented the `javascript:` protocol for use in assignments to the `href` attributes of hyperlink elements (instead of leaving the required attribute empty).

When a scriptable browser encounters an `href` attribute pointing to a `javascript:` pseudo-URL, the browser executes the script content after the colon when the user clicks the element. For example, the `a` elements of Listing 14-3 could have been written to point to `javascript:` pseudo-URLs that invoke script functions on the page, such as:

```
<a href="javascript:goFirst()">
```

Note that the `javascript:` protocol is not a published standard, despite its wide adoption by browser makers. In a public web site that may be accessed by visitors with accessibility concerns (and potentially by browsers having little or no JavaScript capability), a link should point to a server

URL that performs an action (for example, through a server program), which in turn replicates what client-side JavaScript does (faster) for visitors with scriptable browsers.

Popular Dynamic HTML Techniques

Because today's scriptable browsers uniformly permit scripts to access each element of the document and automatically reflow the page's content when anything changes, a high degree of dynamism is possible in your applications. Dynamic HTML (DHTML) is a very deep subject, with lots of browser-specific peculiarities. In this section of the tutorial, you will learn techniques that work in Internet Explorer and W3C DOM-compatible browsers. We'll focus on two of the most common tasks for which DHTML is used: changing element styles and modifying document content.

Changing style sheet settings

Each element that renders on the page (and even some elements that don't) has a property called `style`. This property provides script access to all CSS properties supported for that element by the current browser. Property values are the same as those used for CSS specifications — frequently, a different syntax from similar settings that used to be made by HTML tag attributes. For example, if you want to set the text color of a `blockquote` element whose ID is `FranklinQuote`, the syntax is

```
document.getElementById("FranklinQuote").style.color = "rgb(255, 255, 0)";
```

Because the CSS `color` property accepts other ways of specifying colors (such as the traditional hexadecimal triplet — `#ffff00`), you may use those as well.

Some CSS property names, however, do not conform to JavaScript naming conventions. Several CSS property names contain hyphens. When that occurs, the scripted equivalent of the property compresses the words and capitalizes the start of each word. For example, the CSS property `font-weight` would be set in script as follows:

```
document.getElementById("highlight").style.fontWeight = "bold";
```

A related technique properly gives responsibility for the document design to CSS. For example, if you define CSS rules for two different classes, you can simply switch the class definition being applied to the element by way of the element object's `className` property. Let's say you define two CSS class definitions with different background colors:

```
.normal {background-color: #ffffff};
.highlighted {background-color: #ff0000};
```

In the HTML page, the element first receives its default class assignment as follows:

```
<p id="news" class="normal">...</p>
```

A script statement can then change the class of that element object so that the highlighted style applies to it:

```
document.getElementById("news").className = "highlighted";
```

Restoring the original class name also restores its look and feel. This approach is also a quick way to change multiple style properties of an element with a single statement.

Dynamic content via W3C DOM nodes

In Chapter 10, "Window and Document Objects," you saw the `document.createElement()` and `document.createTextNode()` methods in action. These methods create new document object model (DOM) objects out of thin air, which you may then modify by setting properties (attributes) prior to plugging the new stuff into the document tree for all to see.

As an introduction to this technique, we'll demonstrate the steps you would go through to add an element and its text to a placeholding `span` element on the page. In this example, a paragraph element belonging to a class called `centered` will be appended to a `span` whose `id` is `placeholder`. Some of the text for the content of the paragraph comes from a text field in a form (the visitor's first name). Here is the complete sequence:

```
var newElem = document.createElement("p");
newElem.className = "centered";
var newText = document.createTextNode("Thanks for visiting, " +
   document.forms[0].firstName.value);
// insert text node into new paragraph
newElem.appendChild(newText);
// insert completed paragraph into placeholder
document.getElementById("placeholder").appendChild(newElem);
```

After the element and text nodes are created, the text node must be inserted into the element node. Because the new element node is empty when it is created, the DOM `appendChild()` method plugs the text node into the element (between its start and end tags, if you could see the tags). When the paragraph element is assembled, it is inserted at the end of the initially empty `span` element. Additional W3C DOM methods (described in Chapter 26, "Generic HTML Element Objects," and Chapter 27, "Window and Frame Objects") provide more ways to insert, remove, and replace nodes.

Dynamic content through the innerHTML property

Prior to the W3C DOM specification, Microsoft invented a property of all element objects: `innerHTML`. This property first appeared in Internet Explorer 4, and it became popular due to its practicality. The property's value is a string containing HTML tags and other content, just as it would appear in an HTML document inside the current element's tags. Even though the W3C DOM working group did not implement this property for the published standard, the property proved to be too practical and popular for modern browser makers to ignore. You can find it implemented as a de facto standard in Mozilla-based browsers and Safari, among others.

To illustrate the difference in the approach, the following code example shows the same content creation and insertion as shown in the previous W3C DOM section, but this time with the `innerHTML` property:

```
// accumulate HTML as a string
var newHTML = "<p class='centered'>Thanks for visiting, ";
newHTML += document.forms[0].firstName.value;
newHTML += "</p>";
// blast into placeholder element's content
document.getElementById("placeholder").innerHTML = newHTML;
```

Although the `innerHTML` version seems more straightforward — and makes it easier for HTML coders to visualize what's being added — the DOM node approach is more efficient when the document modification task entails lots of content. Extensive JavaScript string concatenation operations can slow browser script processing. Sometimes, the shortest script is not necessarily the fastest.

And so ends the final lesson of the *JavaScript Bible* tutorial. If you have gone through every lesson and tried your hand at the exercises, you are ready to dive into the rest of the book to learn the fine details and many more features of both the DOM and the JavaScript language. You can work sequentially through the chapters of Part III, "JavaScript Core Language Reference," and Part IV, "Document Objects Reference," but before too long, you should also take a peek at Chapter 48, "Debugging Scripts," to learn some important debugging techniques.

Exercises

1. Explain the difference between a document `img` element object and the memory type of an image object.
2. Write the JavaScript statements needed to pre-cache an image file named `jane.jpg` that later will be used to replace the document image defined by the following HTML:

```
<img name="people" src="john.jpg" height="120" width="100" alt="people">
```

3. With the help of the code you wrote for question 2, write the JavaScript statement that replaces the document image with the memory image.
4. Backward-compatible `img` element objects do not have event handlers for mouse events. How do you trigger scripts needed to swap images for mouse rollovers?
5. Assume that a `table` element contains an empty table cell (`td`) element whose ID is `forwardLink`. Using W3C DOM node creation techniques, write the sequence of script statements that create and insert the following hyperlink into the table cell:

```
<a href="page4.html">Next Page</a>
```

Part III

JavaScript Core Language Reference

IN THIS PART

Chapter 15
The String Object

Chapter 16
The Math, Number, and Boolean Objects

Chapter 17
The Date Object

Chapter 18
The Array Object

Chapter 19
JSON — Native JavaScript Object Notation

Chapter 20
E4X — Native XML Processing

Chapter 21
Control Structures and Exception Handling

Chapter 22
JavaScript Operators

Chapter 23
Function Objects and Custom Objects

Chapter 24
Global Functions and Statements

CHAPTER 15

The String Object

IN THIS CHAPTER

How to parse and work with text

Performing search-and-replace operations

Scripted alternatives to text formatting

The tutorial in Chapter 8, "Programming Fundamentals, Part I," introduced you to the concepts of values and the types of values that JavaScript works with — features such as strings, numbers, and Boolean values. Chapter 12, "Strings, Math, and Dates," described several characteristics and methods of strings. In this chapter, you look more closely at the very important String data type, as well as its relationship to the Number data type. Along the way, you encounter the many ways in which JavaScript enables scripters to manipulate strings.

Note

Much of the syntax that you see in this chapter is identical to that of the Java programming language. Because the scope of JavaScript activity is much narrower than that of Java, you don't have nearly as much to learn for JavaScript as you do for Java. ■

String and Number Data Types

Although JavaScript is what is known as a "loosely typed" language, you still need to be aware of several data types, because of their impact on the way you work with the information in those forms. In this section, we focus on strings and two types of numbers.

Simple strings

A *string* consists of one or more standard text characters placed between matching quote marks. JavaScript is forgiving in one regard: You can use single or double quotes, as long as you match two single quotes or two double quotes around a string. A major benefit of this scheme becomes apparent when you try to include quoted text inside a string. For example, say that you're assembling a line of HTML code in a variable that you will eventually write to a new

window that is completely controlled by JavaScript. The line of text that you want to assign to a variable is the following:

```
<input type="checkbox" name="candy" />Chocolate
```

To assign this entire line of text to a variable, you have to surround the line in quotes. But because quotes appear inside the string, JavaScript (or any language) has problems deciphering where the string begins or ends. By carefully placing the other kind of quote pairs, however, you can make the assignment work. Here are two equally valid ways:

```
result = '<input type="checkbox" name="candy" />Chocolate';
result = "<input type='checkbox' name='candy' />Chocolate";
```

Notice that in both cases, the same unique pair of quotes surrounds the entire string. Inside the string, two quoted strings appear that are treated as such by JavaScript. It is helpful stylistically to settle on one form or the other, and then use that form consistently throughout your scripts.

If the expression you're quoting contains both apostrophes and double quotes, you need to *escape* one or the other within the string, with a backslash:

```
Pat's favorite word is "syzygy."
result = "Pat's favorite word is \"syzygy.\""
result = 'Pat\'s favorite word is "syzygy."'
```

Because escaped characters aren't as easy to read and proofread, you might want to choose a quote character that doesn't appear in the expression you're quoting, if possible.

Building long string variables

The act of joining strings together — concatenation — enables you to assemble long strings out of several little pieces. This feature is very important for some scripting — for example, when you need to build an HTML page's specifications entirely within a variable before writing the page to another frame, with one statement. It is often unwieldy and impractical to include such lengthy information in a single string on one line of code, which is why you may want to build the large string out of substrings.

One tactic keeps the length of each statement in this building process short enough so that it's easily readable in your text editor. This method uses the add-by-value assignment operator (+=) that appends the right-hand side of the equation to the left-hand side. Here is a simple example, which begins by initializing a variable, `newDocument`, as an empty string:

```
var newDocument = "";
newDocument += "<!DOCTYPE html>";
newDocument += "<html>" ;
newDocument += "<head>";
newDocument += '<meta http-equiv="content-type"';
newDocument += ' content="text/html;charset=utf-8">';
newDocument += "<title>Prodigal Summer</title>";
newDocument += "</head>";
newDocument += "<body>";
newDocument += "<h1>Prodigal Summer</h1>";
newDocument += '<p class="byline">by Barbara Kingsolver</p>';
```

Starting with the second line, each statement adds more data to the string being stored in `newDocument`. You can continue appending string data until the entire page's specification is contained in the `newDocument` variable.

Note

Excessive use of the add-by-value operator involving large quantities of text can become inefficient. If you are experiencing slow performance when accumulating large strings, try pushing your string segments into items of an array (see Chapter 18, "The Array Object"). Then use the array's `join()` method to generate the resulting large string value. ■

Joining string literals and variables

In some cases, you need to create a string out of literal strings (characters with quote marks around them) and string variable values. The methodology for concatenating these types of strings is no different from that of multiple string literals. The plus-sign operator does the job. Therefore, in the following example, a variable contains a name. That variable value is made a part of a larger string whose other parts are string literals:

```
teamName = prompt("Please enter your favorite team:","");
var msg = "The " + teamName + " are victorious!";
alert(msg);
```

Some common problems that you may encounter while attempting this kind of concatenation include the following:

- Accidentally omitting one of the quotes around a literal string
- Failing to insert blank spaces in the string literals to accommodate word spacing
- Forgetting to concatenate punctuation after a variable value

Also, don't forget that what we show here as variable values can be any expression that evaluates to a string, including property references and the results of some methods. For example:

```
var msg = "The name of this document is " + document.title + ".";
alert(msg);
```

Special inline characters

The way string literals are created in JavaScript makes adding certain characters to strings difficult — primarily quotes, carriage returns, apostrophes, and tab characters. Fortunately, JavaScript provides a mechanism for entering such characters into string literals. A backslash symbol, followed by the character that you want to appear as inline, makes that task happen. For the "invisible" characters, a special set of letters following the backslash tells JavaScript what to do.

The most common backslash pairs are as follows:

- `\"` Double quote
- `\'` Single quote (apostrophe)
- `\\` Backslash
- `\b` Backspace
- `\t` Tab

- \n New line
- \r Carriage return
- \f Form feed

Use these *inline characters* (also known as *escaped characters*, but this terminology has a different connotation for Internet strings) inside quoted string literals to make JavaScript recognize them. When assembling a block of text that needs a new paragraph, insert the \n character pair. Here are some examples of syntax using these special characters:

```
msg = "I say \"trunk\" and you say \"boot.\"";
msg = 'You\'re doing fine.';
msg = "This is the first line.\nThis is the second line.";
msg = document.title + "\n" + document.links.length + " links present.";
```

Technically speaking, a complete carriage return, as it was known in the days of the typewriter, is both a line feed (advance the line by one) and a carriage return (move the carriage all the way to the left margin). Although JavaScript strings treat a line feed (\n new line) as a full carriage return, you may have to construct \r\n breaks when assembling strings that go back to a cgi script on a server. The format that you use depends on the string-parsing capabilities of the cgi program. (Also see the special requirements for the textarea object in Chapter 36, "Text-Related Form Objects," on the CD.)

It's easy to confuse the strings assembled for display in textarea objects or alert boxes with strings to be written as HTML. In most cases, browsers ignore carriage returns or render them like spaces. For HTML strings, make sure that you use standard HTML tags, such as paragraphs (<p>...</p>) and line-breaks (<br>), rather than, or in addition to, the inline return or line feed symbols.

String Object

Properties	Methods
constructor	anchor()
length	big()
prototype†	blink()
	bold()
	charAt()
	charCodeAt()
	concat()
	fixed()
	fontcolor()
	fontsize()
	fromCharCode()†
	indexOf()

Properties	Methods
	`italics()`
	`lastIndexOf()`
	`link()`
	`localeCompare()`
	`match()`
	`replace()`
	`search()`
	`slice()`
	`small()`
	`split()`
	`strike()`
	`sub()`
	`substr()`
	`substring()`
	`sup()`
	`toLocaleLowerCase()`
	`toLocaleUpperCase()`
	`toLowerCase()`
	`toString()`
	`toUpperCase()`
	`valueOf()`

†Member of the static `String` object

Syntax

Creating a string object:

```
var myString = new String("characters");
```

Creating a string value:

```
var myString = "characters";
```

Accessing static `String` object properties and methods:

```
String.property | method([parameters])
```

Accessing string object properties and methods:

```
string.property | method([parameters])
```

Compatibility: WinIE3+, MacIE3+, NN2+, Moz+, Safari+, Opera+, Chrome+

About this object

JavaScript draws a fine line between a string value and a string object. Both let you use the same methods on their contents, so that by and large, you do not have to create a string object (with the `new String()` constructor) every time you want to assign a string value to a variable. A simple assignment operation (`var myString = "fred"`) is all you need to create a string value that behaves on the surface very much like a full-fledged string object.

Where the difference comes into play is when you want to exploit the "object-ness" of a genuine string object, which is explained further in the discussion of the `string.prototype` property, later in this chapter. You may also encounter the need to use a full-fledged string object when passing string data to Java applets. If you find that your applet doesn't receive a string value as a Java `String` data type, then create a new string object via the JavaScript constructor function before passing the value to the applet.

With string data often comes the need to massage string text in scripts. In addition to concatenating strings, at times you need to extract segments of strings, delete parts of strings, and replace one part of a string with some other text. Unlike many plain-language scripting languages, JavaScript is fairly low-level in its built-in facilities for string manipulation. This characteristic means that unless you can take advantage of the regular expression powers of IE4+/Moz1+ or advanced array techniques, you must fashion your own string handling routines out of the very elemental powers built into JavaScript. Later in this chapter, we provide several functions that you can use in your own scripts for common string handling in a manner fully compatible with older browsers.

As you work with string values, visualize every string value as an object, with properties and methods, like other JavaScript objects. JavaScript defines a few properties and a slew of methods for any string value (and one extra property for the static `String` object that is always present in the context of the browser window). The syntax is the same for string methods as it is for any other object method:

```
stringObject.method()
```

What may seem odd at first is that the *`stringObject`* part of this reference can be any expression that evaluates to a string, including string literals, variables containing strings, methods or functions that return strings, or other object properties. Therefore, the following examples of calling the `toUpperCase()` method are all valid:

```
"blah blah blah".toUpperCase()
yourName.toUpperCase() // yourName is a variable containing a string
window.prompt("Enter your name","").toUpperCase()
document.forms[0].entry.value.toUpperCase() // entry is text field object
```

A very important (and often misunderstood) concept to remember is that invoking a string method does not change the string object that is part of the reference. Rather, the method returns a string value, which can be used as a parameter to another method or function call, or assigned to a variable.

Therefore, to change the contents of a string variable to the results of a method, you must use an assignment operator, as in:

```
yourName = yourName.toUpperCase(); // variable is now all uppercase
```

Properties

constructor

Value: Function reference Read/Write

Compatibility: WinIE4+, MacIE4+, NN4+, Moz+, Safari+, Opera+, Chrome+

The `constructor` property is a reference to the function that was invoked to create the current string. For a native JavaScript string object, the constructor function is the built-in `String()` constructor.

When you use the `new String()` constructor to create a string object, the type of the value returned by the constructor is `object` (meaning that the `typeof` operator returns `object`). Therefore, you can use the `constructor` property on an object value to see if it is a string object:

```
if (typeof someValue == "object" )
{
   if (someValue.constructor == String)
   {
      // statements to deal with string object
   }
}
```

Although the property is read/write, and you can assign a different constructor to the `String.prototype`, the native behavior of a `String` object persists through the new constructor.

Example

Use The Evaluator (see Chapter 4) to test the value of the `constructor` property. One line at a time, enter and evaluate the following statements into the top text box:

```
a = new String("abcd")
a.constructor == String
a.constructor == Number
```

Related Items: `prototype` property

length

Value: Integer Read/Write

Compatibility: WinIE3+, MacIE3+, NN2+, Moz+, Safari+, Opera+, Chrome+

The most frequently used property of a string is `length`. To derive the length of a string, read its property as you would read the `length` property of any object:

```
string.length
```

The `length` value represents an integer count of the number of characters within the string. Spaces and punctuation symbols count as characters. Any backslash special characters embedded in a string count as one character, including such characters as newline and tab. Here are some examples:

```
"Lincoln".length // result = 7
"Four score".length // result = 10
"One\ntwo".length // result = 7
"".length // result = 0
```

The `length` property is commonly summoned when dealing with detailed string manipulation in repeat loops. For example, if you want to iterate through every character in a string, and somehow examine or modify each character, you would use the string's length as the basis for the loop counter.

prototype

Value: String object Read/Write

Compatibility: WinIE4+, MacIE4+, NN3+, Moz+, Safari+, Opera+, Chrome+

`String` objects defined with the `new String("`*`stringValue`*`")` constructor are robust objects compared to run-of-the-mill variables that are assigned string values. You certainly don't have to create this kind of string object for every string in your scripts, but these objects do come in handy if you find that strings in variables go awry. This happens occasionally while trying to preserve string information as script variables in other frames or windows. By using the `String` object constructor, you can be relatively assured that the string value will be available in the distant frame when needed.

Another benefit to using true `String` objects is that you can assign prototype properties and methods to all string objects in the document. A *prototype* is a collection of properties and methods that become part of every new object created after the prototype items are added. As an example, you may want to define a new method for transforming a string that isn't already defined by the JavaScript `String` object. Listing 15-1 shows how to create and use such a prototype.

Note

The property assignment event-handling technique employed throughout the code in this chapter, and much of the book, is `addEvent()`, a cross-browser event handler explained in detail in Chapter 32, "Event Object."

The `addEvent()` function is part of the script file `jsb-global.js`, located on the accompanying CD-ROM in the `Content/` folder, where it is accessible to all chapters' scripts. ■

LISTING 15-1

Using the String Object Prototype

HTML: jsb-15-01.html

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Using the String Object Prototype</title>
      <script type="text/javascript" src="../jsb-global.js"></script>
      <script type="text/javascript" src="jsb-15-01.js"></script>
   </head>
```

```
   <body>
      <h1>Using the String Object Prototype</h1>
      <form action="initialCap.php">
         <p>
            <label for="entry">Enter one or more words to be Initial Capped:</label>
         </p>
         <p>
            <input type="text" id="entry" name="entry" value="">
            <input type="button" id="do-it" value="Initial Caps">
         </p>
      </form>
   </body>
</html>
```

JavaScript: jsb-15-01.js

```
// initialize when the page has loaded
addEvent(window, 'load', initialize);

function initialize()
{
   // do this only if the browser can handle DOM methods
   if (document.getElementById)
   {
      // point to critical elements
      var oButton = document.getElementById('do-it');
      var oEntry = document.getElementById('entry');

         // if they all exist...
         if (oButton && oEntry)
         {
            // apply behavior to button
            addEvent(oButton, 'click', doIt);
         }
   }
}

// try out the new String method
function doIt()
{
   // point to input field
   var oEntry = document.getElementById('entry');

   // use the new method to update the input text
   oEntry.value = oEntry.value.initialCaps();
}


// add a new method to the String prototype
// to capitalize the first letter of each word

String.prototype.initialCaps = function()
```

continued

LISTING 15-1 *(continued)*

```
{
   // get the input text
   var sText = this.toString();

   // separate individual words
   var aWords = sText.split(' ');

   // capitalize each word
   for (var i = 0; i < aWords.length; i++)
   {
      var sInitial = aWords[i].charAt(0);
      var sRemainder = aWords[i].substr(1);
      aWords[i] = sInitial.toUpperCase() + sRemainder.toLowerCase();
   }

   // reassemble the array of words back into a string
   return aWords.join(' ');
}
```

Note

Many of the HTML example pages in this book include a form action to post input to a server-side PHP program — as a reminder that it's important to make sure that our pages can function even when JavaScript is not running. The server-side scripts themselves do not exist and are outside the scope of this book. If written, they would perform essentially the same operations as the JavaScript logic does. ■

The core of the JavaScript in Listing 15-1 is the last block, where we add the `initialCaps()` method to the `String` object prototype, and spell out what that method does. Similar to most object methods, this new one doesn't modify the original string itself, but rather returns a value we can use for any purpose we want. Here, we grab the contents of the input field `entry`, transform it with `initialCaps()`, and use that new value to reset the input field contents.

Modifying object prototypes is a powerful way to add new features to the JavaScript language on-the-fly.

In the next sections, we divide `String` object methods into two distinct categories. The first, parsing methods, focuses on string analysis and character manipulation within strings. The second group, formatting methods, is devoted entirely to assembling strings in HTML syntax for those scripts that assemble the text to be written into new documents or other frames.

Parsing methods

string.charAt(*index*)

Returns: One-character string

Compatibility: WinIE3+, MacIE3+, NN2+, Moz+, Safari+, Opera+, Chrome+

Use the *string*.charAt() method to read a single character from a string when you know the position of that character. For this method, you specify an index value in the string as a parameter to the method. The index value of the first character of the string is 0. To grab the last character of a string, mix string methods:

```
myString.charAt(myString.length - 1)
```

If your script needs to get a range of characters, use the *string*.substring() method. Using *string*.substring() to extract a character from inside a string is a common mistake — the *string*.charAt() method is more efficient.

Example

Enter each of the following statements into the top text box of The Evaluator:

```
a = "banana daiquiri"
a.charAt(0)      // result = 'b'
a.charAt(5)      // result = 'a' (the third 'a')
a.charAt(6)      // result = ' ' (space)
a.charAt(20)     // result = '' (empty string)
```

Related Items: *string*.lastIndexOf(), *string*.indexOf(), *string*.substring() methods

string.charCodeAt([*index*])
string.fromCharCode(*num1* [, *num2* [, ... *numn*]])

Returns: Integer code number for a character; concatenated string value of code numbers supplied as parameters

Compatibility: WinIE4+, MacIE4+, NN4+, Moz+, Safari+, Opera+, Chrome+

Conversions from plain language characters to their numeric equivalents have a long tradition in computer programming. For many years, the most common numbering scheme was the ASCII standard, which covers the basic English alphanumeric characters and punctuation within 128 values (numbered 0 through 127). An extended version with a total of 256 characters, with some variations, depending on the operating system, accounts for other roman characters in other languages, particularly accented vowels. To support all languages, including pictographic languages, and other non-Roman writing systems, a world standard called Unicode provides space for thousands of characters. All modern browsers work with the Unicode system.

In JavaScript, character conversions are handled by string methods. The two methods that perform character conversions work in very different ways syntactically. The first, *string*.charCodeAt(), converts a single string character to its numeric equivalent. The string being converted is the one to the left of the method name — and the string may be a literal string or any other expression that evaluates to a string value. If no parameter is passed, the character being converted is by default the first character of the string. However, you can also specify a different character as an index value into the string (first character is 0), as demonstrated here:

```
"abc".charCodeAt()  // result = 97
"abc".charCodeAt(0)  // result = 97
"abc".charCodeAt(1)  // result = 98
```

If the string value is an empty string, or the index value is beyond the last character, the result is `NaN`.

To convert numeric values to their characters, use the `String.fromCharCode()` method. Notice that the object beginning the method call is the static `String` object, not a string value. Then, as parameters, you can include one or more integers, separated by commas. In the conversion process, the method combines the characters for all of the parameters into one string, an example of which is shown here:

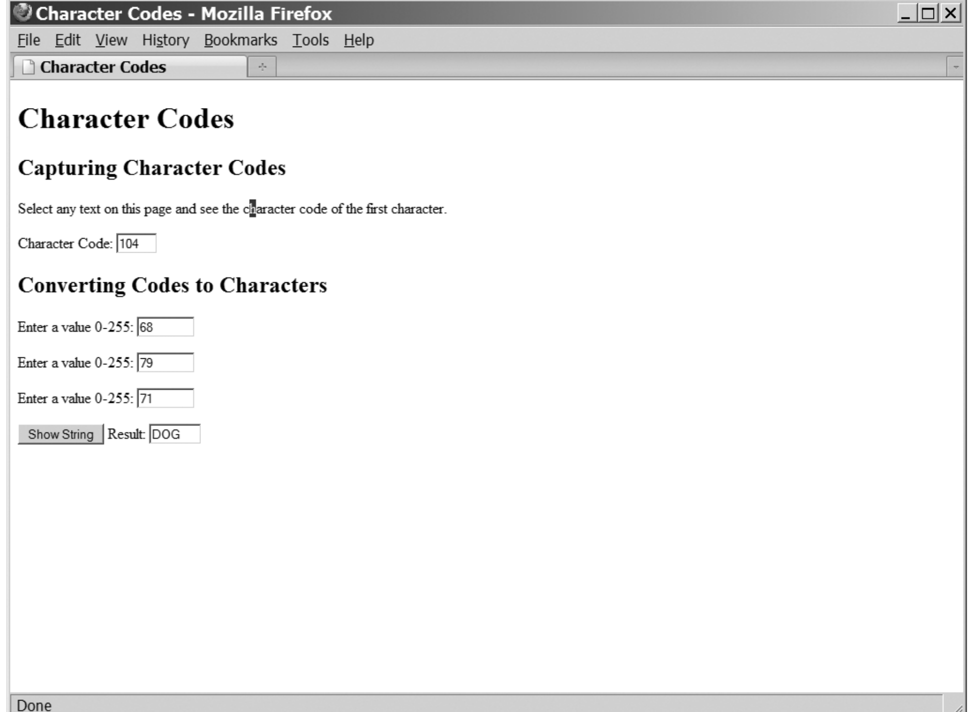
```
String.fromCharCode(97, 98, 99)  // result "abc"
```

Note

Although most modern browsers support character values across the entire Unicode range, the browser won't properly render characters above 255 unless the computer is equipped with language and font support for the designated language. To assist with Unicode character display, all our HTML documents begin with declarations of UTF-8 character encoding. ■

FIGURE 15-1

Conversions from text characters to ASCII values and vice versa.



Example

Listing 15-2 provides examples of both methods on one page. Moreover, because one of the demonstrations relies on the automatic capture of selected text on the page, the scripts include code to accommodate the different handling of selection events, and the capture of the selected text in a variety of browsers.

After you load the page, select part of the body text anywhere on the page. If you start the selection with the lowercase letter "a," the character code displays as 97.

Try entering numeric values in the three fields at the bottom of the page. Values below 32 are ASCII control characters that most fonts represent as blanks or hollow squares. But try other values to see what you get. Notice that the script passes all three values as a group to the `String.fromCharCode()` method, and the result is a combined string. Thus, Figure 15-1 shows what happens when you enter the uppercase ASCII values for a three-letter animal name.

LISTING 15-2

Character Conversions

HTML: jsb-15-02.html

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Character Codes</title>
      <script type="text/javascript" src="../jsb-global.js"></script>
      <script type="text/javascript" src="jsb-15-02.js"></script>
   </head>
   <body>
      <h1>Character Codes</h1>

      <form action="show-string.php">
         <h2>Capturing Character Codes</h2>
         <p>Select any text on this page and see the character code of the first
         character.</p>
         <p>
            <label for="display">Character Code:</label>
            <input type="text" id="display" name="display" size="3" />
         </p>

         <h2>Converting Codes to Characters</h2>
         <p>
            <label for="entry1">Enter a value 0-255:</label>
            <input type="text" id="entry1" name="entry1" size="6" />
         </p>
         <p>
            <label for="entry2">Enter a value 0-255:</label>
            <input type="text" id="entry2" name="entry2" size="6" />
         </p>
         <p>
```

continued

LISTING 15-2 *(continued)*

```
            <label for="entry3">Enter a value 0-255:</label>
            <input type="text" id="entry3" name="entry3" size="6" />
         </p>
         <p>
            <input type="button" id="showstr" value="Show String" />
            <label for="result">Result:</label>
            <input type="text" id="result" name="result" size="5" />
         </p>
      </form>
   </body>
</html>
```

JavaScript: jsb-15-02.js

```
// initialize when the page has loaded
addEvent(window, 'load', initialize);

// show character code when text is selected
addEvent(document, 'mouseup', showCharCode);


function initialize()
{
   // do this only if the browser can handle DOM methods
   if (document.getElementById)
   {
      // point to critical element
      var oButton = document.getElementById('showstr');

         // if it exists...
         if (oButton)
         {
            // apply behavior to button
            addEvent(oButton, 'click', showString);
         }
   }
}


// show the text characters represented by the numbers entered
function showString()
{
   var oEntry1 = document.getElementById('entry1');
   var oEntry2 = document.getElementById('entry2');
   var oEntry3 = document.getElementById('entry3');
   var oResult = document.getElementById('result');

   oResult.value = String.fromCharCode(oEntry1.value, oEntry2.value, oEntry3.value);
}
```

```
// display the character code of the selected text
// (first character only)
function showCharCode()
{
   var theText = "";
   var oDisplay = document.getElementById('display');

      // get the selected text using various browsers' methods
      if (window.getSelection)
      {
         theText = window.getSelection().toString();
      }
      else if (document.getSelection)
      {
         theText = document.getSelection();
      }
      else if (document.selection && document.selection.createRange)
      {
         theText = document.selection.createRange().text;
      }

      // display the result if any
      if (theText)
      {
         oDisplay.value = theText.charCodeAt();
      }
      else
      {
         oDisplay.value = " ";
      }
}
```

string.concat(*string2*)

Returns: Combined string

Compatibility: WinIE4+, MacIE4+, NN4+, Moz+, Safari+, Opera+, Chrome+

JavaScript's add-by-value operator (+=) provides a convenient way to concatenate strings. Most browsers, however, include a string object method that performs the same task. The base string to which more text is appended is the object or value to the left of the period. The string to be appended is the parameter of the method, as the following example demonstrates:

```
"abc".concat("def")  // result: "abcdef"
```

As with the add-by-value operator, the `concat()` method doesn't know about word spacing. You are responsible for including the necessary space between words if the two strings require a space between them in the result.

Related Items: Add-by-value (+=) operator

string.indexOf(*searchString* [, *startIndex*])

Returns: Index value of the character within *string* where *searchString* begins

Compatibility: WinIE3+, MacIE3+, NN2+, Moz+, Safari+, Opera+, Chrome+

Like some programming languages' offset string function, JavaScript's `indexOf()` method enables your script to obtain the position in the main string where a search string begins. Optionally, you can specify where in the main string the search should begin — but the returned value is always relative to the very first character of the main string. Like all string object methods, index values start their count with `0`. If no match occurs within the main string, the returned value is `-1`. Thus, this method is also a convenient way to determine whether one string contains another, regardless of position.

Example

Enter each of the following statements (up to but not including the "//" comment symbols) into the top text box of The Evaluator (you can simply replace the parameters of the `indexOf()` method for each statement after the first one). Compare your results with the results shown below.

```
a = "bananas"
a.indexOf("b")      // result = 0 (index of 1st letter is zero)
a.indexOf("a")      // result = 1
a.indexOf("a",1)    // result = 1 (start from 2nd letter)
a.indexOf("a",2)    // result = 3 (start from 3rd letter)
a.indexOf("a",4)    // result = 5 (start from 5th letter)
a.indexOf("nan")    // result = 2
a.indexOf("nas")    // result = 4
a.indexOf("s")      // result = 6
a.indexOf("z")      // result = -1 (no "z" in string)
```

Related Items: *string*.`lastIndexOf()`, *string*.`charAt()`, *string*.`substring()` methods

string.lastIndexOf(*searchString*[, *startIndex*])

Returns: Index value of the last character within the *string* where *searchString* begins

Compatibility: WinIE3+, MacIE3+, NN2+, Moz+, Safari+, Opera+, Chrome+

The *string*.`lastIndexOf()` method is closely related to the method *string*.`indexOf()`. The only difference is that this method starts its search for a match from the end of the string (*string*.`length - 1`) and works its way backward through the string. All index values are still counted, starting with 0, from the front of the string. The examples that follow use the same values as in the examples for *string*.`indexOf()` so that you can compare the results. In cases where only one instance of the search string is found, the results are the same; but when multiple instances of the search string exist, the results can vary widely — hence the need for this method.

Example

Enter each of the following statements (up to, but not including the "//" comment symbols) into the top text box of The Evaluator (you can simply replace the parameters of the `lastIndexOf()` method for each statement after the first one). Compare your results with the results shown below.

```
a = "bananas"
a.lastIndexOf("b")    // result = 0 (index of 1st letter is zero)
a.lastIndexOf("a")    // result = 5
a.lastIndexOf("a",1)  // result = 1 (from 2nd letter toward the front)
a.lastIndexOf("a",2)  // result = 1 (start from 3rd letter working
```

```
                                        toward front)
a.lastIndexOf("a",4)  // result = 3 (start from 5th letter)
a.lastIndexOf("nan")  // result = 2 [except for -1 Nav 2.0 bug]
a.lastIndexOf("nas")  // result = 4
a.lastIndexOf("s")    // result = 6
a.lastIndexOf("z")    // result = -1 (no "z" in string)
```

Related Items: *string*.lastIndexOf(), *string*.charAt(), *string*.substring() methods

string.localeCompare(*string2*)

Returns: Integer

Compatibility: WinIE5.5+, MacIE-, NN6+, Moz+, Safari+, Opera+, Chrome+

The localeCompare() method lets a script compare the cumulative Unicode values of two strings, taking into account the language system for the browser. This method is necessitated by only a few human languages (Turkish is said to be one). If the two strings, adjusted for the language system, are equal, the value returned is zero. If the string value on which the method is invoked (meaning the string to the left of the period) sorts ahead of the parameter string, the value returned is a negative integer; otherwise the returned value is a positive integer.

The ECMA standard for this method leaves the precise positive or negative values up to the browser designer. NN6+ calculates the cumulative Unicode values for both strings and subtracts the string parameter's sum from the string value's sum. IE5.5+ and FF1+, on the other hand, return -1 or 1 if the strings are not colloquially equal.

Related Items: *string*.toLocaleLowerCase(), *string*.toLocaleUpperCase() methods

string.match(*regExpression*)

Returns: Array of matching substrings

Compatibility: WinIE4+, MacIE4+, NN4+, Moz+, Safari+, Opera+, Chrome+

The *string*.match() method relies on the RegExp (regular expression) object to carry out a match within a string. The string value under scrutiny is to the left of the dot, whereas the regular expression to be used by the method is passed as a parameter. The parameter must be a regular expression object, created according to the two ways these objects can be generated.

This method returns an array value when at least one match turns up; otherwise the returned value is null. Each entry in the array is a copy of the string segment that matches the specifications of the regular expression. You can use this method to uncover how many times a substring or sequence of characters appears in a larger string. Finding the offset locations of the matches requires other string parsing.

Example

To help you understand the *string*.match() method, Listing 15-3 provides you with a regular expression workshop. Three input controls let you set up a test: the first is for the long string, to be examined by the method; the second is for a regular expression; and the third is a checkbox that lets you specify whether the search through the string for matches should be case-sensitive. Some default values are provided, in case you're not yet familiar with the syntax of regular expressions (see Chapter 45 on the CD-ROM). After you click the "Execute match()" button, the script creates a regular expression object out of your input, performs the *string*.match() method on the big string, and

reports two kinds of results to the page. The primary result is a string version of the array returned by the method; the other is a count of items returned.

LISTING 15-3

Regular Expression Match Workshop

HTML: jsb-15-03.html

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Regular Expression Match Workshop</title>
      <script type="text/javascript" src="../jsb-global.js"></script>
      <script type="text/javascript" src="jsb-15-03.js"></script>
   </head>
   <body>
      <h1>Regular Expression Match Workshop</h1>

      <form action="regexp-match.php">
         <p>
           <label for="string">Enter a main string:</label>
           <input type="text" id="string" name="string" size="60"
           value="Many a maN and womAN have meant to visit GerMAny." />
         </p>
         <p>
           <label for="pattern">Enter a regular expression to match:</label>
           <input type="text" id="pattern" name="pattern" size="25" value="\wa\w" />

           <input type="checkbox" id="caseSens" name="caseSens" />
           <label for="caseSens">Case-sensitive</label>
         </p>
         <p>
           <input type="button" id="button" value="Execute match()" />
           <input type="reset" />
         </p>
         <p>
           <label for="result">Result:</label>
           <input type="text" id="result" name="result" size="40" />
         </p>
         <p>
           <label for="count">Count:</label>
           <input type="text" id="count" name="count" size="3" />
         </p>
      </form>
   </body>
</html>
```

JavaScript: jsb-15-03.js

```
// initialize when the page has loaded
addEvent(window, 'load', initialize);
```

```
function initialize()
{
   // do this only if the browser can handle DOM methods
   if (document.getElementById)
   {
      // point to critical element
      var oButton = document.getElementById('button');

         // if it exists...
         if (oButton)
         {
            // apply behavior to button
            addEvent(oButton, 'click', doMatch);
         }
   }
}

// perform regular expression match using input values
function doMatch()
{
   // gather input values
   var oString = document.getElementById('string');
   var sString = oString.value;

   var oPattern = document.getElementById('pattern');
   var oCaseSensitivity = document.getElementById('caseSens');
   var sFlags = 'g';   // g = find all
      if (!oCaseSensitivity.checked) sFlags += 'i';   // i = case-insensitive

   // create regular expression object
   var oRegExp = new RegExp(oPattern.value, sFlags);

   // perform match
   var aResult = sString.match(oRegExp);

   // display results
   var oResult = document.getElementById('result');
   var oCount = document.getElementById('count');

      // got matches?
      if (aResult)
      {
         oResult.value = aResult.toString();
         oCount.value = aResult.length;
      }
      else
      {
         oResult.value = "<no matches>";
         oCount.value = "";
      }
}
```

The default value for the main string has unusual capitalization intentionally. The capitalization lets you see more clearly where some of the matches come from. For example, the default regular expression looks for any three-character string that has the letter "a" in the middle. Six string segments match that expression. With the help of capitalization, you can see where each of the four strings containing "man" is extracted from the main string. The following table lists some other regular expressions to try with the default main string.

RegExp	Description
`man`	Both case-sensitive and not
`man\b`	Where "man" is at the end of a word
`\bman`	Where "man" is at the start of a word
`me*an`	Where zero or more "e" letters occur between "m" and "a"
`.a.`	Where "a" is surrounded by any one character (including space)
`\sa\s`	Where "a" is surrounded by a space on both sides
`z`	Where a "z" occurs (none in the default string)

In the scripts for Listing 15-3, if the *`string`*`.match()` method returns `null`, you are informed politely, and the count field is emptied.

Related Items: `RegExp` object (see Chapter 45 on the CD-ROM)

string.replace(*regExpression*, *replaceString*)

Returns: Changed string

Compatibility: WinIE4+, MacIE4+, NN4+, Moz+, Safari+, Opera+, Chrome+

Regular expressions are commonly used to perform search-and-replace operations. In conjunction with the *`string`*`.search()` method, JavaScript's *`string`*`.replace()` method provides a simple framework in which to perform this kind of operation on any string.

A replace operation using RegExp requires three components. The first is the main string that is the target of the operation. Second is the regular expression pattern to search for. And third is the string to replace each instance of the text found by the operation.

```
stringToSearch.replace(pattern, replacementString);
```

For the *`string`*`.replace()` method, the main string is the string value, or object referenced, to the left of the period. This string can also be a literal string (that is, text surrounded by quotes). The regular expression to search for is the first parameter, whereas the replacement string is the second parameter.

The regular expression definition determines whether the replacement is of just the first match encountered in the main string, or all matches in the string. If you add the `g` flag to the end of the regular expression, then one invocation of the `replace()` method performs a global search-and-replace throughout the entire main string.

As long as you know how to generate a regular expression, you can easily use the *`string`*`.replace()` method to perform simple replacement operations. But using regular expressions can make the operation more powerful. Consider these soliloquy lines by Hamlet:

```
To be, or not to be: that is the question:
Whether 'tis nobler in the mind to suffer
```

If you wanted to replace both instances of "be" with "exist," you can do it in this case by specifying

```
var regexp = /be/g;
soliloquy.replace(regexp, "exist");
```

But you can't always be assured that the letters "b" and "e" will be standing alone as a word. What happens if the main string contains the word "being" or "saber"? The above example would replace the "be" letters in them as well.

Rescue comes from special characters that better define what to search for. In the example here, the search is for the word "be." Therefore, the regular expression surrounds the search text with word boundaries (the `\b` special character), as in

```
var regexp = /\bbe\b/g;
soliloquy.replace(regexp, "exist");
```

This syntax also takes care of the fact that the first two "be" words are followed by punctuation, rather than a space, as you may expect for a freestanding word. For more about regular expression syntax, see Chapter 45 on the CD-ROM.

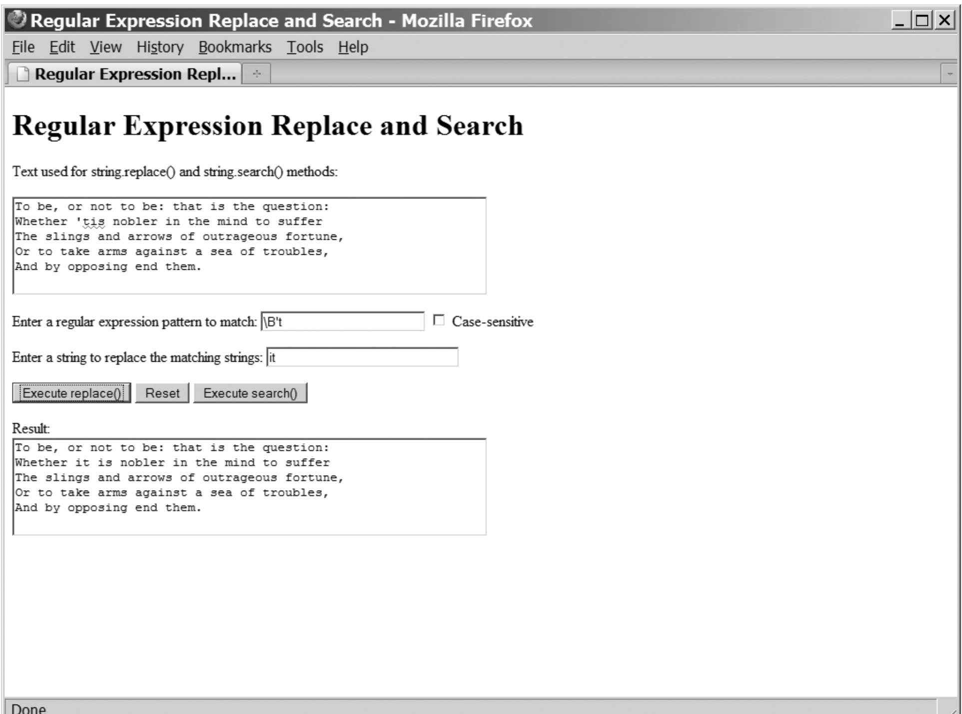
Example

The page created by Listing 15-4 lets you practice with the *`string`*`.replace()` and *`string`*`.search()` methods, and regular expressions, in a friendly environment. The source text is a five-line excerpt from *Hamlet*. You can enter the regular expression to search on, and the replacement text as well. Note that the script completes the job of creating the regular expression object, so that you can focus on the other special characters used to define the matching string. All replacement activities act globally because the `g` flag is automatically appended to any expression you enter.

Default values in the fields replace the contraction 'tis with "it is" after you click the "Execute replace()" button (see Figure 15-2). Notice that the backslash character in front of the apostrophe of 'tis (in the string assembled in `mainString`) makes the apostrophe a non-word boundary, and thus allows the `\B't` regular expression to find a match there. As described in the section on the *`string`*`.search()` method, the button connected to that method returns the offset character number of the matching string (or `-1` if no match occurs).

FIGURE 15-2

Using the default replacement regular expression.



LISTING 15-4

Lab for *string*.replace() and *string*.search()

```
HTML: jsb-15-04.html

<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Regular Expression Replace and Search</title>
      <script type="text/javascript" src="../jsb-global.js"></script>
      <script type="text/javascript" src="jsb-15-04.js"></script>
   </head>
   <body>
      <h1>Regular Expression Replace and Search</h1>

      <form action="regexp-replace.php">
         <p>
            <label for="source">Text used for string.replace() and
```

```
                         string.search() methods:</label>
         </p>
         <p>
            <textarea id="source" name="source" cols="60" rows="5">
To be, or not to be: that is the question:
Whether 'tis nobler in the mind to suffer
The slings and arrows of outrageous fortune,
Or to take arms against a sea of troubles,
And by opposing end them.
</textarea>
         </p>
         <p>
           <label for="pattern">Enter a regular expression pattern to match:</label>
           <input type="text" id="pattern" name="pattern" size="25" value="\B't" />

           <input type="checkbox" id="caseSens" name="caseSens" />
           <label for="caseSens">Case-sensitive</label>
         </p>
         <p>
           <label for="replacement">Enter a string to replace
                         the matching strings:</label>
           <input type="text" id="replacement" name="replacement"
                          size="30" value="it " />
         </p>
         <p>
           <input type="button" id="button-replace" value="Execute replace()"
                          onclick="doReplace(this.form)" />
           <input type="reset" />
           <input type="button" id="button-search" value="Execute search()"
                          onclick="doSearch(this.form)" />
         </p>
         <p>
           <label for="result">Result:</label><br />
           <textarea id="result" name="result" cols="60" rows="5"></textarea>
         </p>
      </form>
   </body>
</html>
```

JavaScript: jsb-15-04.js

```
// initialize when the page has loaded
addEvent(window, 'load', initialize);

function initialize()
{
   // do this only if the browser can handle DOM methods
   if (document.getElementById)
   {
      // check for critical elements
      var oSource = document.getElementById('source');
      var oPattern = document.getElementById('pattern');
```

continued

LISTING 15-4 *(continued)*

```
      var oCaseSensitivity = document.getElementById('caseSens');
      var oReplacement = document.getElementById('replacement');
      var oButtonReplace = document.getElementById('button-replace');
      var oButtonSearch = document.getElementById('button-search');
      var oResult = document.getElementById('result');

         // if they all exist...
         if (oSource && oPattern && oCaseSensitivity && oReplacement
            && oButtonReplace && oButtonSearch && oResult)
         {
            // apply behaviors
            addEvent(oButtonReplace, 'click', doReplace);
            addEvent(oButtonSearch, 'click', doSearch);
         }
         else
         {
            alert('Critical element not found');
         }
   }
}



function doReplace()
{
   var sSource = document.getElementById('source').value;
   var sReplacement = document.getElementById('replacement').value;
   var sPattern = document.getElementById('pattern').value;
   var sFlags = 'g';   // g = find all; i = case-insensitive
      if (!document.getElementById('caseSens').checked) sFlags += 'i';

   // create regular expression object
   var oRegExp = new RegExp(sPattern, sFlags);

   // perform replace
   var sResult = sSource.replace(oRegExp, sReplacement);

   // display results
   document.getElementById('result').value = sResult;
}


function doSearch()
{
   var sSource = document.getElementById('source').value;
   var sPattern = document.getElementById('pattern').value;
   var sFlags = 'g';   // g = find all; i = case-insensitive
      if (!document.getElementById('caseSens').checked) sFlags += 'i';

   // create regular expression object
```

```
   var oRegExp = new RegExp(sPattern, sFlags);

   // perform replace
   var sResult = sSource.search(oRegExp);

   // display results
   document.getElementById('result').value = sResult;
}
```

Related Items: *string*.match() method; RegExp object

string.search(*regExpression*)

Returns: Offset integer

Compatibility: WinIE4+, MacIE4+, NN4+, Moz+, Safari+, Opera+, Chrome+

The results of the *string*.search() method may remind you of the *string*.indexOf() method. In both cases, the returned value is the character number where the matching string first appears in the main string, or -1 if no match occurs. The big difference, of course, is that the matching string for *string*.search() is a regular expression.

Example

Listing 15-4, which illustrates the string.replace() method, also provides a laboratory to experiment with the *string*.search() method.

Related Items: *string*.match() method; RegExp object

string.slice(*startIndex* [, *endIndex*])

Returns: String

Compatibility: WinIE4+, MacIE4+, NN4+, Moz+, Safari+, Opera+, Chrome+

The *string*.slice() method resembles the method *string*.substring(), in that both let you extract a portion of one string and create a new string as a result (without modifying the original string). A helpful improvement in *string*.slice(), however, is that specifying an ending index value relative to the end of the main string is easier.

Using *string*.substring() to extract a substring that ends before the end of the string requires machinations such as the following:

```
string.substring(4, (string.length-2))
```

Instead, you can assign a negative number to the second parameter of *string*.slice() to indicate an offset from the end of the string:

```
string.slice(4, -2)
```

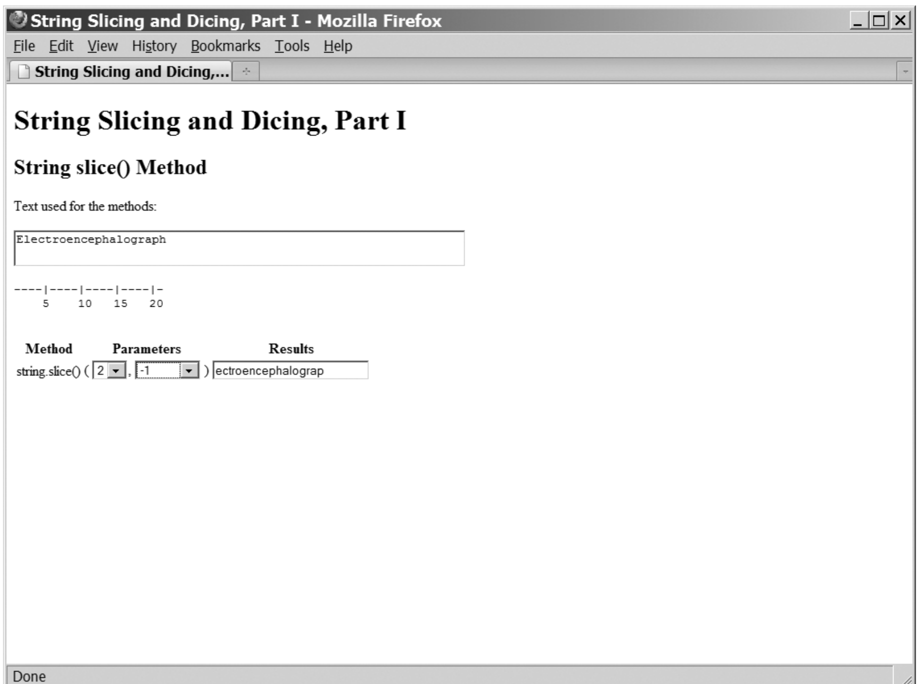
The second parameter is optional. If you omit the second parameter, the returned value is a string from the starting offset to the end of the main string.

Example

With Listing 15-5, you can try several combinations of parameters with the `string.slice()` method (see Figure 15-3). A base string is provided (along with character measurements). Select from the different choices available for parameters, and study the outcome of the slice.

FIGURE 15-3

Lab for exploring the *string*.slice() method.



LISTING 15-5

Slicing a String

HTML: jsb-15-05.html

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>String Slicing and Dicing, Part I</title>
      <script type="text/javascript" src="../jsb-global.js"></script>
      <script type="text/javascript" src="jsb-15-05.js"></script>
   </head>
   <body>
   <h1>String Slicing and Dicing, Part I</h1>
```

```
      <b>String slice() Method</b>

      <form action="string-slice.php">
         <p>
            <label for="source">Text used for the methods:</label>
         </p>
         <p>
            <textarea id="source" name="source" cols="60" rows="1">
Electroencephalograph</textarea>
            <pre>
----|----|----|----|-
    5    10   15   20
            </pre>
         </p>

         <table>
            <thead>
               <tr>
                  <th>Method</th>
                  <th>Parameters</th>
                  <th>Results</th>
               </tr>
            </thead>
            <tbody>
               <tr>
                  <td>string.slice()</td>
                  <td>(
                     <select id="parameter1" name="parameter1">
                        <option value="0">0</option>
                        <option value="1">1</option>
                        <option value="2">2</option>
                        <option value="3">3</option>
                        <option value="5">5</option>
                     </select>,
                     <select id="parameter2" name="parameter2">
                        <option>(None)</option>
                        <option value="5">5</option>
                        <option value="10">10</option>
                        <option value="-1">-1</option>
                        <option value="-5">-5</option>
                        <option value="-10">-10</option>
                     </select>
                  )</td>
                  <td>
                     <input type="text" id="result" name="result" size="25" />
                  </td>
               </tr>
            </tbody>
         </table>
      </form>
   </body>
</html>
```

continued

LISTING 15-5 *(continued)*

JavaScript: jsb-15-05.js

```
// initialize when the page has loaded
addEvent(window, 'load', initialize);

function initialize()
{
   // do this only if the browser can handle DOM methods
   if (document.getElementById)
   {
      // check for critical elements
      var oSource = document.getElementById('source');
      var oParameter1 = document.getElementById('parameter1');
      var oParameter2 = document.getElementById('parameter2');
      var oResult = document.getElementById('result');

         // if they all exist...
         if (oSource && oParameter1 && oParameter2 && oResult)
         {
            // apply behaviors
            addEvent(oParameter1, 'change', showResults);
            addEvent(oParameter2, 'change', showResults);
         }
         else
         {
            alert('Critical element not found');
         }
   }
}


// execute the method and display the result
function showResults()
{
   // get source text
   var oSource = document.getElementById('source');
   var sSource = oSource.firstChild.nodeValue;

   // get paramaters
   var oParameter1 = document.getElementById('parameter1');
   var iParameter1 = parseInt(oParameter1.options[oParameter1.selectedIndex].value);

   var oParameter2 = document.getElementById('parameter2');
   var iParameter2 = parseInt(oParameter2.options[oParameter2.selectedIndex].value);

   // generate result & display it
   var oResult = document.getElementById('result');

      if (!iParameter2)
      {
```

```
      oResult.value = sSource.slice(iParameter1);
   }
   else
   {
      oResult.value = sSource.slice(iParameter1, iParameter2);
   }
}
```

Related Items: *string*.substr(), *string*.substring() methods

string.split("*delimiterCharacter*" [, *limitInteger*])

Returns: Array of delimited items

Compatibility: WinIE4+, MacIE4+, NN3+, Moz+, Safari+, Opera+, Chrome+

The split() method is the functional opposite of the array.join() method (see Chapter 18, “The Array Object”). From the string object point of view, JavaScript splits a long string into pieces delimited by a specific character, and then creates a dense array with those pieces. You do not need to initialize the array via the new Array() constructor. Given the powers of array object methods, such as array.sort(), you may want to convert a series of string items to an array to take advantage of those powers. Also, if your goal is to divide a string into an array of single characters, you can still use the split() method, but specify an empty string as a parameter. For some older browsers, such as NN3 and IE4, only the first parameter is observed.

In modern browsers, you can use a regular expression object for the first parameter, enhancing your power to find delimiters in strings. For example, consider the following string:

```
var nameList = "Fred - 123-4567,Jane - 234-5678,Steve - 345-6789";
```

To convert that string into an array of only the names takes a lot of parsing without regular expressions before you can even use *string*.split(). However, with a regular expression as a parameter,

```
var regexp = / - [\d-]+,?\b/;
var newArray = nameList.split(regexp);
    // result = an array "Fred", "Jane", "Steve", ""
```

the new array entries hold only the names, and not the phone numbers or punctuation.

A second optional parameter, an integer value, allows you to specify a limit to the number of array elements generated by the method.

Example

Use The Evaluator (Chapter 4) to see how the string.split() method works. Begin by assigning a comma-delimited string to a variable:

```
a = "Anderson,Smith,Johnson,Washington"
```

Now split the string at comma positions so that the string pieces become items in an array, saved as `b`:

```
b = a.split(",")
```

To prove that the array contains four items, inspect the array's `length` property:

```
b.length    // result: 4
```

Related Items: `array.join()` method

string.substr(*start* [, *length*])

Returns: String

Compatibility: WinIE4+, MacIE4+, NN4+, Moz+, Safari+, Opera+, Chrome+

The *string*.`substr()` method offers a variation of the *string*.`substring()` method that has been in the JavaScript language since the beginning. The distinction is that the *string*.`substr()` method's parameters specify the starting index and a number of characters to be included from that start point. In contrast, the `string.substring()` method parameters specify index points for the start and end characters within the main string.

As with all string methods requiring an index value, the *string*.`substr()` first parameter is zero-based. If you do not specify a second parameter, the returned substring starts at the indexed point and extends to the end of the string. A second parameter value that exceeds the end point of the string means that the method returns a substring to the end of the string.

Even though this method is newer than its partner, it is not part of the latest (Fifth) edition of the ECMA standard for the language. But because the method is so widely used, the standard does acknowledge it as an annex, so that other scripting contexts can implement the method, consistent with browser practice.

Example

Listing 15-6 lets you experiment with a variety of values to see how the *string*.`substr()` method works.

LISTING 15-6

Reading a Portion of a String

HTML: jsb-15-06.html

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>String Slicing and Dicing, Part II</title>
      <script type="text/javascript" src="../jsb-global.js"></script>
      <script type="text/javascript" src="jsb-15-06.js"></script>
   </head>
   <body>
      <h1>String Slicing and Dicing, Part II</h1>
```

```
      <h2>String substr() Method</h2>

      <form action="string-substr.php">
         <p>
            <label for="source">Text used for the methods:</label>
         </p>
         <p>
            <textarea id="source" name="source" cols="60" rows="1">
Electroencephalograph</textarea>
            <pre>
----|----|----|----|-
    5    10   15   20
            </pre>
         </p>

         <table>
            <thead>
               <tr>
                  <th>Method</th>
                  <th>Parameters</th>
                  <th>Results</th>
               </tr>
            </thead>
            <tbody>
               <tr>
                  <td>string.substr()</td>
                  <td>(
                     <select id="parameter1" name="parameter1">
                        <option value="0">0</option>
                        <option value="1">1</option>
                        <option value="2">2</option>
                        <option value="3">3</option>
                        <option value="5">5</option>
                     </select>,
                     <select id="parameter2" name="parameter2">
                  <option>(None)</option>
                  <option value="5">5</option>
                  <option value="10">10</option>
                  <option value="20">20</option>
                     </select>
                  )</td>
                  <td>
                     <input type="text" id="result" name="result" size="25" />
                  </td>
               </tr>
            </tbody>
         </table>
      </form>
   </body>
</html>
```

continued

LISTING 15-6 *(continued)*

JavaScript: jsb-15-06.js

```
// initialize when the page has loaded
addEvent(window, 'load', initialize);

function initialize()
{
   // do this only if the browser can handle DOM methods
   if (document.getElementById)
   {
      // check for critical elements
      var oSource = document.getElementById('source');
      var oParameter1 = document.getElementById('parameter1');
      var oParameter2 = document.getElementById('parameter2');
      var oResult = document.getElementById('result');

         // if they all exist...
         if (oSource && oParameter1 && oParameter2 && oResult)
         {
            // apply behaviors
            addEvent(oParameter1, 'change', showResults);
            addEvent(oParameter2, 'change', showResults);
         }
         else
         {
            alert('Critical element not found');
         }
   }
}

// execute the method and display the result
function showResults()
{
   // get source text
   var oSource = document.getElementById('source');
   var sSource = oSource.firstChild.nodeValue;

   // get paramaters
   var oParameter1 = document.getElementById('parameter1');
   var iParameter1 = parseInt(oParameter1.options[oParameter1.selectedIndex].value);

   var oParameter2 = document.getElementById('parameter2');
   var iParameter2 = parseInt(oParameter2.options[oParameter2.selectedIndex].value);

   // generate result & display it
   var oResult = document.getElementById('result');

      if (!iParameter2)
      {
         oResult.value = sSource.substr(iParameter1);
```

```
      }
      else
      {
         oResult.value = sSource.substr(iParameter1, iParameter2);
      }
}
```

Related Items: *string*.substring() method

string.substring(*indexA[, indexB]*)

Returns: String of characters between index values *indexA* and *indexB*

Compatibility: WinIE3+, MacIE3+, NN2+, Moz+, Safari+, Opera+, Chrome+

The *string*.substring() method enables your scripts to extract a copy of a contiguous range of characters from any string. The parameters to this method are the starting and ending index values (the first character of the string object is index value 0) of the main string from which the excerpt should be taken. An important item to note is that the excerpt goes up to, but does not include, the character pointed to by the higher index value.

It makes no difference which index value in the parameters is larger than the other: The method starts the excerpt from the lowest value and continues to (but does not include) the highest value. If both index values are the same, the method returns an empty string; and if you omit the second parameter, the end of the string is assumed to be the endpoint.

Example

Listing 15-7 lets you experiment with a variety of values to see how the *string*.substring() method works.

LISTING 15-7

Reading a Portion of a String

HTML: jsb-15-07.html

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>String Slicing and Dicing, Part III</title>
      <script type="text/javascript" src="../jsb-global.js"></script>
      <script type="text/javascript" src="jsb-15-07.js"></script>
   </head>
   <body>
      <h1>String Slicing and Dicing, Part III</h1>
      <h2>String substring() Method</h2>

      <form action="string-substring.php">
         <p>
            <label for="source">Text used for the methods:</label>
```

continued

LISTING 15-7 *(continued)*

```
         </p>
         <p>
            <textarea id="Textarea1" name="source" cols="60" rows="1">
Electroencephalograph</textarea>
            <pre>
----|----|----|----|-
    5    10   15   20
            </pre>
         </p>

         <table>
            <thead>
               <tr>
                  <th>Method</th>
                  <th>Parameters</th>
                  <th>Results</th>
               </tr>
            </thead>
            <tbody>
               <tr>
                  <td>string.substring()</td>
                  <td>(
                     <select id="parameter1" name="parameter1">
                        <option value="0">0</option>
                        <option value="1">1</option>
                        <option value="2">2</option>
                        <option value="3">3</option>
                        <option value="5">5</option>
                     </select>,
                     <select id="parameter2" name="parameter2">
                        <option>(None)</option>
                        <option value="3">3</option>
                        <option value="5">5</option>
                        <option value="10">10</option>
                     </select>
                  )</td>
                  <td>
                     <input type="text" id="result" name="result" size="25" />
                  </td>
               </tr>
            </tbody>
         </table>
      </form>
   </body>
</html>
```

JavaScript: jsb-15-07.js

```
// initialize when the page has loaded
addEvent(window, 'load', initialize);
```

```
function initialize()
{
   // do this only if the browser can handle DOM methods
   if (document.getElementById)
   {
      // check for critical elements
      var oSource = document.getElementById('source');
      var oParameter1 = document.getElementById('parameter1');
      var oParameter2 = document.getElementById('parameter2');
      var oResult = document.getElementById('result');

         // if they all exist...
         if (oSource && oParameter1 && oParameter2 && oResult)
         {
            // apply behaviors
            addEvent(oParameter1, 'change', showResults);
            addEvent(oParameter2, 'change', showResults);
         }
         else
         {
            alert('Critical element not found');
         }
   }
}


// execute the method and display the result
function showResults()
{
   // get source text
   var oSource = document.getElementById('source');
   var sSource = oSource.firstChild.nodeValue;

   // get paramaters
   var oParameter1 = document.getElementById('parameter1');
   var iParameter1 = parseInt(oParameter1.options[oParameter1.selectedIndex].value);

   var oParameter2 = document.getElementById('parameter2');
   var iParameter2 = parseInt(oParameter2.options[oParameter2.selectedIndex].value);

   // generate result & display it
   var oResult = document.getElementById('result');

      if (!iParameter2)
      {
         oResult.value = sSource.substring(iParameter1);
      }
      else
      {
         oResult.value = sSource.substring(iParameter1, iParameter2);
      }
}
```

Related Items: *string*.substr(), *string*.slice() methods

string.toLocaleLowerCase()
string.toLocaleUpperCase()

Returns: String

Compatibility: WinIE5.5+, MacIE-, NN6+, Moz+, Safari-, Opera+, Chrome+

These two methods are variations on the standard methods for changing the case of a string. They take into account some language systems whose cases for a particular character don't necessarily map to the Latin alphabet character mappings.

Related Items: *string*.toLowerCase(), *string*.toUpperCase() methods

string.toLowerCase()
string.toUpperCase()

Returns: The string in all lower- or uppercase, depending on which method you invoke

Compatibility: WinIE3+, MacIE3+, NN2+, Moz+, Safari+, Opera+, Chrome+

A great deal of what takes place on the Internet (and in JavaScript) is case-sensitive. URLs on some servers, for instance, are case-sensitive for directory names and filenames.

These two methods, the simplest of the string methods, return a copy of a string converted to either all lowercase or all uppercase. Any mixed-case strings get converted to a uniform case. If you want to compare user input from a field against some coded string without worrying about matching case, you can convert copies of both strings to the same case for the comparison.

Example

You can use the toLowerCase() and toUpperCase() methods on literal strings, as follows:

```
var newString = "HTTP://www.Mozilla.ORG".toLowerCase();
    // result = "http://www.mozilla.org"
```

The methods are also helpful in comparing strings when case is not important, as follows:

```
if (guess.toUpperCase() == answer.toUpperCase()) {...}
    // comparing strings without case sensitivity
```

Note that these methods do not change the case of the strings themselves, but rather output modified copies, leaving the originals intact.

Related Items: *string*.toLocaleLowerCase(), *string*.toLocaleUpperCase() methods

string.toString()
string.valueOf()

Returns: String value

Compatibility: WinIE4+, MacIE4+, NN4+, Moz+, Safari+, Opera+, Chrome+

Both of these methods return string values (as opposed to full-fledged string objects). If you have created a string object via the `new String()` constructor, the type of that item is `object`. Therefore, if you want to examine more precisely what kind of value is held by the object, you can use the `valueOf()` method to get the value and then examine it via the `typeof` operator. The `toString()` method is present for this object primarily because a string object inherits the method from the root object of JavaScript.

Example

Use The Evaluator (Chapter 4) to test the `valueOf()` method. Enter the following statements into the top text box and examine the values that appear in the Results field:

```
a = new String("hello")
typeof a
b = a.valueOf()
typeof b
```

Because all other JavaScript core objects also have the `valueOf()` method, you can build generic functions that receive a variety of object types as parameters, and the script can branch its code based on the type of value that is stored in the object.

Related Items: `typeof` operator (Chapter 22)

String Utility Functions

Figuring out how to apply the various string object methods to a string manipulation challenge is not always an easy task. The situation is only made worse if you need to support legacy or mobile browsers with limited JavaScript support. It's also difficult to anticipate every possible way you may need to massage strings in your scripts. But to help you get started, Listing 15-8 contains a fully backward-compatible library of string functions for inserting, deleting, and replacing chunks of text in a string. If your audience uses browsers capable of including external `.js` library files, that would be an excellent way to make these functions available to your scripts.

LISTING 15-8

Utility String Handlers

JavaScript: jsb-15-08.js

```
// extract front part of string prior to searchString
function getFront(sMain, sSearch)
{
   iOffset = sMain.indexOf(sSearch);

      if (iOffset == -1)
      {
         return null;
      }
      else
```

continued

LISTING 15-8 *(continued)*

```
      {
         return sMain.substring(0, iOffset);
      }
}


// extract back end of string after searchString
function getEnd(sMain, sSearch)
{
   iOffset = sMain.indexOf(sSearch);

      if (iOffset == -1)
      {
         return null;
      }
      else
      {
         return sMain.substring(iOffset + sSearch.length, sMain.length);
      }
}


// insert insertString immediately before searchString
function insertString(sMain, sSearch, sInsert)
{
   var sFront = getFront(sMain, sSearch);
   var sEnd = getEnd(sMain, sSearch);

      if (sFront == null || sEnd == null)
      {
         return null;
      }
      else
      {
         return sFront + sInsert + sSearch + sEnd;
      }
}


// remove sDelete from sMain
function deleteString(sMain, sDelete)
{
   return replaceString(sMain, sDelete, "");
}


// replace sSearch with sReplace
function replaceString(sMain, sSearch, sReplace)
{
```

```
   var sFront = getFront(sMain, sSearch);
   var sEnd = getEnd(sMain, sSearch);

      if (sFront == null || sEnd == null)
      {
         return null;
      }
      else
      {
         return sFront + sReplace + sEnd;
      }
}
```

The first two functions extract the front or end components of strings, as needed, for some of the other functions in this suite. The final three functions are the core of these string-handling functions. If you plan to use these functions in your scripts, be sure to notice the dependence that some functions have on others. Including all five functions as a group ensures that they work as designed.

A modern alternative to Listing 15-8 utilizes a combination of string and array methods to perform a global replace operation in a one-statement function:

```
function replaceString(sMain, sSearch, sReplace)
{
   return sMain.split(sSearch).join(sReplace);
}
```

Going one step further, you can create a custom method to use with all string values or objects in your scripts. Simply let the following statement execute as the page loads:

```
String.prototype.replaceString = function replaceString(sMain,
      sSearch, sReplace)
{
   return sMain.split(sSearch).join(sReplace);
}
```

Then, invoke this method on any string value in other scripts on the page, as in:

```
myString = myString.replaceString(" CD ", " MP3 ");
```

(Note that global search and replace can also be accomplished using regular expressions; see Chapter 45.)

HTML markup methods

Now we come to the other group of string object methods that effectively enclose text in HTML elements. None of these methods are part of the ECMAScript Fifth Edition standard; their persistence in JavaScript implementations is clearly in support of ancient scripts. Modern web pages use stylesheets to apply cosmetic formatting to elements, and DOM methods to insert new nodes.

string.anchor("*anchorName*")	*string*.link(*locationOrURL*)
string.blink()	*string*.big()
string.bold()	*string*.small()
string.fixed()	*string*.strike()
string.fontcolor(*colorValue*)	*string*.sub()
string.fontsize(*integer1to7*)	*string*.sup()
string.italics()	

First examine the methods that don't require any parameters. You probably see a pattern: All of these methods are font-style attributes that have settings of on or off. To turn on these attributes in an HTML document, you surround the text in the appropriate tag pairs, such as `<b> ... </b>` for boldface text. These methods take the string object, attach those tags, and return the resulting text, which is ready to be put into any HTML that your scripts are building. Therefore, the expression

```
"Good morning!".bold()
```

evaluates to

```
<b>Good morning!</b>
```

Listing 15-9 shows an example of applying these string methods to text on the page. (Internet Explorer, Chrome, and Safari do not support the `<blink>` element, so while they execute the *string*.blink() method, there is no resulting blink in the display.)

LISTING 15-9

Using Simple String Methods

HTML: jsb-15-09.html

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>HTML by JavaScript</title>
      <script type="text/javascript" src="../jsb-global.js"></script>
      <script type="text/javascript" src="jsb-15-09.js"></script>
   </head>
   <body>
      <h1>HTML by JavaScript</h1>
      <ul id="tag-list">
         <li id="anchor">anchor</li>
         <li id="big">big</li>
         <li id="blink">blink</li>
         <li id="bold">bold</li>
         <li id="fixed">fixed</li>
         <li id="fontcolor">fontcolor</li>
```

```
         <li id="fontsize">fontsize</li>
         <li id="italics">italics</li>
         <li id="link">link</li>
         <li id="small">small</li>
         <li id="strike">strike</li>
         <li>sub H<span id="sub">2</span>O</li>
         <li>sup E=mc<span id="sup">2</span></li>
      </ul>
      <form action="html-by-javascript.php">
         <p>
            <button id="apply-markup">Apply Markup</button>
         </p>
      </form>
   </body>
</html>
```

JavaScript: jsb-15-09.js

```
// initialize when the page has loaded
addEvent(window, 'load', initialize);

function initialize()
{
   // do this only if the browser can handle DOM methods
   if (document.getElementById)
   {
      // check for button
      var oButton = document.getElementById('apply-markup');

         // if it exists...
         if (oButton)
         {
            // apply behavior
            addEvent(oButton, 'click', applyMarkup);
         }
   }
}


// apply markup to text elements
function applyMarkup(evt)
{
      // consolidate event handling
      if (!evt) evt = window.event;

   var oList = document.getElementById('tag-list');
   var aElements = oList.getElementsByTagName('*');

   for (var i = 0; i < aElements.length; i++)
   {
      // get this element's id & text value
      sId = aElements[i].id;
```

continued

LISTING 15-9 *(continued)*

```
         //var sText = aElements[i].firstChild.nodeValue;
         var sText = aElements[i].innerHTML;

         switch (sId)
         {
            case 'anchor':   sText = sText.anchor('anchor-name');   break;
            case 'big':   sText = sText.big();   break;
            case 'blink':   sText = sText.blink();   break;
            case 'bold':   sText = sText.bold();   break;
            case 'fixed':   sText = sText.fixed();   break;
            case 'fontcolor':   sText = sText.fontcolor('red');   break;
            case 'fontsize':   sText = sText.fontsize(7);   break;
            case 'italics':   sText = sText.italics();   break;
            case 'link':   sText = sText.link('http://example.com/');   break;
            case 'small':   sText = sText.small();   break;
            case 'strike':   sText = sText.strike();   break;
            case 'sub':   sText = sText.sub();   break;
            case 'sup':   sText = sText.sup();   break;
            default: continue;
         }

         // replace the element's contents with the marked-up text
         aElements[i].innerHTML = sText;
      }
      alert('Here is the modified markup:\n' + oList.innerHTML);

      // cancel form submission
         // W3C DOM method (hide from IE)
         if (evt.preventDefault) evt.preventDefault();
      // IE method
      return false;
   }
```

string.fontsize() and *string*.fontcolor() affect the font characteristics of strings displayed in the HTML page. The parameters for these items are pretty straightforward — an integer between 1 and 7, corresponding to the seven browser font sizes, and a color value (as either a hexadecimal triplet or color constant name) for the designated text.

The final two string methods let you create an anchor and a link out of a string. The *string*.anchor() method uses its parameter to create a name for the anchor. Thus, the following expression

```
"Table of Contents".anchor("toc")
```

evaluates to

```
<a name="toc">Table of Contents</a>
```

In a similar fashion, the *string*.`link()` method expects a valid location or URL as its parameter, creating a genuine HTML link out of the string:

```
"Back to Home".link("index.html")
```

This evaluates to the following:

```
<a href="index.html">Back to Home</a>
```

The primary reason that we rarely, if ever, use these methods in contemporary web site scripting is that we endeavor to separate development layers — in this case, HTML structure from CSS presentation and from JavaScript behavior. The more we keep these layers apart, the more modular and efficient development, debugging, and modification will be. For example, if all the font size and color details are specified in the style sheet, and not hard-coded in to HTML or JavaScript, then the appearance of the text on a page can be changed independently of the page's markup and behavior, and by different personnel with different skill sets. Similarly, it makes sense to keep markup in the HTML document and to enable JavaScript to operate on that markup, but not to stipulate it unless absolutely necessary.

Changing the cosmetic appearance of a page by changing its markup is a very "90s" thing to do. It relies on the default styling of the browser for a given tag, instead of taking responsibility for styling in one's own stylesheet. A modern script will change the appearance of an element by assigning an `id` or `className`, and letting the stylesheet determine how that will be presented.

URL String Encoding and Decoding

When browsers and servers communicate, some non-alphanumeric characters that we take for granted (such as spaces) cannot make the journey in their native form. Only a narrower set of letters, numbers, and punctuation is allowed. To accommodate the rest, the characters must be encoded with a special symbol (%) and their hexadecimal ASCII values. For example, the space character is hex 20 (ASCII decimal 32). When encoded, it looks like `%20`. You may have seen this symbol in browser history lists or URLs.

JavaScript includes two functions, `encodeURIComponent()` and `decodeURIComponent()`, that offer instant conversion of whole strings. To convert a plain string to one with these escape codes, use the escape function, as in

```
encodeURIComponent("Howdy Pardner"); // result = "Howdy%20Pardner"
```

The `decodeURIComponent()` function converts the escape codes into human-readable form.

Cross-Reference

Both of these functions are covered in Chapter 24, "Global Functions and Statements." ■

CHAPTER 16

The Math, Number, and Boolean Objects

IN THIS CHAPTER

Advanced math operations

Number base conversions

Working with integers and floating-point numbers

The introduction to data types and values in Chapter 8's tutorial scratched the surface of JavaScript's numeric and Boolean powers. In this chapter, you look more closely at JavaScript's way of working with numbers and Boolean data.

Math often frightens away budding programmers. As you've seen so far in this book, however, you don't really have to be a math genius to program in JavaScript. The powers described in this chapter are here when you need them — if you need them. So if math is not your strong suit, don't freak out over the terminology here.

An important point to remember about the objects described in this chapter is that (like string values and string objects) numbers and Booleans are both values and objects. Fortunately for script writers, the differentiation is rarely, if ever, a factor, unless you get into some very sophisticated programming. To those who actually write the JavaScript interpreters inside the browsers we use, the distinctions are vital.

For most scripters, the information about numeric data types and conversions, as well as the `Math` object, are important to know.

Numbers in JavaScript

More powerful programming languages have many different kinds of numbers, each related to the amount of memory the number occupies in the computer. Managing all these different types may be fun for some, but it gets in the way of quick scripting. A JavaScript number has only two possibilities. It can be an integer or a floating-point value. An *integer* is any whole number within a humongous range that does not have any fractional part. Integers never contain a decimal point in their representation. *Floating-point numbers* in JavaScript spread across the same range, but they are represented with a decimal point and some fractional value. If you are an experienced programmer, refer to the discussion about the `Number`

object later in this chapter to see how the JavaScript number type lines up with the numeric data types you use in other programming environments.

Integers and floating-point numbers

Deep inside a computer, the microprocessor has an easier time performing math on integer values compared to any number with a decimal value tacked onto it. The microprocessor must go through extra work to add even two such floating-point numbers. We, as scripters, are unfortunately saddled with this historical baggage and must be conscious of the type of number used in certain calculations.

Most internal values generated by JavaScript, such as index values and `length` properties, consist of integers. Floating-point numbers usually come into play as the result of the division of numeric values, special values such as pi, and human-entered values such as dollars and cents. Fortunately, JavaScript is forgiving if you try to perform math operations on mixed numeric data types. Notice how the following examples resolve to the appropriate data type:

```
3 + 4 = 7 // integer result
3 + 4.1 = 7.1 // floating-point result
3.9 + 4.1 = 8 // integer result
```

Of the three examples, perhaps only the last result is unexpected. When two floating-point numbers yield a whole number, the result is rendered as an integer.

When dealing with floating-point numbers, be aware that not all browser versions return the precise same value down to the last digit to the right of the decimal. For example, the following table shows the result of 8/9 as calculated by various scriptable browsers, and converted for string display:

NN3 & NN4	.8888888888888888
NN6+/Moz+/Safari+/Opera+/Chrome+	0.8888888888888888
WinIE3	0.888888888888889
WinIE4+	0.8888888888888888

Clearly, from this display, you don't want to use floating-point math in JavaScript browsers to plan space flight trajectories or other highly accurate mission-critical calculations. Even for everyday math, however, you need to be cognizant of floating-point errors that accrue in PC arithmetic.

In Navigator, JavaScript relies on the operating system's floating-point math for its own math. Operating systems that offer accuracy to as many places to the right of the decimal as JavaScript displays are exceedingly rare. As you can detect from the preceding table, modern browsers agree about how many digits to display and how to perform internal rounding for this display. That's good for the math, but not particularly helpful when you need to display numbers in a specific format.

Until you get to IE5.5, Mozilla-based browsers, and other W3C-compatible browsers, JavaScript does not offer built-in facilities for formatting the results of floating-point arithmetic. (For modern browsers, see the `Number` object later in this chapter for formatting methods.) Listing 16-1 demonstrates a generic formatting routine for positive values, plus a specific call that turns a value into a dollar value. Remove the comments, and the routine is fairly compact.

Cross-Reference

The function to assign event handlers throughout the code in this chapter, and much of the book, is `addEvent()`**, a cross-browser event handler explained in detail in Chapter 32, "Event Objects."** ■

LISTING 16-1

A Generic Number-Formatting Routine

HTML: jsb-16-01.html

```
<!DOCTYPE html>
<html>
   <head>
     <meta http-equiv="content-type" content="text/html;charset=utf-8">
     <title>Number Formatting</title>
     <script type="text/javascript" src="../jsb-global.js"></script>
     <script type="text/javascript" src="jsb-16-01.js"></script>
  </head>
  <body>
     <h1>How to Make Money</h1>
     <form>
        <p>
          <label for="entry">Enter a positive floating point value or arithmetic
          expression to be converted to a currency format:</label>
        </p>
        <p>
          <input type="text" id="entry" name="entry" value="1/3">
          <input type="button" id="dollars" value="&gt; Dollars and Cents &gt;">
          <input type="text" id="result" name="result">
        </p>
     </form>
   </body>
</html>
```

JavaScript: jsb-16-01.js

```
// initialize when the page has loaded
addEvent(window, 'load', initialize);

function initialize()
{
   // do this only if the browser can handle DOM methods
   if (document.getElementById)
   {
      // point to critical elements
      var oButton = document.getElementById('dollars');
      var oEntry = document.getElementById('entry');
      var oResult = document.getElementById('result');

         // if they all exist...
         if (oButton && oEntry && oResult)
```

continued

LISTING 16-1 *(continued)*

```
        {
           // apply behavior to button
           oButton.onclick =
              function() { oResult.value = dollarize(oEntry.value); }
        }
    }
}
// turn incoming expression into a dollar value
function dollarize(expr)
{
   return "$" + format(expr,2);
}

// generic positive number decimal formatting function
function format(expr, decplaces)
{
   // evaluate the incoming expression
   var val = eval(expr);

   // raise the value by power of 10 times the number of decimal places;
   // round to an integer; convert to string
   var str = "" + Math.round(val * Math.pow(10, decplaces));

   // pad small value strings with zeros to the left of rounded number
   while (str.length <= decplaces)
   {
      str = "0" + str;
   }

   // establish location of decimal point
   var decpoint = str.length - decplaces;

   // assemble final result from:
   // (a) the string up to the position of the decimal point;
   // (b) the decimal point; and
   // (c) the balance of the string. Return finished product.
   return str.substring(0,decpoint) + "." + str.substring(decpoint, str.length);
}
```

This routine may seem like a great deal of work, but it's essential if your application relies on floating-point values and specific formatting for all browsers.

Note

The global `eval()` method used in this example is not a terrific choice for production work. Because `eval()` faithfully interprets all JavaScript expressions, it lays bare all the details of the script, its object and variable values, and the DOM, for the savvy visitor to peruse and act upon. The JavaScript language has deliberately been constrained to prevent security breaches on the client computer, but `XMLHttpRequest` gives access to the server. It's not appropriate to give strangers an open and unfiltered `eval()` input. ■

You can also enter floating-point numbers with exponents. An exponent is signified by the letter *e* (upper- or lowercase), followed by a sign (+ or −) and the exponent value. Here are examples of floating-point values expressed as exponents:

```
1e6 // 1,000,000 (the "+" symbol is optional on positive exponents)
1e-4 // 0.0001 (plus some error further to the right of the decimal)
-4e-3 // -0.004
```

For values between 1e-5 and 1e15, JavaScript renders numbers without exponents (although you can force a number to display in exponential notation in modern browsers). All other values outside these boundaries return with exponential notation in all browsers.

Hexadecimal and octal integers

JavaScript enables you to work with values in decimal (base-10), hexadecimal (base-16), and octal (base-8) formats. You have only a few rules to follow when dealing with any of these values.

Decimal values cannot begin with a leading 0. Therefore, if your page asks users to enter decimal values that begin with a 0, your script must strip those zeros from the input string, or use the number parsing global functions (described in the next section), before performing any math on the values.

Hexadecimal integer values are expressed with a leading `0x` or `0X`. (That's a zero, not the letter *o*.) The A through F values can appear in upper- or lowercase, as you prefer. Here are some hex values:

```
0X2B
0X1a
0xcc
```

Don't confuse the hex values used in arithmetic with the hexadecimal values used in color property specifications for web documents. Those values are expressed in a special *hexadecimal triplet* format, which begins with a crosshatch symbol followed by the three hex values bunched together (such as `#c0c0c0`).

Octal values are represented by a leading 0 followed by any digits between 0 and 7. Octal values consist only of integers.

You are free to mix and match base values in arithmetic expressions, but JavaScript renders all results in decimal form. For conversions to other number bases, you have to employ a user-defined function in your script. Listing 16-2, for example, is a function that converts any decimal value from 0 to 255 into a JavaScript hexadecimal value.

LISTING 16-2

Decimal-to-Hexadecimal Converter Function

```
function toHex(dec)
{
   hexChars = "0123456789ABCDEF";
   if (dec > 255)
   {
```

continued

LISTING 16-2 *(continued)*

```
      return null;
   }
   var i = dec % 16;
   var j = (dec - i) / 16;
   result = "0X";
   result += hexChars.charAt(j);
   result += hexChars.charAt(i);
   return result;
}
```

The `toHex()` conversion function assumes that the value passed to the function is a decimal integer. If you simply need a hexadecimal representation of a number in string format, see the `toString()` method in Chapter 15, "The String Object."

Converting strings to numbers

What is missing so far from this discussion is a way to convert a number represented as a string to a number with which the JavaScript arithmetic operators can work. Before you get too concerned about this, be aware that most JavaScript operators and math methods gladly accept string representations of numbers and handle them without complaint. You will run into data type incompatibilities most frequently when trying to accomplish addition with the + operator (which is also the string concatenation operator). Also know that if you perform math operations on values retrieved from form text boxes, those object `value` properties are strings. Therefore, in many cases, you need to convert those string values to a number type for math operations.

Conversion to numbers requires one of two JavaScript functions:

```
parseInt(string [,radix])
parseFloat(string  [,radix])
```

These functions are inspired by the Java language. The term *parsing* has many implied meanings in programming. One meaning is the same as *extracting*. The `parseInt()` function returns whatever integer value it can extract from the string passed to it; the `parseFloat()` function returns the floating-point number that can be extracted from the string. Here are some examples and their resulting values:

```
parseInt("42")         // result = 42
parseInt("42.33")      // result = 42
parseFloat("42.33")    // result = 42.33
parseFloat("42")       // result = 42
parseFloat("fred")     // result = NaN (Not a Number)
```

Because the `parseFloat()` function can also work with an integer and return an integer value, you may prefer using this function in scripts that have to deal with either kind of number, depending on the string entered into a text field by a user.

An optional second parameter to both functions enables you to specify the base of the number represented by the string. This comes in handy particularly when you need a decimal number from a string that starts with one or more zeros. Normally, the leading zero indicates an

octal value. But if you force the conversion to recognize the string value as a decimal, it is converted the way you expect:

```
parseInt("010")       // result = 8
parseInt("010",10)    // result = 10
parseInt("F2")        // result = NaN
parseInt("F2", 16)    // result = 242
```

Use these functions wherever you need the integer or floating-point value. For example:

```
var result = 3 + parseInt("3");    // result = 6
var ageVal = parseInt(document.forms[0].age.value);
```

The latter technique ensures that the string value of this property is converted to a number, although you should do more data validation — see Chapter 46, "Data-Entry Validation" (on the CD-ROM) — before trying any math on a user-entered value.

Both the `parseInt()` and `parseFloat()` methods start working on the first character of a string and continue until there are no more numbers or decimal characters. That's why you can use them on strings — such as the one returned by the `navigator.appVersion` property (for example, `6.0 (Windows; en-US)`) — to obtain just the leading, numeric part of the string. If the string does not begin with an acceptable character, the methods return `NaN` (not a number).

Converting numbers to strings

If you attempt to pass a numeric data type value to many of the string methods discussed in Chapter 15, JavaScript complains. Therefore, you should convert any number to a string before, for example, you find out how many digits make up a number.

Several ways exist to force conversion from any numeric value to a string. The old-fashioned way is to precede the number with an empty string and the concatenation operator. For example, assume that a variable named `dollars` contains the integer value of `2500`. To use the string object's `length` property (discussed later in this chapter) to find out how many digits the number has, use this construction:

```
("" + dollars).length    // result = 4
```

The parentheses force JavaScript to evaluate the concatenation before attempting to extract the `length` property.

A more elegant way is to use the `toString()` method. Construct such statements as you would to invoke any object's method. For example, to convert the `dollars` variable value to a string, use this statement:

```
dollars.toString()    // result = "2500"
```

This method has one added power in modern browsers: You can specify a number base for the string representation of the number. Called the *radix,* the base number is added as a parameter to the method name. Here is an example of creating a numeric value for conversion to its hexadecimal equivalent as a string:

```
var x = 30;
var y = x.toString(16);    // result = "1e"
```

Use a parameter of `2` for binary results and `8` for octal. The default is base `10`. Be careful not to confuse these conversions with true numeric conversions. You cannot use results from the `toString()` method as numeric operands in other statements.

Finally, in IE5.5+, Mozilla-based browsers, and other W3C browsers, three additional methods of the `Number` object — `toExponential()`, `toFixed()`, and `toPrecision()` — return string versions of numbers formatted according to the rules and parameters passed to the methods. We describe these in detail later in this chapter.

When a number isn't a number

In a couple of examples in the previous section, you probably noticed that the result of some operations was a value named `NaN`. That value is not a string, but rather a special value that stands for Not a Number. For example, if you try to convert the string `"joe"` to an integer with `parseFloat()`, the function cannot possibly complete the operation. It reports back that the source string, when converted, is not a number.

When you design an application that requests user input or retrieves data from a server-side database, you cannot be guaranteed that a value you need to be numeric is, or can be converted to, a number. If that's the case, you need to see if the value is a number before performing a math operation on it. JavaScript provides a special global function, `isNaN()`, that enables you to test the "numberness" of a value. The function returns `true` if the value is not a number and `false` if it is a number. For example, you can examine a form field that should be a number:

```
var ageEntry = parseInt(document.forms[0].age.value);
   if (isNaN(ageEntry))
   {
      alert("Try entering your age again.");
   }
```

Math Object

Whenever you need to perform math that is more demanding than simple arithmetic, look through the list of `Math` object methods for the solution.

Syntax

Accessing `Math` object properties and methods:

```
Math.property
Math.method(value [, value])
```

Compatibility: WinIE3+, MacIE3+, NN2+, Moz+, Safari+, Opera+, Chrome+

About this object

In addition to the typical arithmetic operations (covered in detail in Chapter 22, "JavaScript Operators"), JavaScript includes more advanced mathematical powers that you can access in a way that may seem odd to you if you have not programmed in true object-oriented environments before. Although most arithmetic takes place on the fly (such as `var result = 2 + 2`), the rest requires use of the

JavaScript internal `Math` object (with a capital "M"). The `Math` object brings with it several properties (which behave like some other languages' constants) and many methods (which behave like some other languages' math functions).

The way you use the `Math` object in statements is the same way you use any JavaScript object: You create a reference beginning with the `Math` object's name, a period, and the name of the property or method you need:

```
Math.property | method([parameter]. . . [,parameter])
```

Property references return built-in values, such as pi. Method references require one or more values to be sent as parameters of the method. Every method returns a result.

Properties

JavaScript `Math` object properties represent a number of valuable constant values in math. Table 16-1 shows you those methods and their values, as displayed to 16 decimal places.

Because these property expressions return their constant values, you use them in your regular arithmetic expressions. For example, to obtain the circumference of a circle whose diameter is in variable `d`, employ this statement:

```
circumference = d * Math.PI;
```

Perhaps the most common mistakes scripters make with these properties are failing to capitalize the `Math` object name and observing the case-sensitivity of property names.

TABLE 16-1

JavaScript Math Properties

Property	Value	Description
`Math.E`	`2.718281828459045091`	Euler's constant
`Math.LN2`	`0.6931471805599452862`	Natural log of 2
`Math.LN10`	`2.302585092994045901`	Natural log of 10
`Math.LOG2E`	`1.442695040888963387`	Log base-2 of E
`Math.LOG10E`	`0.4342944819032518167`	Log base-10 of E
`Math.PI`	`3.141592653589793116`	π
`Math.SQRT1_2`	`0.7071067811865475727`	Square root of 0.5
`Math.SQRT2`	`1.414213562373095145`	Square root of 2

Methods

Methods make up the balance of JavaScript `Math` object powers. With the exception of the `Math.random()` method, all `Math` object methods take one or more values as parameters. Typical

trigonometric methods operate on the single values passed as parameters; others determine which of the numbers passed along are the highest or lowest of the group. The `Math.random()` method takes no parameters, but returns a randomized, floating-point value between `0` and `1`. Table 16-2 lists all the `Math` object methods with their syntax, and descriptions of the values they return.

TABLE 16-2

Math Object Methods

Method Syntax	Returns
`Math.abs(val)`	Absolute value of *val*
`Math.acos(val)`	Arc cosine (in radians) of *val*
`Math.asin(val)`	Arc sine (in radians) of *val*
`Math.atan(val)`	Arc tangent (in radians) of *val*
`Math.atan2(val1, val2)`	Angle of polar coordinates *x* and *y*
`Math.ceil(val)`	Next integer greater than or equal to *val*
`Math.cos(val)`	Cosine of *val*
`Math.exp(val)`	Euler's constant to the power of *val*
`Math.floor(val)`	Next integer less than or equal to *val*
`Math.log(val)`	Natural logarithm (base e) of *val*
`Math.max(val1, val2)`	The greater of *val1* or *val2*
`Math.min(val1, val2)`	The lesser of *val1* or *val2*
`Math.pow(val1, val2)`	*Val1* to the power of *val2*
`Math.random()`	Random number between 0 and 1
`Math.round(val)`	N+1 when *val* >= n.5; otherwise N
`Math.sin(val)`	Sine (in radians) of *val*
`Math.sqrt(val)`	Square root of *val*
`Math.tan(val)`	Tangent (in radians) of *val*

Although HTML may not exactly be a graphic artist's dream environment, it is possible to use JavaScript trig functions to obtain a series of values for HTML-generated charting. Since the advent of positionable elements, scripters have been able to apply their knowledge of using these functions to define fancy trajectories for flying elements. For scripters who are not trained in programming, math is often a major stumbling block. But as you've seen so far, you can accomplish a great deal with JavaScript by using simple arithmetic and a little bit of logic — leaving the heavy-duty math for those who love it.

Creating random numbers

One of the handiest methods in the Math object is Math.random(), which returns a random floating-point value between 0 and 1. If you design a script to act like a card game, you need random integers between 1 and 52; for dice, the range is 1 to 6 per die. To generate a random integer between zero and any top value (n), use the following formula (where n is the top number):

```
Math.floor(Math.random() * n)
```

This produces integers between 0 and n-1. To produce the spread between 1 and n, either add 1 or change floor() to ceil():

```
Math.floor(Math.random() * n) + 1
Math.ceil(Math.random() * n)
```

To generate random numbers within a range that starts somewhere other than zero, use this formula:

```
Math.floor(Math.random() * (n - m + 1)) + m
```

Here, m is the lowest possible integer value of the range, and n equals the top number of the range. For the dice game, the formula for each die is

```
newDieValue = Math.floor(Math.random() * 6) + 1;
```

Math object shortcut

In Chapter 21, "Control Structures and Exception Handling," you see details about a JavaScript construction that enables you to simplify the way you address multiple Math object properties and methods in statements. The trick is to use the with statement.

In a nutshell, the with statement tells JavaScript that the next group of statements (inside the braces) refers to a particular object. In the case of the Math object, the basic construction looks like this:

```
with (Math)
{
   //statements
}
```

For all intervening statements, you can omit the specific references to the Math object. Compare the long reference way of calculating the area of a circle (with a radius of six units)

```
result = Math.pow(6,2) * Math.PI;
```

to the shortcut reference way:

```
with (Math)
{
   result = pow(6,2) * PI;
}
```

Though the latter occupies more lines of code, the object references are shorter and more natural to read. For a longer series of calculations involving Math object properties and methods, the with

construction saves keystrokes and reduces the likelihood of a case-sensitive mistake with the object name in a reference. You can also include other full-object references within the `with` construction; JavaScript attempts to attach the object name only to those references lacking an object name. On the downside, the `with` construction is not particularly efficient in JavaScript because it must perform a lot of internal tracking in order to work. So, it's best not to use it in loops that iterate a large number of times.

Number Object

Properties	Methods
constructor	toExponential()
MAX_VALUE	toFixed()
MIN_VALUE	toLocaleString()
NaN	toString()
NEGATIVE_INFINITY	toPrecision()
POSITIVE_INFINITY	valueOf()
prototype	

Syntax

Creating a `Number` object:

```
var val = new Number(number);
```

Accessing number and `Number` object properties and methods:

```
number.property | method([parameters])
Number.property | method([parameters])
```

Compatibility: WinIE4+, MacIE4+, NN3+, Moz+, Safari+, Opera+, Chrome+

About this object

The `Number` object is rarely used because (for the most part) JavaScript satisfies day-to-day numeric needs with a plain number value. But the `Number` object contains some information and power of value to serious programmers.

First on the docket are properties that define the ranges for numbers in the language. The largest number is 1.79E+308; the smallest number is 2.22E-308. Any number larger than the maximum is `POSITIVE_INFINITY`; any number smaller than the minimum is `NEGATIVE_INFINITY`. Rarely will you accidentally encounter these values.

More to the point of a JavaScript object, however, is the `prototype` property. Chapter 15 shows how to add a method to a `string` object's prototype, such that every newly created object contains that method. The same goes for the `Number.prototype` property. If you have a need to add common functionality to every number object, this is where to do it. This prototype facility is unique to full-fledged number objects and does not apply to plain number values. For experienced programmers who care about such matters, JavaScript number objects and values are defined internally as IEEE double-precision 64-bit values.

Properties

constructor

(See `string.constructor` in Chapter 15)

MAX_VALUE
MIN_VALUE
NEGATIVE_INFINITY
POSITIVE_INFINITY

Value: Number Read-Only

Compatibility: WinIE4+, MacIE4+, NN3+, Moz+, Safari+, Opera+, Chrome+

The `Number.MAX_VALUE` and `Number.MIN_VALUE` properties belong to the static `Number` object. They represent constants for the largest and smallest possible positive numbers that JavaScript (and ECMAScript) can work with. Their actual values are $1.7976931348623157 \infty 10^{308}$, and $5 \infty 10^{-324}$, respectively. A number that falls outside the range of allowable numbers is equal to the constant `Number.POSITIVE_INFINITY` or `Number.NEGATIVE_INFINITY`.

Example

Enter each of the four `Number` object expressions into the top text field of The Evaluator (Chapter 4, "Javascript Essentials") to see how the browser reports each value.

```
Number.MAX_VALUE
Number.MIN_VALUE
Number.NEGATIVE_INFINITY
Number.POSITIVE_INFINITY
```

Related Items: `NaN` property; `isNaN()` global function

NaN

Value: NaN Read-Only

Compatibility: WinIE4+, MacIE4+, NN3+, Moz+, Safari+, Opera+, Chrome+

The `NaN` property is a constant that JavaScript uses to report when a number-related function or method attempts to work on a value other than a number, or when the result is something other than a number. You encounter the `NaN` value, most commonly as the result of the `parseInt()` and `parseFloat()` functions, whenever a string undergoing conversion to a number lacks a numeral as the first character. Use the `isNaN()` global function to see if a value is an `NaN` value.

Example

See the discussion of the `isNaN()` function in Chapter 24.

Related Items: `isNaN()` global function

`prototype`

(See `String.prototype` in Chapter 15)

Methods

`number.toExponential(`*`fractionDigits`*`)`
`number.toFixed(`*`fractionDigits`*`)`
`number.toPrecision(`*`precisionDigits`*`)`

Returns: String

Compatibility: WinIE5.5+, MacIE-, NN6+, Moz+, Safari+, Opera+, Chrome+

These three methods let scripts control the formatting of numbers for display as string text. Each method has a unique purpose, but they all return strings. You should perform all math operations as unformatted number objects because the values have the most precision. Only after you are ready to display the results should you use one of these methods to convert the number to a string for display as body text or assignment to a text field.

The `toExponential()` method forces a number to display in exponential notation, even if the number is in the range in which JavaScript normally uses standard notation. The parameter is an integer specifying how many digits to the right of the decimal should be returned. All digits that far to the right of the decimal are returned, even if they are zero. For example, if a variable contains the numeric value `345`, applying `toExponential(3)` to that value yields `3.450e+2`, which is JavaScript's exponential notation for $3.45 \infty 10^2$.

Use the `toFixed()` method when you want to format a number with a specific number of digits to the right of the decimal. This is the method you use, for instance, to display the results of a financial calculation in units and hundredths of units (for example, dollars and cents). The parameter to the method is an integer indicating the number of digits to be displayed to the right of the decimal. If the number being formatted has more numbers to the right of the decimal than the number of digits specified by the parameter, the method rounds the rightmost visible digit — but only with respect to the unrounded value of the next digit. For example, the value `123.455` fixed to two digits to the right of the decimal is rounded up to `123.46`. But if the starting value is `123.4549`, the method ignores the 9 and sees that the 4 to the right of the 5 should be rounded down; therefore, the result is `123.45`. Do not consider the `toFixed()` method to be an accurate rounder of numbers; however, it does a satisfactory job in most cases.

The final method is `toPrecision()`, which enables you to define how many total digits of a number (including digits to the left and right of the decimal) to display. In other words, you define the precision of a number. The following list demonstrates the results of several parameter values signifying a variety of precisions:

```
var num = 123.45
num.toPrecision(1)    // result = 1e+2
num.toPrecision(2)    // result = 1.2e+2
```

```
num.toPrecision(3)    // result = 123
num.toPrecision(4)    // result = 123.5
num.toPrecision(5)    // result = 123.45
num.toPrecision(6)    // result = 123.450
```

Notice that the same kind of rounding can occur with `toPrecision()` as it does for `toFixed()`.

Example

You can use The Evaluator (see Chapter 4) to experiment with all three of these methods, with a variety of parameter values. Before invoking any method, be sure to assign a numeric value to one of the built-in global variables in The Evaluator (`a` through `z`).

```
a = 10/3
a.toFixed(4)
"$" + a.toFixed(2)
```

None of these methods works with number literals (for example, `123.toExponential(2)` does not work).

Related Items: `Math` object

number.toLocaleString()

Returns: String

Compatibility: WinIE5.5+, MacIE5+, NN6+, Moz+, Safari+, Opera+, Chrome+

The `number.toLocaleString()` method returns a string value version of the current number in a format that may vary according to a browser's locale settings. According to the ECMA Edition 5 standard, browsers have some leeway in determining exactly how the `toLocaleString()` method should return a string value that conforms with the language standard of the client system or browser.

Related Items: `number.toFixed()`, `number.toString()` methods

number.toString([*radix*])

Returns: String

Compatibility: WinIE4+, MacIE4+, NN4+, Moz+, Safari+, Opera+, Chrome+

The `number.toString()` method returns a string value version of the current number. The default radix parameter (`10`) converts the value to base-10 notation if the original number isn't already of that type. Or, you can specify other number bases (for example, `2` for binary, `16` for hexadecimal) to convert the original number to the other base — as a string, not a number — for further calculation.

Example

Use The Evaluator (Chapter 4) to experiment with the `toString()` method. Assign the number `12` to the variable `a` and see how the number is converted to strings in a variety of number bases:

```
a = 12
a.toString()    // base 10
a.toString(2)
a.toString(16)
```

Related Items: `toLocaleString()` method

number.valueOf()

(See `string.valueOf()` in Chapter 15)

Boolean Object

Properties	Methods
`constructor`	`toString()`
`prototype`	`valueOf()`

Syntax

Creating a `Boolean` object:

```
var val = new Boolean(BooleanValue);
```

Accessing `Boolean` object properties:

```
BooleanObject.property | method
```

Compatibility: WinIE4+, MacIE4+, NN3+, Moz+, Safari+, Opera+, Chrome+

About this object

You work with Boolean values a lot in JavaScript — especially as the result of conditional tests. Just as string values benefit from association with string objects, and their properties and methods, so, too, do Boolean values receive aid from the `Boolean` object. For example, when you display a Boolean value in a text box, the `"true"` or `"false"` string is provided by the `Boolean` object's `toString()` method, so you don't have to invoke it directly.

The only time you need to even think about a `Boolean` object is if you wish to attach some property or method to `Boolean` objects that you create with the new `Boolean()` constructor. Parameter values for the constructor include the string versions of the values, numbers (0 for `false`; any other integer for `true`), and expressions that evaluate to a Boolean value. Any such new `Boolean` object is imbued with the new properties or methods you add to the `prototype` property of the core `Boolean` object.

For details about the properties and methods of the `Boolean` object, see the corresponding listings for the `String` object in Chapter 15.

CHAPTER

17

The Date Object

IN THIS CHAPTER

Working with date and time values in JavaScript

Performing date calculations

Validating date entry form fields

Perhaps the most untapped power of JavaScript is its date and time handling. Scripters passed over the `Date` object with good cause in the early days of JavaScript, because in earlier versions of scriptable browsers, significant bugs and platform-specific anomalies made date and time programming hazardous without significant testing. Even with the improved bug situation, working with dates requires a working knowledge of the world's time zones and their relationships with the standard reference point, known as Greenwich Mean Time (GMT) or Coordinated Universal Time (abbreviated UTC).

Now that date- and time-handling has stabilized in modern browsers, I hope more scripters look into incorporating these kinds of calculations into their pages. In Chapter 57, "Application: Intelligent 'Updated' Flags," on the CD-ROM, for example, I show you an application that lets your web site highlight the areas that have been updated since each visitor's last surf ride through your pages — an application that relies heavily on date arithmetic and time-zone conversion.

Before getting to the JavaScript part of date discussions, however, the chapter summarizes key facts about time zones and their impact on scripting date and time on a browser. If you're not sure what GMT and UTC mean, the following section is for you.

Time Zones and GMT

By international agreement, the world is divided into distinct time zones that allow the inhabitants of each zone to say with confidence that when the Sun appears directly overhead, it is roughly noon, squarely in the middle of the day. The current time in the zone is what we set our clocks to — the local time.

That's fine when your entire existence and scope of life go no further than the width of your own time zone. But with instant communication among all parts of the world, your scope reaches well beyond local time. Periodically you must

be aware of the local time in other zones. After all, if you live in New York, you don't want to wake up someone in Los Angeles before dawn with a phone call from your office.

Note

For the rest of this section, we speak of the Sun ''moving'' as if Earth were the center of the solar system. We do so for the convenience of our daily perception of the Sun arcing across what appears to us as a stationary sky. In point of fact, we believe Copernicus's theories, so please delete that email you were about to send us. ■

From the point of view of the time zone over which the Sun is positioned at any given instant, all time zones to the east have already had their noon, so it is later in the day for them — one hour later per time zone (except for those few time zones offset by fractions of an hour). That's why when U.S. television networks broadcast simultaneously to the eastern and central time zones, the announced schedule for a program is "10 eastern, 9 central."

Many international businesses must coordinate time schedules of far-flung events. Doing so and taking into account the numerous time zone differences (not to mention seasonal national variations, such as daylight saving time) would be a nightmare. To help everyone out, a standard reference point was devised: the time zone running through the celestial observatory at Greenwich (pronounced GREN-itch), England. This time zone is called Greenwich Mean Time, or GMT for short. The "mean" part comes from the fact that on the exact opposite side of the globe (through the Pacific Ocean) is the international date line, another world standard that decrees where the first instance of the next calendar day appears on the planet. Thus, GMT is located at the middle, or mean, of the full circuit of the day. Not that many years ago, GMT was given another abbreviation that is not based on any one language of the planet. The abbreviation is UTC (pronounced as its letters: yu-tee-see), and the English version is Coordinated Universal Time. Whenever you see UTC, it is for all practical purposes the same as GMT.

If your personal computer's system clock is set correctly, the machine ticks away in GMT time. But because you set your local time zone in the appropriate control panel, all file time stamps and clock displays are in your local time. The machine knows what the offset time is between your local time and GMT. For daylight savings time, you may have to check a preference setting so that the offset is adjusted accordingly; in Windows-based operating systems, the system knows when the changeover occurs and prompts you if changing the offset is okay. In any case, if you travel across time zones with a laptop, you should change the computer's time zone setting, not its clock.

JavaScript's inner handling of date and time works a lot like the PC clock (on which your programs rely). Date values that you generate in a script are stored internally in GMT time; however, almost all the displays and extracted values are in the local time of the visitor (not the web site server). And remember that the date values are created on the visitor's machine by virtue of your script's generating that value — you don't send "living" date objects to the client from the server. This concept is perhaps the most difficult to grasp as you work with JavaScript date and time.

Whenever you program time and date in JavaScript for a public web page, you must take the world-view. This view requires knowing that the visitor's computer settings determine the accuracy of the conversion between GMT and local time. You'll also have to do some testing by changing your PC's clock to times in other parts of the world and making believe you are temporarily in those remote locations, which isn't always easy to do. It reminds me (Danny) of the time I was visiting Sydney, Australia. I was turning in for the night and switched on the television in the hotel. This hotel received a live satellite relay of a long-running U.S. television program, *Today*. The program broadcast from New York was for the morning of the same day I was just finishing in Sydney. Yes, this time zone stuff can make your head hurt.

The Date Object

Like a handful of other objects in JavaScript and the document object models, there is a distinction between the single, static `Date` object that exists in every window (or frame) and a date object that contains a specific date and time. The static `Date` object (uppercase *D*) is used in only a few cases: Primarily to create a new instance of a date and to invoke a couple of methods that the `Date` object offers for the sake of some generic conversions.

Most of your date and time work, however, is with instances of the `Date` object. These instances are referred to generically as date objects (lowercase *d*). Each date object is a snapshot of an exact millisecond in time, whether it be for the instant at which you generate the object or for a specific time in the past or future you need for calculations. If you need to have a live clock ticking away, your scripts will repeatedly create new date objects to grab up-to-the-millisecond snapshots of your computer's clock. To show the time on the page, extract the hours, minutes, and seconds from the snapshot date object, and then display the values as you like (for example, a digital readout, a graphical bar chart, and so on). By and large, it is the methods of a date object instance that your scripts invoke to read or modify individual components of a date object (for example, the month or hour).

Despite its name, every date object contains information about date and time. Therefore, even if you're concerned only about the date part of an object's data, time data is standing by as well. As you learn in a bit, the time element can catch you off-guard for some operations.

Creating a date object

The statement that asks JavaScript to make an object for your script uses the special object construction keyword `new`. The basic syntax for generating a new date object is as follows:

```
var dateObjectName = new Date([parameters]);
```

The date object evaluates to an object data type rather than to some string or numeric value.

With the date object's reference safely tucked away in the variable name, you access all date-oriented methods in the dot-syntax fashion with which you're already familiar:

```
var result = dateObjectName.method();
```

With variables, such as `result`, your scripts perform calculations or displays of the date object's data (some methods extract pieces of the date and time data from the object). If you then want to put some new value into the date object (such as adding a year to the date object), you assign the new value to the object by way of the method that lets you set the value:

```
dateObjectName.method(newValue);
```

This example doesn't look like the typical JavaScript assignment statement, which has an equals sign operator. But this statement is the way in which methods that set date object data work.

You cannot get very far into scripting dates without digging into time zone arithmetic. Although JavaScript may render the string equivalent of a date object in your local time zone, the internal storage is strictly GMT.

Even though you haven't yet seen details of a date object's methods, here is how you use two of them to add one year to today's date:

```
var oneDate = new Date();          // creates object with current GMT date
var theYear = oneDate.getYear();   // stores the current four-digit year
```

```
theYear = theYear + 1;            // theYear now is next year
oneDate.setYear(theYear);         // the new year value now in the object
```

At the end of this sequence, the `oneDate` object automatically adjusts all the other date components for the next year's date. The day of the week, for example, will be different, and JavaScript takes care of that for you, should you need to extract that data. With next year's data in the `oneDate` object, you may now want to extract that new date as a string value for display in a field on the page or submit it quietly to a CGI program on the server.

The issue of parameters for creating a new date object is a bit complex, mostly because of the flexibility that JavaScript offers the scripter. Recall that the job of the `new Date()` statement is to create a place in memory for all data that a date needs to store. What is missing from that task is the data — what date and time to enter into that memory spot. That's where the parameters come in.

If you leave the parameters empty, JavaScript takes that to mean you want today's date and the current time to be assigned to that new date object. JavaScript isn't any smarter, of course, than the setting of the internal clock of your page visitor's personal computer. If the clock isn't correct, JavaScript won't do any better of a job identifying the date and time.

Note

Remember that when you create a new date object, it contains the current time as well. The fact that the current date may include a time of 16:03:19 (in 24-hour time) may throw off things, such as days-between-dates calculations. Be careful. ■

To create a date object for a specific date or time, you have five ways to send values as a parameter to the `new Date()` constructor function:

```
new Date("Month dd, yyyy hh:mm:ss")
new Date("Month dd, yyyy")
new Date(yy,mm,dd,hh,mm,ss)
new Date(yy,mm,dd)
new Date(milliseconds)
```

The first four variations break down into two styles — a long string versus a comma-delimited list of data — each with optional time settings. If you omit time settings, they are set to 0 (midnight) in the date object for whatever date you entered. You cannot omit date values from the parameters — every date object must have a real date attached to it, whether you need it or not.

In the long string versions, the month is spelled out in full in English. No abbreviations are allowed. The rest of the data is filled with numbers representing the date, year, hours, minutes, and seconds, even if the order is different from your local way of indicating dates. For single-digit values, you can use either a one- or two-digit version (such as 4:05:00). Colons separate hours, minutes, and seconds.

The short versions contain a non-quoted list of integer values in the order indicated. JavaScript cannot know that a 30 means the day if you accidentally place it in the month slot.

You use the last version only when you have the millisecond value of a date and time available. This generally occurs after some math arithmetic (described later in this chapter), leaving you with a date and time in millisecond format. To convert that numeric value to a date object, use the `new Date()` constructor. From the new date object created, you can retrieve more convenient values about the date and time.

Native object properties and methods

Like the String and Array objects, the Date object features a small handful of properties and methods that all native JavaScript objects have in common. On the property side, the Date object has a prototype property, which enables you to apply new properties and methods to every date object created in the current page. You can see examples of how this works in discussions of the prototype property for String and Array objects (Chapters 15 and 18, respectively). At the same time, every instance of a date object in modern browsers has a constructor property that references the constructor function that generated the object.

A date object has numerous methods that convert date object types to strings, most of which are more specific than the generic toString() one. The valueOf() method returns the millisecond integer that is stored for a particular date.

Date methods

The bulk of a date object's methods are for reading parts of the date and time information and for changing the date and time stored in the object. These two categories of methods are easily identifiable because they all begin with the word "get" or "set."

Table 17-1 lists all of the methods of both the static Date object and, by inheritance, date object instances. The list is impressive — some would say frightening — but there are patterns you should readily observe. Most methods deal with a single component of a date and time value: year, month, date, and so forth. Each block of "get" and "set" methods also has two sets of methods: one for the local date and time conversion of the date stored in the object; one for the actual UTC date stored in the object. After you see the patterns, the list should be more manageable. Unless otherwise noted, a method has been part of the Date object since the first generation of scriptable browsers, and is therefore also supported in newer browsers.

Deciding between using the UTC or local versions of the methods depends on several factors. If the browsers you must support go back to the beginning, you will be stuck with the local versions in any case. But even for newer browsers, activities, such as calculating the number of days between dates or creating a countdown timer for a quiz, won't care which set you use, but you must use the same set for all calculations. If you start mixing local and UTC versions of date methods, you'll be destined to get wrong answers. The UTC versions come in most handy when your date calculations must take into account the time zone of the client machine compared to some absolute in another time zone — calculating the time remaining to the chiming of Big Ben signifying the start of the New Year in London.

JavaScript maintains its date information in the form of a count of milliseconds (thousandths of a second) starting from January 1, 1970, in the GMT (UTC) time zone. Dates before that starting point are stored as negative values (but see the section on bugs and gremlins later in this chapter). Regardless of the country you are in or the date and time formats specified for your computer, the millisecond is the JavaScript universal measure of time. Any calculations that involve adding or subtracting times and dates should be performed in the millisecond values to ensure accuracy. Therefore, though you may never display the milliseconds value in a field or dialog box, your scripts will probably work with them from time to time in variables. To derive the millisecond equivalent for any date and time stored in a date object, use the dateObj.getTime() method, as in

```
var startDate = new Date();
var started = startDate.getTime();
```

TABLE 17-1

Date Object Methods

Method	Value Range	Description
`dateObj.getFullYear()`	1970–...	Specified year (NN4+, Moz1+, IE3+)
`dateObj.getYear()`	70–...	(See text of chapter)
`dateObj.getMonth()`	0–11	Month within the year (January = 0)
`dateObj.getDate()`	1–31	Date within the month
`dateObj.getDay()`	0–6	Day of week (Sunday = 0)
`dateObj.getHours()`	0–23	Hour of the day in 24-hour time
`dateObj.getMinutes()`	0–59	Minute of the specified hour
`dateObj.getSeconds()`	0–59	Second within the specified minute
`dateObj.getTime()`	0–...	Milliseconds since 1/1/70 00:00:00 GMT
`dateObj.getMilliseconds()`	0-999	Milliseconds since the previous full second (NN4+, Moz1+, IE3+)
`dateObj.getUTCFullYear()`	1970–...	Specified UTC year (NN4+, Moz1+, IE3+)
`dateObj.getUTCMonth()`	0–11	UTC month within the year (January = 0) (NN4+, Moz1+, IE3+)
`dateObj.getUTCDate()`	1–31	UTC date within the month (NN4+, Moz1+, IE3+)
`dateObj.getUTCDay()`	0–6	UTC day of week (Sunday = 0) (NN4+, Moz1+, IE3+)
`dateObj.getUTCHours()`	0–23	UTC hour of the day in 24-hour time (NN4+, Moz1+, IE3+)
`dateObj.getUTCMinutes()`	0–59	UTC minute of the specified hour (NN4+, Moz1+, IE3+)
`dateObj.getUTCSeconds()`	0–59	UTC second within the specified minute (NN4+, Moz1+, IE3+)
`dateObj.getUTCMilliseconds()`	0-999	UTC milliseconds since the previous full second (NN4+, Moz1+, IE3+)
`dateObj.setYear(`*`val`*`)`	1970–...	Be safe: always specify a four-digit year
`dateObj.setFullYear(`*`val`*`)`	1970–...	Specified year (NN4+, Moz1+, IE3+)
`dateObj.setMonth(`*`val`*`)`	0–11	Month within the year (January = 0)
`dateObj.setDate(`*`val`*`)`	1–31	Date within the month

Method	Value Range	Description
`dateObj.setDay(val)`	0–6	Day of week (Sunday = 0)
`dateObj.setHours(val)`	0–23	Hour of the day in 24-hour time
`dateObj.setMinutes(val)`	0–59	Minute of the specified hour
`dateObj.setSeconds(val)`	0–59	Second within the specified minute
`dateObj.setMilliseconds(val)`	0-999	Milliseconds since the previous full second (NN4+, Moz1+, IE3+)
`dateObj.setTime(val)`	0–...	Milliseconds since 1/1/70 00:00:00 GMT
`dateObj.setUTCFullYear(val)`	1970–...	Specified UTC year (NN4+, Moz1+, IE3+)
`dateObj.setUTCMonth(val)`	0–11	UTC month within the year (January = 0) (NN4+, Moz1+, IE3+)
`dateObj.setUTCDate(val)`	1–31	UTC date within the month (NN4+, Moz1+, IE3+)
`dateObj.setUTCDay(val)`	0–6	UTC day of week (Sunday = 0) (NN4+, Moz1+, IE3+)
`dateObj.setUTCHours(val)`	0–23	UTC hour of the day in 24-hour time (NN4+, Moz1+, IE3+)
`dateObj.setUTCMinutes(val)`	0–59	UTC minute of the specified hour (NN4+, Moz1+, IE3+)
`dateObj.setUTCSeconds(val)`	0–59	UTC second within the specified minute (NN4+, Moz1+, IE3+)
`dateObj.setUTCMilliseconds(val)`	0-999	UTC milliseconds since the previous full second (NN4+, Moz1+, IE3+)
`dateObj.getTimezoneOffset()`	0–...	Minutes offset from GMT/UTC
`dateObj.toDateString()`		Date-only string in a format determined by browser (WinIE5.5+)
`dateObj.toGMTString()`		Date/time string in universal format
`dateObj.toLocaleDateString()`		Date-only string in your system's localized format (NN6+, Moz1+, WinIE5.5+)
`dateObj.toLocaleString()`		Date/time string in your system's localized format
`dateObj.toLocaleTimeString()`		Time-only string in your system's localized format (NN6+, Moz1+, WinIE5.5+)

continued

TABLE 17-1 *(continued)*

Method	Value Range	Description
`dateObj.toString()`		Date/time string in a format determined by browser
`dateObj.toTimeString()`		Time-only string in a format determined by browser (WinIE5.5+)
`dateObj.toUTCString()`		Date/time string in universal format (NN4+, Moz1+, IE3+)
`Date.parse("`*`dateString`*`")`		Converts string date to milliseconds integer
`Date.UTC(`*`date values`*`)`		Converts GMT string date to milliseconds integer

Although the method has the word "time" in its name, the fact that the value is the total number of milliseconds from January 1, 1970, means the value also conveys a date.

Other date object get methods read a specific component of the date or time. You have to exercise some care here, because some values begin counting with 0 when you may not expect it. For example, January is month 0 in JavaScript's scheme; December is month 11. Hours, minutes, and seconds all begin with 0, which, in the end, is logical. Calendar dates, however, use the actual number that would show up on the wall calendar: the first day of the month is date value 1. For the twentieth-century years, the year value is whatever the actual year number is, minus 1900. For 1996, that means the year value is 96. But for years before 1900 and after 1999, JavaScript uses a different formula, showing the full year value. This means you have to check whether a year value is less than 100 and add 1900 to it before displaying that year.

```
var today = new Date();
var thisYear = today.getYear();
   if (thisYear < 100)
   {
      thisYear += 1900;
   }
```

This assumes, of course, you won't be working with years before A.D. 100. If you can assume that your audience is using a modern browser, which is quite likely, use only the `getFullYear()` method. This method returns the complete set of year digits from all ranges.

To adjust any one of the elements of a date value, use the corresponding set method in an assignment statement. If the new value forces the adjustment of other elements, JavaScript takes care of that. For example, consider the following sequence and how some values are changed for us:

```
myBirthday = new Date("July 4, 1776");
result = myBirthday.getDay(); // result = 4, a Thursday
myBirthday.setYear(1777);     // bump up to next year
result = myBirthday.getDay(); // result = 5, a Friday
```

Because the same date in the following year is on a different day, JavaScript tracks that for you.

Accommodating time zones

Understanding the `dateObj.getTimezoneOffset()` method involves both your operating system's time control panel setting and an internationally recognized (in computerdom, anyway) format for representing dates and times. If you have ignored the control panel stuff about setting your local time zone, the values you get for this property may be off for most dates and times. In the eastern part of North America, for instance, the eastern standard time zone is five hours earlier than Greenwich Mean Time. With the `getTimezoneOffset()` method producing a value of minutes' difference between GMT and the PC's time zone, the five hours difference of eastern standard time is rendered as a value of 300 minutes. On the Windows platform, the value automatically changes to reflect changes in daylight saving time in the user's area (if applicable). Offsets to the east of GMT (to the date line) are expressed as negative values.

Dates as strings

When you generate a date object, JavaScript automatically applies the `toString()` method to the object if you attempt to display that date either in a page or alert box. The format of this string varies with browser and operating system platforms. For example, in IE8 for Windows XP, the string is in the following format:

```
Sun Dec 5 16:47:20 PST 2010
```

But in Firefox 3 for Windows XP, the string is

```
Sun Dec 05 2010 16:47:20 GMT-0800 (Pacific Standard Time)
```

Other browsers return their own variations on the string. The point is not to rely on a specific format and character location of this string for the components of dates. Use the date object methods to read date object components.

JavaScript does, however, provide two methods that return the date object in more constant string formats. One, `dateObj.toGMTString()`, converts the date and time to the GMT equivalent on the way to the variable that you use to store the extracted data. Here is what such data looks like:

```
Mon, 06 Dec 2010 00:47:20 GMT
```

If you're not familiar with the workings of GMT and how such conversions can present unexpected dates, exercise great care in testing your application. A quarter to five on a Sunday afternoon on the North American west coast is well after midnight Monday morning at the Royal Observatory in Greenwich, London.

If time zone conversions make your head hurt, you can use the second string method, `dateObj.toLocaleString()`. In Firefox for North American Windows users, the returned value can look like this:

```
Sunday, December 05, 2010 4:47:20 PM
```

Ever since IE5.5 and NN6/Moz1, you can also have JavaScript convert a date object to just the date or time portions in a nicely formatted version. The best pair of methods for this are `toLocaleDateString()` and `toLocaleTimeString()`, because these methods return values that make the most sense to the user, based on the localization settings of the user's operating system and browser.

Friendly date formats for older browsers

If you don't have the luxury of writing script code only for modern browsers, you can create your own formatting function to do the job for a wide range of browsers. Listing 17-1 demonstrates one way of creating this kind of string from a date object (in a form that will work back to version 4 browsers).

Note

The function to assign event handlers throughout the code in this chapter and much of the book is `addEvent()`**, a cross-browser event handler explained in detail in Chapter 32, "Event Objects."**

The `addEvent()` **function is part of the script file** `jsb-global.js` **located on the accompanying CD-ROM in the** `Content/` **folder, where it is accessible to all chapters' scripts.** ■

LISTING 17-1

Creating a Friendly Date String

HTML: jsb-17-01.html

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Date String Maker</title>
      <script type="text/javascript" src="../jsb-global.js"></script>
      <script type="text/javascript" src="jsb-17-01.js"></script>
   </head>
   <body>
      <h1>Date String Maker</h1>
      <p id="output">Today's date</p>
   </body>
</html>
```

JavaScript: jsb-17-01.js

```
// initialize when the page has loaded
addEvent(window, 'load', initialize);

monthNames = ["January", "February", "March", "April", "May", "June", "July",
   "August", "September", "October", "November", "December"];
dayNames = ["Sunday", "Monday", "Tuesday", "Wednesday", "Thursday", "Friday",
   "Saturday"];

function initialize()
{
   // do this only if the browser can handle DOM methods
   if (document.getElementById)
   {
      // point to the paragraph to contain the date
      oOutput = document.getElementById('output');

         // if it exists, replace its contents with the date
```

```
        if (oOutput)
        {
           // remove all child nodes from the paragraph
           while (oOutput.firstChild)
           {
              oOutput.removeChild(oOutput.firstChild);
           }

           // create a text node with the date
           var sDate = customDateString(new Date());
           var oNewText = document.createTextNode(sDate);

           // insert that content into the paragraph
           oOutput.appendChild(oNewText);
        }
   }
}

// generate a formatted date
function customDateString(oDate)
{
   var theDay = dayNames[oDate.getDay()];
   var theMonth = monthNames[oDate.getMonth()];
   var theYear = oDate.getFullYear();
   return theDay + ", " + theMonth + " " + oDate.getDate() + ", " + theYear;
}
```

Assuming the user has the PC's clock set correctly (a big assumption), the date appearing just below the opening headline is the current date — making it appear as though the document had been updated today. The downside to this approach (as opposed to the newer `toLocaleDateString()` method) is that international users are forced to view dates in the format you design, which may be different from their local custom.

More conversions

The last two methods shown in Listing 17-1 are methods of the static `Date` object. These utility methods convert dates from string or numeric forms into millisecond values of those dates. The primary beneficiary of these actions is the `dateObj.setTime()` method, which requires a millisecond measure of a date as a parameter. You use this method to throw an entirely different date into an existing `date` object.

`Date.parse()` accepts as a parameter date strings similar to the ones you've seen in this section, including the internationally approved version. `Date.UTC()`, on the other hand, requires the comma-delimited list of values (in proper order: `yy,mm,dd,hh,mm,ss`) in the GMT zone. The `Date.UTC()` method gives you a backward-compatible way to hard-code a GMT time (you can do the same in version 4 browsers via the UTC methods). The following is an example that creates a new date object for 6 p.m. on October 1, 2011, GMT in WinIE8:

```
var newObj = new Date(Date.UTC(2011, 9, 1, 18, 0, 0));
result = newObj.toString();    // result = "Sat Oct 1 11:00:00 PDT2011"
```

The second statement returns a value in a local time zone, because all non-UTC methods automatically convert the GMT time stored in the object to the client's local time.

Date and time arithmetic

You may need to perform some math with dates for any number of reasons. Perhaps you need to calculate a date at some fixed number of days or weeks in the future or figure out the number of days between two dates. When calculations of these types are required, remember the *lingua franca* of JavaScript date values: milliseconds.

What you may need to do in your date-intensive scripts is establish some variable values representing the number of milliseconds for minutes, hours, days, or weeks, and then use those variables in your calculations. Here is an example that establishes some practical variable values, building on each other:

```
var oneMinute = 60 * 1000;
var oneHour = oneMinute * 60;
var oneDay = oneHour * 24;
var oneWeek = oneDay * 7;
```

With these values established in a script, I can use one to calculate the date one week from today:

```
var targetDate = new Date();
var dateInMs = targetDate.getTime();
dateInMs += oneWeek;
targetDate.setTime(dateInMs);
```

Another example uses components of a date object to assist in deciding what kind of greeting message to place in a document, based on the local time of the user's PC clock. Listing 17-2 adds to the scripting from Listing 17-1, bringing some quasi-intelligence to the proceedings.

LISTING 17-2

A Dynamic Welcome Message

HTML: jsb-17-02.html

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Date String Maker</title>
      <script type="text/javascript" src="../jsb-global.js"></script>
      <script type="text/javascript" src="jsb-17-02.js"></script>
   </head>
   <body>
      <h1>Welcome!</h1>
      <p id="date">Today's date</p>
      <p>We hope you are enjoying the <span id="day-part">day</span>.</p>
   </body>
</html>
```

JavaScript: jsb-17-02.js

```
// initialize when the page has loaded
addEvent(window, 'load', initialize);

monthNames = ["January", "February", "March", "April", "May", "June", "July",
   "August", "September", "October", "November", "December"];
dayNames = ["Sunday", "Monday", "Tuesday", "Wednesday", "Thursday", "Friday",
   "Saturday"];

function initialize()
{
   // do this only if the browser can handle DOM methods
   if (document.getElementById)
   {
      // point to the target paragraphs
      var oDateDisplay = document.getElementById('date');
      var oGreeting = document.getElementById('greeting');

         // if they exist, plug in new content
         if (oDateDisplay && oGreeting)
         {
            // plug in date
            var oNow = new Date();
            var sDate = customDateString(oNow);
            replaceTextContent(oDateDisplay, sDate);

            // plug day-part into greeting
            var sDayPart = dayPart(oNow);
            replaceTextContent(oGreeting, sDayPart);
         }
   }
}

// generate a formatted date
function customDateString(oDate)
{
   var theDay = dayNames[oDate.getDay()];
   var theMonth = monthNames[oDate.getMonth()];
   var theYear = oDate.getFullYear();
   return theDay + ", " + theMonth + " " + oDate.getDate() + ", " + theYear;
}

// get the part of the day
function dayPart(oDate)
{
   var theHour = oDate.getHours();
   if (theHour < 6 )
      return "wee hours";
   if (theHour < 12)
      return "morning";
   if (theHour < 18)
```

continued

LISTING 17-2 *(continued)*

```
      return "afternoon";
   return "evening";
}

// replaces the text contents of a page element
function replaceTextContent(oElement, sContent)
{
   // if the object exists
   if (oElement)
   {
      // remove all child nodes
      while (oElement.firstChild)
      {
         oElement.removeChild(oElement.firstChild);
      }

      // create a text node with the new content
      var oNewText = document.createTextNode(sContent);

      // insert that content
      oElement.appendChild(oNewText);
   }
}
```

The script divides the day into four parts and presents a different greeting for each part of the day. The greeting that plays is based, simply enough, on the hour element of a date object representing the time the page is loaded into the browser. Because this greeting is embedded in the page, the greeting does not change no matter how long the user displays the page, but it will update each time the page is reloaded.

Counting the days . . .

You may find one or two more date arithmetic applications useful. One displays the number of shopping days left until Christmas (in the user's time zone); the other is a countdown timer to the start of the year 2100.

Listing 17-3 demonstrates how to calculate the number of days between the current day and some fixed date in the future. The assumption in this application is that all calculations take place in the user's time zone. The example shows the display of the number of shopping days before the next Christmas day (December 25). The basic operation entails converting the current date and the next December 25 to milliseconds, calculating the number of days represented by the difference in milliseconds. If you let the millisecond values represent the dates, JavaScript automatically takes care of leap years.

The only somewhat tricky part is setting the year of the next Christmas day correctly. You can't just slap the fixed date with the current year, because if the program is run on December 26, the year of the next Christmas must be incremented by one. That's why the constructor for the Christmas date

object doesn't supply a fixed date as its parameters, but rather, sets individual components of the object.

Also, note that while `setDate()` accepts a counting number 1-31, `setMonth()` accepts a whole number 0-11, making December month 11 for that method call.

LISTING 17-3

How Many Days Until Christmas

HTML: jsb-17-03.html

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Christmas Countdown</title>
      <script type="text/javascript" src="../jsb-global.js"></script>
      <script type="text/javascript" src="jsb-17-03.js"></script>
   </head>
   <body>
      <h1>Christmas Countdown</h1>
      <p>You have <em id="days-left">too few</em> shopping
         <span id="day-word">days</span> until Christmas.</p>
   </body>
</html>
```

JavaScript: jsb-17-03.js

```
// initialize when the page has loaded
addEvent(window, 'load', initialize);

function initialize()
{
   // do this only if the browser can handle DOM methods
   if (document.getElementById)
   {
      // point to the target paragraphs
      var oOutput = document.getElementById('days-left');
      var oDayWord = document.getElementById('day-word');

         // if they exist, plug in new content
         if (oOutput && oDayWord)
         {
            // plug in days left
            var sDaysLeft = getDaysUntilXmas();
            replaceTextContent(oOutput, sDaysLeft);
```

continued

LISTING 17-3 *(continued)*

```
            // pluralize day if not one
            var sDayWord = (sDaysLeft == 1) ? 'day' : 'days';
            replaceTextContent(oDayWord, sDayWord);
         }
      }
   }

// calculate the number of days till next Christmas
function getDaysUntilXmas()
{
   var oneMinute = 60 * 1000;
   var oneHour = oneMinute * 60;
   var oneDay = oneHour * 24;

   var today = new Date();
   var nextXmas = new Date();
   nextXmas.setMonth(11);
   nextXmas.setDate(25);
      if (today.getMonth() == 11 && today.getDate() > 25)
      {
         nextXmas.setFullYear(nextXmas.getFullYear() + 1);
      }
   var diff = nextXmas.getTime() - today.getTime();
   diff = Math.floor(diff/oneDay);
   return diff;
}

// replaces the text contents of a page element
function replaceTextContent(oElement, sContent)
{
   [see listing 17-2]
}
```

The second variation on calculating the amount of time before a certain event takes time zones into account. For this demonstration, the page is supposed to display a countdown timer to the precise moment when the flame for the 2008 Summer Games in Beijing is to be lit. That event takes place in a time zone that may be different from that of the page's viewer, so the countdown timer must calculate the time difference accordingly.

Listing 17-4 shows a simplified version that simply displays the ticking timer in a text field. The output, of course, could be customized in any number of ways, depending on the amount of dynamic HTML you want to employ on a page. The time of the lighting for this demo is set at 11:00 GMT on August 8, 2008 (the date is certainly accurate, but the officials may set a different time closer to the actual event).

Because this application is implemented as a live ticking clock, the code starts by setting some global variables that should be calculated only once so that the function that gets invoked repeatedly has a minimum of calculating to do (to be more efficient). The `Date.UTC()` method

provides the target time and date in standard time. The `getTimeUntil()` function accepts a millisecond value (as provided by the `targetDate` variable) and calculates the difference between the target date and the actual internal millisecond value of the client's PC clock.

The core of the `getCountDown()` function peels off the number of whole days, hours, minutes, and seconds from the total number of milliseconds difference between now and the target date. Notice that each chunk is subtracted from the total so that the next-smaller chunk can be calculated from the leftover milliseconds.

One extra touch on this page is a display of the local date and time of the actual event.

LISTING 17-4

Summer Games Countdown

HTML: jsb-17-04.html

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Time Since the First Moon Landing</title>
      <script type="text/javascript" src="../jsb-global.js"></script>
      <script type="text/javascript" src="jsb-17-04.js"></script>
   </head>
   <body>
      <h1>Time Since the First Moon Landing</h1>
      <p>It is now <span id="now">the present</span> in your timezone.</p>
      <p><span id="time-since">Many days</span> have passed since the first
         moon landing on <span id="past-day">a day long ago</span>.</p>
   </body>
</html>
```

JavaScript: jsb-17-04.js

```
// initialize when the page has loaded
addEvent(window, 'load', initialize);

function initialize()
{
   // do this only if the browser can handle DOM methods
   if (document.getElementById)
   {
      // point to the target elements
      oOutputNow = document.getElementById('now');
      oOutputTimeSince = document.getElementById('time-since');
      var oOutputPastDay = document.getElementById('past-day');

         // if they exist, plug in new content
```

continued

LISTING 17-4 *(continued)*

```
            if (oOutputNow && oOutputTimeSince && oOutputPastDay)
            {
               // globals -- calculate only once
               // set target date to 20:17 UTC on July 20, 1969
               targetDate = Date.UTC(1969, 6, 20, 20, 17, 0, 0);
               oneSecond = 1000;
               oneMinute = oneSecond * 60;
               oneHour = oneMinute * 60;
               oneDay = oneHour * 24;

               // plug in the past date
               var sPastDay = (new Date(targetDate)).toLocaleString();
               replaceTextContent(oOutputPastDay, sPastDay);

               // plug in current time & time since
               updateCounter();
            }
      }
}

// timer loop to display changing values
function updateCounter()
{
   // plug in current time
   var sNow = (new Date()).toLocaleString();
   replaceTextContent(oOutputNow, sNow);

   // plug in days since
   var sTimeSince = formatTimeSince();
   replaceTextContent(oOutputTimeSince, sTimeSince);

   setTimeout("updateCounter()", 1000);
}

// format elapsed time
function formatTimeSince()
{
   var ms = getTimeDiff(targetDate);
   var days, hrs, mins, secs;

   days = Math.floor(ms/oneDay);
   output = days + " Day";
      if (days != 1) output += 's';

   ms -= oneDay * days;
   hrs = Math.floor(ms/oneHour);
   output += ", " + hrs + " Hour";
      if (hrs != 1) output += 's';
```

```
   ms -= oneHour * hrs;
   mins = Math.floor(ms/oneMinute);
   output += ", " + mins + " Minute";
      if (mins != 1) output += 's';

   ms -= oneMinute * mins;
   secs = Math.floor(ms/oneSecond);
   output += ", and " + secs + " Second";
      if (secs != 1) output += 's';

   return output;
}

// get the difference between two datetimes in milliseconds
function getTimeDiff(targetMS)
{
   var today = new Date();
   var diff = Math.abs(today.valueOf() - targetMS);
   return Math.floor(diff);
}

// replaces the text contents of a page element
function replaceTextContent(oElement, sContent)
{
   [see listing 17-2]
}
```

Early browser date bugs and gremlins

Each new browser generation improves the stability and reliability of scripted date objects. For example, Netscape Navigator 2 had so many bugs and crash problems that it made scripting complex world-time applications for this browser impossible. NN3 improved matters a bit, but some glaring problems still existed. And lest you think I'm picking on Netscape, rest assured that early versions of Internet Explorer also had plenty of date and time problems. IE3 couldn't handle dates before January 1, 1970 (GMT), and also completely miscalculated the time zone offset, following the erroneous pattern of NN2. Bottom line — you're asking for trouble if you must work extensively with dates and times while supporting legacy browsers.

You should be aware of one more discrepancy between Mac and Windows versions of Navigator through Version 4. In Windows, if you generate a date object for a date in another part of the year, the browser sets the time zone offset for that object according to the time zone setting for that time of year. On the Mac, the current setting of the control panel governs whether the normal or daylight savings time offset is applied to the date, regardless of the actual date within the year. This discrepancy affects Navigator 3 and 4 and can throw off calculations from other parts of the year by one hour.

It may sound as though the road to `Date` object scripting is filled with land mines. Although date and time scripting is far from hassle free, you can put it to good use with careful planning and a lot of testing. Better still, if you make the plausible assumption that the majority of users have a modern browser (WinIE6+, NN6+, Moz1+, FF1+, Cam1+, Safari1+, etc.), then things should go very smoothly.

Validating Date Entries in Forms

Given the bug horror stories in the previous section, you may wonder how you can ever perform data entry validation for dates in forms. The problem is not so much in the calculations as it is in the wide variety of acceptable date formats around the world. No matter how well you instruct users to enter dates in a particular format, many will follow their own habits and conventions. Moreover, how can you know whether an entry of 03/04/2010 is the North American March 4, 2010 or the European April 3, 2010? The answer: you can't.

My recommendation is to divide a date field into three components: month, day, and year. Let the user enter values into each field and validate each field individually for its valid range. Listing 17-5 shows an example of how this is done. The page includes a form that is to be validated before it is submitted. Each component field does its own range checking on-the-fly as the user enters values. But because this kind of validation can be defeated, the page includes one further check triggered by the form's `onsubmit` event handler. If any field is out of whack, the form submission is cancelled.

LISTING 17-5

Date Validation in a Form

HTML: jsb-17-05.html

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Date Entry Validation</title>
      <script type="text/javascript" src="../jsb-global.js"></script>
      <script type="text/javascript" src="jsb-17-05.js"></script>
   </head>
   <body>
      <h1>Date Entry Validation</h1>
      <form id="birthdate" action="example.php" method="post">
         <p>Please enter your birthdate...</p>
         <p>
            <label for="month">Month:</label>
            <input type="text" id="month" name="month" value="1" size="2"
               maxlength="2">

            <label for="day">Day:</label>
            <input type="text" id="day" name="day" value="1" size="2"
               maxlength="2">

            <label for="year">Year:</label>
            <input type="text" id="year" name="year" value="1900" size="4"
               maxlength="4">
         </p>
         <p>Thank you for entering <span id="fullDate">your birthdate</span>.</p>
         <p>
            <input type="submit">
```

```
            <input type="Reset">
         </p>
      </form>
   </body>
</html>
```

JavaScript: jsb-17-05.js

```
// initialize when the page has loaded
addEvent(window, 'load', initialize);

function initialize()
{
   // do this only if the browser can handle DOM methods
   if (document.getElementById)
   {
      // point to the target elements
      oForm = document.getElementById('birthdate');
      oMonth = document.getElementById('month');
      oDay = document.getElementById('day');
      oYear = document.getElementById('year');
      oFullDate = document.getElementById('fullDate');

         // if they exist, plug in new content
         if (oForm && oMonth && oDay && oYear && oFullDate)
         {
            // apply behaviors to form elements
            oForm.onsubmit = checkForm;
            oMonth.onchange = function() { return validateMonth(oMonth); }
            oDay.onchange = function() { return validateDay(oDay); }
            oYear.onchange = function() { return validateYear(oYear); }
         }
   }
}

function checkForm(evt)
{
      if (validateMonth(oMonth))
      {
         if (validateDay(oDay))
         {
            if (validateYear(oYear))
            {
               // do nothing
            }
         }
      }
   return false;
}

// validate input month
function validateMonth(oField, bBypassUpdate)
```

continued

LISTING 17-5 *(continued)*

```
{
   var sInput = oField.value;

      if (isEmpty(sInput))
      {
         alert("Be sure to enter a month value.");
         selectField(oField);
         return false;
      }
      else
      {
         sInput = parseInt(oField.value, 10);
            if (isNaN(sInput))
            {
               alert("Entries must be numbers only.");
               selectField(oField);
               return false;
            }
            else
            {
               if (!inRange(sInput,1,12))
               {
                  alert("Enter a number between 1 (January) and 12 (December).");
                  selectField(oField);
                  return false;
               }
            }
      }

      if (!bBypassUpdate)
      {
         calcDate();
      }
   return true;
}

// validate input day
function validateDay(oField)
{
   var sInput = oField.value;

   if (isEmpty(sInput))
   {
      alert("Be sure to enter a day value.");
      selectField(oField);
      return false;
   }
   else
   {
      sInput = parseInt(oField.value, 10);
```

```
      if (isNaN(sInput))
      {
         alert("Entries must be numbers only.");
         selectField(oField);
         return false;
      }
      else
      {
         var monthField = oMonth;

            if (!validateMonth(monthField, true))
               return false;

         var iMonthVal = parseInt(monthField.value, 10);
         var iMonthMax = new Array(31,31,29,31,30,31,30,31,31,30,31,30,31);
         var iTop = iMonthMax[iMonthVal];

            if (!inRange(sInput,1,iTop))
            {
               alert("Enter a number between 1 and " + iTop + ".");
               selectField(oField);
               return false;
            }
      }
   }
   calcDate();
   return true;
}

// validate input year
function validateYear(oField)
{
   var iCurrentYear = (new Date).getFullYear();

   var sInput = oField.value;
      if (isEmpty(sInput))
      {
         alert("Be sure to enter a year value.");
         selectField(oField);
         return false;
      }
      else
      {
         sInput = parseInt(oField.value, 10);
            if (isNaN(sInput))
            {
               alert("Entries must be numbers only.");
               selectField(oField);
               return false;
            }
            else
            {
               if (!inRange(sInput, 1900, iCurrentYear))
```

continued

LISTING 17-5 *(continued)*

```
                {
                   alert("Enter a number between 1900 and " + iCurrentYear + ".");
                   selectField(oField);
                   return false;
                }
             }
         }
   calcDate();
   return true;
}

// place the edit cursor in the requested field
function selectField(oField)
{
   oField.focus();
   oField.select();
}

// format a complete date
function calcDate()
{
   var mm = parseInt(oMonth.value, 10);
   var dd = parseInt(oDay.value, 10);
   var yy = parseInt(oYear.value, 10);
   sDate = mm + "/" + dd + "/" + yy;
   replaceTextContent(oFullDate, sDate);
}

// **BEGIN GENERIC VALIDATION FUNCTIONS**
// general purpose function to see if an input value has been entered at all
function isEmpty(sInput)
{
      if (sInput == "" || sInput == null)
      {
         return true;
      }
   return false;
}

// determine if value is in acceptable range
function inRange(sInput, iLow, iHigh)
{
   var num = parseInt(sInput, 10);

      if (num < iLow || num > iHigh)
      {
         return false;
      }
   return true;
}
// **END GENERIC VALIDATION FUNCTIONS**
```

As a general rule, JavaScript validation is only the first line of defense and must be backed up by server-side validation using PHP or another server scripting language. Because some users and user agents are going to be running without JavaScript, the server-side program will need to perform the validation all over again, but for those using JavaScript the validation feedback will be immediate.

Not every date entry validation must be divided in this way. For example, an intranet application can be more demanding in the way users are to enter data. Therefore, you can have a single field for date entry, but the parsing required for such a validation is quite different from that shown in Listing 17-5. See Chapter 46 on the CD-ROM for an example of such a one-field date validation routine.

Data entry validation is also an area of scripting that can benefit from asynchronous JavaScript, also known as Ajax, when it's necessary to look up values on the server to validate input. However, the advantages should be considered carefully since every Ajax operation requires a round-trip to the server and might not be any faster than submitting the whole page.

Cross-Reference

Check out Chapter 39, "Ajax, E4X, and XML," for more on how Ajax can be used to carry out dynamic data entry validation. ■

CHAPTER 18

The Array Object

IN THIS CHAPTER

Working with ordered collections of data

Simulating multidimensional arrays

Manipulating information stored in an array

New additions to JavaScript array handling

An array is one of the major JavaScript structures provided for storing and manipulating ordered collections of data. But unlike some other programming languages, JavaScript's arrays are very forgiving as to the kind of data you store in each cell or entry of the array. This allows, for example, an array of arrays, providing the equivalent of multidimensional arrays customized to the kind of data your application needs.

If you have not done a lot of programming in the past, the notion of arrays may seem like an advanced topic. But if you ignore their capabilities, you set yourself up for a harder job when implementing many kinds of tasks. Whenever we approach a script, one of our first thoughts is about the data being controlled by the application and whether handling it as an array will offer some shortcuts for creating the document and handling interactivity with the user.

We hope that by the end of this chapter you will not only be familiar with the properties and methods of JavaScript arrays but you will also begin to look for ways to make arrays work for you.

Structured Data

In programming, an *array* is defined as an ordered collection of data. You can best visualize an array as a table, not much different from a spreadsheet. In JavaScript, arrays are limited to a table holding one column of data, with as many rows as needed to hold your data. As you have seen in many chapters in Part IV, a JavaScript-enabled browser creates a number of internal arrays for the objects in your HTML documents and browser properties. For example, if your document contains five links, the browser maintains a table of those links. You access them by number (with 0 being the first link) in the array syntax: the array name is followed by the index number in square brackets, as in `document.links[0]`, which represents the first link in the document.

For many JavaScript applications, you will want to use an array as an organized warehouse for data that users of your page access, depending on their interaction

with form elements. In the application shown in Chapter 53, "Application: A Lookup Table" on the CD-ROM, for example, we demonstrate an extended version of this usage in a page that lets users search a small table of data for a match between the first three digits of their U.S. Social Security numbers and the state in which they registered with the agency. Arrays are one way JavaScript-enhanced pages can re-create the behavior of more sophisticated server-side applications such as CGI scripts and Java servlets. When the collection of data you embed in the script is no larger than a typical `.gif` image file, the user won't experience significant delays in loading your page; yet he or she has the full power of your small database collection for instant searching without any calls back to the server. Such database-oriented arrays are important applications of JavaScript for what we call *serverless CGIs*.

As you design an application, look for clues as to potential uses of arrays. If you have a number of objects or data points that interact with scripts the same way, you have a good candidate for array structures. For example, you can assign like names to every text field in a column of an order form. In that sequence, like-named objects are treated as elements of an array. To perform repetitive row calculations down an order form, your scripts can use array syntax to perform all the extensions within a handful of JavaScript statements, rather than perhaps dozens of statements hard-coded to each field name. Chapter 54, "Application: A 'Poor Man's' Order Form" (on the CD-ROM) shows an example of this application.

You can also create arrays that behave like the Java hash table: a lookup table that gets you to a desired data point instantaneously if you know the name associated with the entry.

If you can somehow conceive your data in a table format, an array is in your future.

Creating an Empty Array

Full-fledge array objects in JavaScript go all the way back to NN3 and IE4. It was possible to simulate some array characteristics in even earlier browsers, but since those first-generation browsers have thankfully disappeared from most users' computers, this chapter focuses on the modern array and its hefty powers.

To create a new array object, use the static `Array` object's constructor method. For example:

```
var myArray = new Array();
```

An array object automatically has a `length` property (0 for an empty array).

Should you want to presize the array (for example, preload entries with `null` values), you can specify an initial size as a parameter to the constructor. For example, here is how to create a new array to hold information about a 500-item compact disc collection:

```
var myCDCollection = new Array(500);
```

Unlike with many other programming languages, presizing a JavaScript array does not give you any particular advantage, because you can assign a value to any slot in an array at any time: The `length` property adjusts itself accordingly. For instance, if you assign a value to `myCDCollection[700]`, the array object adjusts its length upward to meet that slot (with the count starting at 0):

```
myCDCollection [700] = "The Smiths/Louder Than Bombs";
collectionSize = myCDCollection.length;    // result = 701
```

Since the count of array elements starts at 0, assigning a value to location 700 results in an array that contains 701 items.

A true array object features a number of methods and the capability to add prototype properties, described later in this chapter.

Populating an Array

Entering data into an array is as simple as creating a series of assignment statements, one for each element of the array. Listing 18-1 generates an array containing a list of the eight planets of the solar system.

LISTING 18-1

Generating and Populating a New Array

```
solarSys = new Array();
solarSys[0] = "Mercury";
solarSys[1] = "Venus";
solarSys[2] = "Earth";
solarSys[3] = "Mars";
solarSys[4] = "Jupiter";
solarSys[5] = "Saturn";
solarSys[6] = "Uranus";
solarSys[7] = "Neptune";
```

This way of populating a single array is a bit tedious when you're writing the code, but after the array is set, it makes accessing collections of information as easy as any array reference:

```
onePlanet = solarSys[4];     // result = "Jupiter"
```

A variant of this method takes advantage of the fact that, because JavaScript arrays are zero-based, the array length property always points past the end of the array at the next item to be filled. The following code produces an array identical to the one above:

```
solarSys = new Array();
solarSys[solarSys.length] = "Mercury";
solarSys[solarSys.length] = "Venus";
solarSys[solarSys.length] = "Earth";
solarSys[solarSys.length] = "Mars";
solarSys[solarSys.length] = "Jupiter";
solarSys[solarSys.length] = "Saturn";
solarSys[solarSys.length] = "Uranus";
solarSys[solarSys.length] = "Neptune";
```

This makes sense if you think about it: a newly created empty array has a length of zero, and the first item to be added to it is `array[0]`. The length is then one and the next item to be added is `array[1]`; and so on.

Assigning array elements like this makes it easy to edit the script later to add or remove elements without having to change all the subsequent subscripts:

```
solarSys = new Array();
solarSys[solarSys.length] = "Mercury";
```

```
solarSys[solarSys.length] = "Venus";
solarSys[solarSys.length] = "Earth";
solarSys[solarSys.length] = "Mars";
solarSys[solarSys.length] = "asteroid belt";
solarSys[solarSys.length] = "Jupiter";
solarSys[solarSys.length] = "Saturn";
solarSys[solarSys.length] = "Uranus";
solarSys[solarSys.length] = "Neptune";
```

A more compact way to create an array is available if you know that the data will be in the desired order (such as the preceding `solarSys` array). Instead of writing a series of assignment statements (as in Listing 18-1), you can create what is called a *dense array* by supplying the data as comma-delimited parameters to the `Array()` constructor:

```
solarSys = new Array("Mercury","Venus","Earth","Mars","Jupiter","Saturn",
   "Uranus","Neptune");
```

The term "dense array" simply means that data is packed into the array, without gaps, starting at index position 0.

The example in Listing 18-1 shows what you may call a vertical collection of data. Each data point contains the same type of data as the other data points — the name of a planet — and the data points appear in the relative order of the planets from the Sun.

JavaScript Array Creation Enhancements

JavaScript provides one more way to create a dense array and also clears up a bug in the way older browsers handled arrays. This improved approach does not require the `Array` object constructor. Instead, JavaScript (as of version 1.2) accepts what is called *literal notation* to generate an array. To demonstrate the difference, the following statement is the regular dense array constructor that works all the way back to NN3:

```
solarSys = new Array("Mercury","Venus","Earth","Mars","Jupiter","Saturn",
   "Uranus","Neptune");
```

While JavaScript 1.2+ fully accepts the preceding syntax, it also accepts the new literal notation:

```
solarSys = ["Mercury","Venus","Earth","Mars","Jupiter","Saturn",
   "Uranus","Neptune"];
```

The square brackets stand in for the call to the `Array` constructor. If you use this streamlined approach to array creation, be aware that it may stop JavaScript execution cold in old browsers.

The bug fix we mentioned has to do with how to treat the earlier dense array constructor if the scripter enters only the numeric value 1 as the parameter — `new Array(1)`. In NN3 and IE4, JavaScript erroneously creates an array of length 1, but that element is `undefined`. For NN4 and all later browsers, the same statement creates that one-element array and places the value 1 in that element.

Deleting Array Entries

You can easily wipe out any data in an array element by setting the value of the array entry to `null` or an empty string. But until the `delete` operator came along in version 4 browsers, you could not completely remove an element.

Deleting an array element eliminates the index from the list of accessible index values but does not reduce the array's length, as in the following sequence of statements:

```
myArray.length    // result: 5
delete myArray[2]
myArray.length    // result: 5
myArray[2]        // result: undefined
```

The process of deleting an array entry does not necessarily release memory occupied by that data. The JavaScript interpreter's internal garbage collection mechanism (beyond the reach of scripters) is supposed to take care of such activity. See the `delete` operator in Chapter 22, "JavaScript Operators," for further details.

If you want tighter control over the removal of array elements, you might want to consider using the `splice()` method, which is supported in modern browsers. The `splice()` method can be used on any array and lets you remove an item (or sequence of items) from the array — causing the array's length to adjust to the new item count. See the `splice()` method later in this chapter.

Parallel Arrays

Using an array to hold data is frequently desirable so that a script can do a lookup to see if a particular value is in the array (perhaps verifying that a value typed into a text box by the user is permissible); however, even more valuable is if, upon finding a match, a script can look up some related information in another array. One way to accomplish this is with two or more parallel arrays: the same indexed slot of each array contains related information.

Consider the following three arrays:

```
var regionalOffices = ["New York", "Chicago", "Houston", "Portland"];
var regionalManagers = ["Shirley Smith", "Todd Gaston",
   "Leslie Jones", "Harold Zoot"];
var regOfficeQuotas = [300000, 250000, 350000, 225000];
```

The assumption for these statements is that Shirley Smith is the regional manager out of the New York office, and her office's quota is 300,000. This represents the data that is included with the document, perhaps retrieved by a server-side program that gets the latest data from a SQL database and embeds the data in the form of array constructors. Listing 18-2 shows how this data appears in a simple page that looks up the manager name and quota values for whichever office is chosen in the `select` element. The order of the items in the list of `select` is not accidental: the order is identical to the order of the array for the convenience of the lookup script.

Lookup action in Listing 18-2 is performed by the `getData()` function. Because the index values of the options inside the `select` element match those of the parallel arrays index values, the `selectedIndex` property of the `select` element makes a convenient way to get directly at the corresponding data in other arrays.

Cross-Reference

The property assignment event-handling technique employed throughout the code in this chapter and much of the book is `addEvent()`, a cross-browser event handler explained in detail in Chapter 32, "Event Objects."

The `addEvent()` function is part of the script file `jsb-global.js` located on the accompanying CD-ROM in the `Content/` folder where it is accessible to all chapters' scripts. ■

LISTING 18-2

A Simple Parallel Array Lookup

HTML: jsb-18-02.html

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Parallel Array Lookup</title>
      <script type="text/javascript" src="../jsb-global.js"></script>
      <script type="text/javascript" src="jsb-18-02.js"></script>
   </head>
   <body>
      <h1>Parallel Array Lookup</h1>
      <form id="officeData" action="" method="post">
         <p>
            <label for="offices">Select a regional office:</label>
            <select id="offices" name="offices">
            </select>
         </p>
         <p>
            <label for="manager">The manager is:</label>
            <input type="text" id="manager" name="manager" size="35" />
         </p>
         <p>
            <label for="quota">The office quota is:</label>
            <input type="text" id="quota" name="quota" size="8" />
         </p>
      </form>
   </body>
</html>
```

JavaScript: jsb-18-02.js

```
// initialize when the page has loaded
addEvent(window, 'load', initialize);
```

```
function initialize()
{
   // do this only if the browser can handle DOM methods
   if (document.getElementById)
   {
      // set up the data in global variables (by omitting 'var')
      aRegionalOffices = ["New York", "Chicago", "Houston", "Portland"];
      aRegionalManagers = ["Shirley Smith", "Todd Gaston", "Leslie Jones",
                           "Harold Zoot"];
      aRegOfficeQuotas = [300000, 250000, 350000, 225000];

      // point to the critical input fields & save in global variables
      oSelect = document.getElementById('offices');
      oManager = document.getElementById('manager');
      oQuota = document.getElementById('quota');

         // if they all exist...
         if (oSelect && oManager && oQuota)
         {
            // build the drop-down list of regional offices
            for (var i = 0; i < aRegionalOffices.length; i++)
            {
               oSelect.options[i] = new Option(aRegionalOffices[i]);
            }

            // set the onchange behavior
            addEvent(oSelect, 'change', getData);
         }
      // plug in data for the default select option
      getData();
   }
}


// when a new option is selected, do the lookup into parallel arrays
function getData(evt)
{
   // get the offset of the selected option
   var index = oSelect.selectedIndex;

   // get data from the same offset in the parallel arrays
   oManager.value = aRegionalManagers[index];
   oQuota.value = aRegOfficeQuotas[index];
}
```

On the other hand, if the content to be looked up is typed into a text box by the user, you have to loop through one of the arrays to get the matching index. Listing 18-3 is a variation of Listing 18-2, but instead of the `select` element, a text field asks users to type in the name of the region. Assuming that users will always spell the input correctly (admittedly an outrageous assumption), the version of `getData()` in Listing 18-3 performs actions that more closely resemble what you may think a "lookup" should be doing: looking for a match in one array, and displaying corresponding results from the parallel arrays. The `for` loop iterates through items in the `aRegionalOffices` array. An

if condition compares all uppercase versions of both the input and each array entry. If there is a match, the for loop breaks, with the value of i still pointing to the matching index value. Outside the for loop, another if condition makes sure that the index value has not reached the length of the array, which means that no match is found. Only when the value of i points to one of the array entries does the script retrieve corresponding entries from the other two arrays.

LISTING 18-3

A Looping Array Lookup

HTML: jsb-18-03.html

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Parallel Array Lookup II</title>
      <script type="text/javascript" src="../jsb-global.js"></script>
      <script type="text/javascript" src="jsb-18-03.js"></script>
   </head>
   <body>
      <h1>Parallel Array Lookup II</h1>
      <form id="officeData" action="" method="post">
         <p>
            <label for="officeInput">Enter a regional office:</label>
            <input id="officeInput" name="officeInput" size="35">
            <input id="officeSearch" type="button" value="Search">
         </p>
         <p>
            <label for="manager">The manager is:</label>
            <input type="text" id="manager" name="manager" size="35" />
         </p>
         <p>
            <label for="quota">The office quota is:</label>
            <input type="text" id="quota" name="quota" size="8" />
         </p>
      </form>
   </body>
</html>
```

JavaScript: jsb-18-03.js

```
// initialize when the page has loaded
addEvent(window, 'load', initialize);

function initialize()
{
   // do this only if the browser can handle DOM methods
   if (document.getElementById)
   {
      // set up the data in global variables (by omitting 'var')
```

```
      aRegionalOffices = ["New York", "Chicago", "Houston", "Portland"];
      aRegionalManagers = ["Shirley Smith", "Todd Gaston", "Leslie Jones",
                           "Harold Zoot"];
      aRegOfficeQuotas = [300000, 250000, 350000, 225000];

      // point to the critical input fields & save in global variables
      oOffice = document.getElementById('officeInput');
      oSearch = document.getElementById('officeSearch');
      oManager = document.getElementById('manager');
      oQuota = document.getElementById('quota');

         // if they all exist...
         if (oOffice && oSearch && oManager && oQuota)
         {
            // apply behavior to the search button
            addEvent(oSearch, 'click', getData);
         }
   }
}

// when the Search button is clicked, do the lookup into parallel arrays
function getData(evt)
{
   // make a copy of the text box contents
   var sInputText = oOffice.value;

      // nothing entered?
      if (!sInputText)
      {
         alert('Please enter an office location.');
      }
      else
      {
         // loop through all entries of regionalOffices array
         for (var i = 0; i < aRegionalOffices.length; i++)
         {
            // compare entered text against each item in regionalOffices
            // (make both uppercase for easy comparison)
            if (sInputText.toUpperCase() == aRegionalOffices[i].toUpperCase())
            {
               // if they're the same, then break out of the for loop
               break;
            }
         }

         // make sure the i counter hasn't exceeded the max index value
         if (i < aRegionalOffices.length)
         {
            // display corresponding entries from parallel arrays
            oManager.value = aRegionalManagers[i];
            oQuota.value = aRegOfficeQuotas[i];
         }
         // otherwise the loop went all the way with no matches
```

continued

LISTING 18-3 *(continued)*

```
        else
        {
           // empty any previous values
           oManager.value = "";
           oQuota.value = "";

           // advise user
           alert("No match found for '" + sInputText + "'.");
        }
     }
   // return the focus to the office input field
   oOffice.focus();
}
```

Multidimensional Arrays

An alternate to parallel arrays is the simulation of a multidimensional array. While it's true that JavaScript arrays are one-dimensional, you can create a one-dimensional array of arrays or other objects. A logical approach is to make an array of custom objects, because the objects easily allow for naming of object properties, making references to multidimensional array data more readable. (Custom objects are discussed at length in Chapter 23, "Function Objects and Custom Objects." See also Chapter 19, "JSON–Native JavaScript Object Notation.")

Using the same data from the examples of parallel arrays, the following statements define an object constructor for each "data record." A new object is then assigned to each of four entries in the main array.

```
// custom object constructor
function officeRecord(city, manager, quota)
{
   this.city = city;
   this.manager = manager;
   this.quota = quota;
}

// create new main array
var regionalOffices = new Array();

// stuff main array entries with objects
regionalOffices[0] = new officeRecord("New York", "Shirley Smith", 300000);
regionalOffices[1] = new officeRecord("Chicago", "Todd Gaston", 250000);
regionalOffices[2] = new officeRecord("Houston", "Leslie Jones", 350000);
regionalOffices[3] = new officeRecord("Portland", "Harold Zoot", 225000);
```

The object constructor function (`officeRecord()`) assigns incoming parameter values to properties of the object. Therefore, to access one of the data points in the array, you use both array notations to get to the desired entry in the array and the name of the property for that entry's object:

```
var eastOfficeManager = regionalOffices[0].manager;
```

You can also assign string index values for this kind of array, as in:

```
regionalOffices["east"] = new officeRecord("New York", "Shirley Smith",
   300000);
```

and access the data via the same index:

```
var eastOfficeManager = regionalOffices["east"].manager;
```

But if you're more comfortable with the traditional multidimensional array (from your experience in other programming languages), you can also implement the above as an array of arrays with less code:

```
// create new main array
var regionalOffices = new Array();
// stuff main array entries with arrays
regionalOffices[0] = new Array("New York", "Shirley Smith", 300000);
regionalOffices[1] = new Array("Chicago", "Todd Gaston", 250000);
regionalOffices[2] = new Array("Houston", "Leslie Jones", 350000);
regionalOffices[3] = new Array("Portland", "Harold Zoot", 225000);
```

or with the extreme brevity of literal notation:

```
// create new main array
var regionalOffices = [
                         ["New York", "Shirley Smith", 300000],
                         ["Chicago", "Todd Gaston", 250000],
                         ["Houston", "Leslie Jones", 350000],
                         ["Portland", "Harold Zoot", 225000]
                      ];
```

Accessing a single data point of an array of arrays requires a double array reference. For example, retrieving the manager's name for the Houston office requires the following syntax:

```
var HoustonMgr = regionalOffices[2][1];
```

The first index in brackets is for the outermost array (`regionalOffices`); the second index in brackets points to the item of the array returned by `regionalOffices[2]`.

Simulating a Hash Table

Nearly all arrays shown so far in this chapter have used integers as their index values. A JavaScript array is a special type of object (the `object` type is covered in Chapter 23, "Function Objects and Custom Objects"). As a result, you can also assign values to customized properties of an array without interfering with the data stored in the array or the length of the array. In other words, you can "piggy-back" data in the array object. You may reference the values of these properties either using "dot" syntax (`array.propertyName`) or through array-looking syntax consisting of square brackets and the property name as a string inside the brackets (`array["propertyName"]`). An array used in this fashion is also known as an *associative array*. If you use the dot syntax, then you can't use characters such as hyphens, spaces, periods, and quotation marks in the `propertyName` values,

whereas the only limitation on the quoted bracket syntax is that the values not contain the same type of quotation marks as you're using to quote the expressions.

These do *not* work:

```
// Dot notation:
oArray.Chris Smith              // space
oArray.Chris-Smith              // hyphen
oArray.Chris.Smith              // period
oArray.Smith,Chris              // comma
oArray.Chris"Punky"Smith        // quotes

// Quoted bracket notation:
oArray["Chris "Punky" Smith"]   // matching quotes
oArray['Chris 'Punky' Smith']   // matching quotes
```

These *do* work:

```
// Dot notation:
oArray.ChrisSmith               // one word
oArray.Chris_Smith              // underscore

// Quoted bracket notation (nearly everything works):
oArray["Chris_Smith"]
oArray["Chris.Smith"]
oArray["Chris Smith"]
oArray["Smith, Chris"]
oArray["Chris 'Punky' Smith"]   // non-matching quotes
oArray['Chris "Punky" Smith']   // non-matching quotes
```

Addressing object properties by way of string indexes is sometimes very useful. For example, the multidimensional array described in the previous section consists of four objects. If your page contains a form whose job is to look through the array to find a match for a city chosen from a `select` list, the typical array lookup would loop through the length of the array, compare the chosen value against the `city` property of each object, and then retrieve the other properties when there was a match. For a four-item list, this isn't a big deal. But for a 100-item list, the process could get time consuming. A faster approach would be to jump directly to the array entry whose `city` property is the chosen value. That's what a simulated hash table can do for you (some programming languages have formal hash table constructions especially designed to act like a lookup table).

Create a simulated hash table after the array is populated by looping through the array and assigning properties to the array object as string values. Use string values that you expect to use for lookup purposes. For example, after the `regionalOffices` array has its component objects assigned, run through the following routine to make the hash table:

```
for (var i = 0; i < regionalOffices.length; i++)
{
   regionalOffices[regionalOffices[i].city] = regionalOffices[i];
}
```

You can retrieve the manager property of the Houston office object as follows:

```
var HoustonMgr = regionalOffices["Houston"].manager;
```

With the aid of the hash table component of the array, your scripts have the convenience of both numeric lookup (if the script needs to cycle through all items) and an immediate jump to an item.

Cross-Reference

For more ways to represent data structures in JavaScript, see Chapters 19 and 20. ■

Array Object

Properties	Methods
`constructor`	`concat()`
`length`	`every()*`
`prototype`	`filter()*`
	`forEach()*`
	`indexOf()*`
	`join()`
	`lastIndexOf()*`
	`map()*`
	`pop()`
	`push()`
	`reduce()`
	`reduceRight()`
	`reverse()`
	`shift()`
	`slice()`
	`some()*`
	`sort()`
	`splice()`
	`toLocaleString()`
	`toString()`
	`unshift()`

*Items marked with an asterisk in the list of `Array` object properties and methods are relatively recent additions with JavaScript 1.6 and later. They are supported by Mozilla browsers such as Firefox 2.0+ but not by Internet Explorer as of version 8, which has a browser market share too large to ignore. We have provided code in this chapter that adds these new methods to the `Array` object if they don't already exist.

Array object properties

constructor

(See *string*.constructor in Chapter 15, "The String Object")

length

Value: Integer Read/Write

Compatibility: WinIE4+, MacIE4+, NN3+, Moz+, Safari+, Opera+, Chrome+

A true array object's length property reflects the number of entries in the array. An entry can be any kind of JavaScript value, including null. If an entry is in the 10th cell and the rest are null, the length of that array is 10. Note that because array index values are zero-based, the index of the last cell of an array is one less than the length (9 in this case). This characteristic makes it convenient to use the property as an automatic counter to append a new item to an array:

```
myArray[myArray.length] = valueOfAppendedItem;
```

Thus, a generic function does not have to know which specific index value to apply to an additional item in the array.

prototype

Value: Variable or function Read/Write

Compatibility: WinIE4+, MacIE4+, NN3+, Moz+, Safari+, Opera+, Chrome+

Inside JavaScript, an array object has its dictionary definition of methods and length property — items that all array objects have in common. The prototype property enables your scripts to ascribe additional properties or methods that apply to all the arrays you create in the currently loaded documents. You can override this prototype, however, for any individual object.

Example

To demonstrate how the prototype property works, Listing 18-4 creates a prototype property for all array objects generated from the static Array object. As the script generates new arrays (instances of the Array object, just as a date object is an instance of the Date object), the property automatically becomes a part of those arrays. In one array, c, you override the value of the prototype sponsor property. By changing the value for that one object, you don't alter the value of the prototype for the Array object. Therefore, another array created afterward, d, still gets the original sponsor property value.

LISTING 18-4

Adding a prototype Property

HTML: jsb-18-04.html

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
```

```
      <title>Array prototypes</title>
      <script type="text/javascript" src="jsb-18-04.js"></script>
   </head>
   <body>
      <h1>Array prototypes</h1>
   </body>
</html>
```

JavaScript: jsb-18-04.js

```
// add prototype to all Array objects
Array.prototype.sponsor = "DG";

// create new arrays
var a = new Array();
var b = new Array();
var c = new Array();

// override prototype property for one 'instance'
c.sponsor = "JS";

// this one picks up the original prototype
var d = new Array();

// display results
var sMsg = "Array a is brought to you by: " + a.sponsor + "\n";
sMsg += "Array b is brought to you by: " + b.sponsor + "\n";
sMsg += "Array c is brought to you by: " + c.sponsor + "\n";
sMsg += "Array d is brought to you by: " + d.sponsor;
alert(sMsg);
```

You can assign properties and functions to a prototype. To assign a function, define the function as you normally would in JavaScript. Then assign the function to the `prototype` by name:

```
function newFunc(param1)
{
   // statements
}
Array.prototype.newMethod = newFunc;  // omit parentheses in this reference
```

where `newMethod` is whatever you want to name the method (the function name will not be retained).

When you need to call upon that function (which has essentially become a new temporary method for the `Array` object), invoke it as you would any object method. Therefore, if an array named `CDCollection` has been created and a prototype method `showCoverImage()` has been attached to the array, the call to invoke the method for a tenth listing in the array is

```
CDCollection.showCoverImage(9);
```

where the parameter of the function uses the index value to perhaps retrieve an image whose URL is a property of an object assigned to the 10th item of the array.

Array object methods

After you have information stored in an array, JavaScript provides several methods to help you manage that data. These methods, all of which belong to array objects you create, have evolved over time, so pay close attention to browser compatibility if you're in need of supporting legacy (pre-version 4) browsers.

array.concat(*array2*)

Returns: Array object

Compatibility: WinIE4+, MacIE4+, NN4+, Moz+, Safari+, Opera+, Chrome+

The `array.concat()` method allows you to join two array objects into a new, third array object. The action of concatenating the arrays does not alter the contents or behavior of the two original arrays. To join the arrays, you refer to the first array object to the left of the period before the method; a reference to the second array is the parameter to the method. For example:

```
var array1 = new Array(1,2,3);
var array2 = new Array("a","b","c");
var array3 = array1.concat(array2);
    // result: array with values 1,2,3,"a","b","c"
```

If an array element is a string or number value (not a string or number object), the values are copied from the original arrays into the new one. All connection with the original arrays ceases for those items. But if an original array element is a reference to an object of any kind, JavaScript copies a reference from the original array's entry into the new array; so, if you make a change to either array's entry, the change occurs to the object, and both array entries reflect the change to the object.

Example

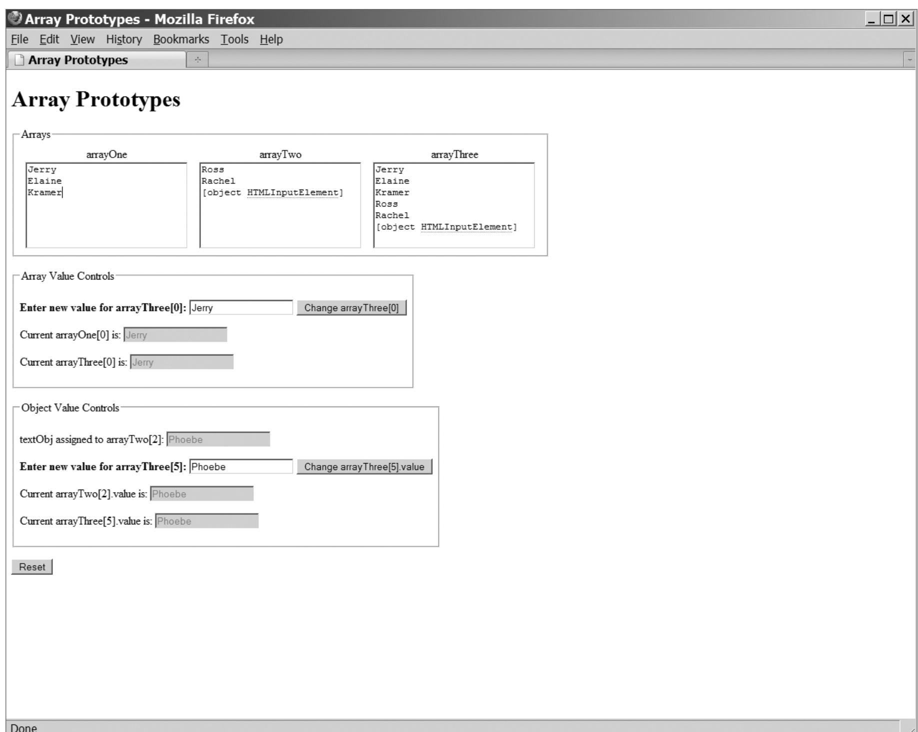
Listing 18-5 is a bit complex, but it demonstrates both how arrays can be joined with the `array.concat()` method and how values and objects in the source arrays do or do not propagate based on their data type. The page is shown in Figure 18-1.

After you load the page, you see readouts of three arrays. The first array consists of all string values; the second array has two string values and a reference to a form object on the page (a text box named "original" in the HTML). In the initialization routine of this page, not only are the two source arrays created, but they are joined with the `array.concat()` method, and the result is shown in the third box. To show the contents of these arrays in columns, we use the `array.join()` method, which brings the elements of an array together as a string delimited in this case by a return character — giving us an instant column of data.

Two series of fields and buttons let you experiment with the way values and object references are linked across concatenated arrays. In the first group, if you enter a new value to be assigned to `arrayThree[0]`, the new value replaces the string value in the combined array. Because regular values do not maintain a link back to the original array, only the entry in the combined array is changed. A call to `showArrays()` proves that only the third array is affected by the change.

FIGURE 18-1

Object references remain ''alive'' in a concatenated array.



More complex is the object relationship for this demonstration. A reference to the first text box of the second grouping has been assigned to the third entry of `arrayTwo`. After concatenation, the same reference is now in the last entry of the combined array. If you enter a new value for a property of the object in the last slot of `arrayThree`, the change goes all the way back to the original object — the first text box in the lower grouping. Thus, the text of the original field changes in response to the change of `arrayThree[5]`. And because all references to that object yield the same result, the reference in `arrayTwo[2]` points to the same text object, yielding the same new answer. The display of the array contents doesn't change, because both arrays still contain a reference to the same object (and the `value` attribute showing in the `<input>` tag of the column listings refers to the default value of the tag, not to its current algorithmically retrievable value shown in the last two fields of the page).

LISTING 18-5

Array Concatenation

HTML: jsb-18-05.html

```
<!DOCTYPE html>
<html>
   <head>
```

continued

LISTING 18-5 *(continued)*

```
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Array Prototypes</title>
      <link type="text/css" rel="stylesheet" href="jsb-18-05.css">
      <script type="text/javascript" src="../jsb-global.js"></script>
      <script type="text/javascript" src="jsb-18-05.js"></script>
   </head>
   <body>
      <h1>Array Prototypes</h1>
      <form>
         <fieldset id="arrays">
         <legend>Arrays</legend>
         <p>
            <label>arrayOne</label>
            <textarea name="array1" cols="25" rows="6"></textarea>
         </p>
         <p>
            <label>arrayTwo</label>
            <textarea name="array2" cols="25" rows="6"></textarea>
         </p>
         <p>
            <label>arrayThree</label>
            <textarea name="array3" cols="25" rows="6"></textarea>
         </p>
      </fieldset>
         <fieldset>
         <legend>Array Value Controls</legend>
         <p>
            <label class="input" for="source1">Enter new value for
                                           arrayThree[0]:</label>
            <input type="text" name="source1" value="Jerry">
            <input id="button1" type="button" value="Change arrayThree[0]">
         </p>
         <p>
            <label>Current arrayOne[0] is:</label>
            <input id="result1" type="text" disabled="disabled">
         </p>
         <p>
            <label>Current arrayThree[0] is:</label>
            <input id="result2" type="text" disabled="disabled">
         </p>
         </fieldset>

         <fieldset>
         <legend>Object Value Controls</legend>
         <p>
            <label>textObj assigned to arrayTwo[2]:</label>
            <input type="text" name="original" disabled="disabled">
         </p>
         <p>
            <label class="input">Enter new value for arrayThree[5]:</label>
```

```
            <input type="text" name="source2" value="Phoebe">
            <input id="button2" type="button" value="Change arrayThree[5].value">
         </p>
         <p>
            <label>Current arrayTwo[2].value is:</label>
            <input type="text" name="result3" disabled="disabled">
         </p>
         <p>
            <label>Current arrayThree[5].value is:</label>
            <input type="text" name="result4" disabled="disabled">
         </p>
         </fieldset>
      <p class="reset">
         <input id="buttonReset" type="button" value="Reset">
      </p>
      </form>
   </body>
</html>
```

JavaScript: jsb-18-05.js

```
// initialize when the page has loaded
addEvent(window, 'load', initialize);

// global variables
var arrayOne, arrayTwo, arrayThree, textObj;

function initialize()
{
   // do this only if the browser can handle DOM methods
   if (document.getElementById)
   {
      form = document.forms[0];

      var oButton = document.getElementById('button1');
         if (oButton) oButton.onclick = update1;

      oButton = document.getElementById('button2');
         if (oButton) oButton.onclick = update2;

      oButton = document.getElementById('buttonReset');
         if (oButton) oButton.onclick = function() { location.reload(); };

      // populate arrays
         textObj = form.original;
         arrayOne = new Array("Jerry", "Elaine", "Kramer");
         arrayTwo = new Array("Ross", "Rachel", textObj);
         arrayThree = arrayOne.concat(arrayTwo);

         // perform initial update
         update1();
         update2();
```

continued

arrayObject.concat()

LISTING 18-5 *(continued)*

```
         showArrays();
      }
}

// display current values of all three arrays
function showArrays()
{
   form.array1.value = arrayOne.join("\n");
   form.array2.value = arrayTwo.join("\n");
   form.array3.value = arrayThree.join("\n");
}

// change the value of first item in Array Three
function update1(evt)
{
   arrayThree[0] = form.source1.value;
   form.result1.value = arrayOne[0];
   form.result2.value = arrayThree[0];
   showArrays();
}

// change value of object property pointed to in Array Three
function update2(evt)
{
   arrayThree[5].value = form.source2.value;
   form.result3.value = arrayTwo[2].value;
   form.result4.value = arrayThree[5].value;
   showArrays();
}
```

Stylesheet: jsb-18-05.css

```
fieldset
{
   clear: left;
   float: left;
   margin-bottom: 1em;
   padding: .5em;
}
fieldset#arrays p
{
   float: left;
   margin: 0 .5em;
}
fieldset#arrays label
{
   display: block;
   text-align: center;
}
```

```
label.input
{
   font-weight: bold;
}
p.reset
{
   clear: left;
}
```

Related Item: `array.join()` method

array.every(*callback[, thisObject]*)
array.some(*callback[, thisObject]*)

Returns: Boolean

Compatibility: FF2+, Safari3+, Chrome1+, Opera 9.5+, WinIE−, MacIE−

`every()` and `some()` let you test the elements of an array with a callback function of your choice. (A *callback* is a function that is passed as an argument to another function.) The difference between the two methods is that `every()` returns `true` only if every array element tests true, while `some()` returns `true` if even one element tests true. The original array is not modified unless your callback function explicitly modifies it.

The callback function can contain any logic you want but in this case is required to return a Boolean value (either `true` or `false`). It takes three arguments: the value of the array element being examined, its index in the array, and a reference to the array object being traversed. `every()` uses the callback to test each member of the array in turn until a false value is returned or the end of the array is reached. `some()` uses the callback to test each member of the array until a true value is returned or the end of the array is reached. Note that if the array is empty, `every()` will return true because it hasn't encountered a value that would test false!

Here's a simple example that checks whether an array contains all positive numbers:

```
function isPositive(iValue, iIndex, aArray)
{
   return (iValue > 0);
}

var a = new Array(5, 4, 3, 2, 1);

var bResult = a.every(isPositive);
// returns true because there are no non-positive elements

bResult = a.some(isPositive);
// returns true because there is at least one positive element


a = [5, 4, 3, 2, 1, 0];

bResult = a.every(isPositive);
// returns false because zero is non-positive
```

```
bResult = a.some(isPositive);
// returns true because there is at least one positive element


a = new Array();

bResult = a.every(isPositive);
// returns true since there are no non-positive elements in an empty array!

bResult = a.some(isPositive);
// returns false because there are no positive elements in an empty array
```

The following example takes advantage of the callback function's other arguments to use other array elements to evaluate the current one. A *Fibonacci sequence* is one in which each number is the sum of the previous two. Of course, the first two numbers in the sequence are exceptions to this algorithm and are "hard-coded" as zero and one, but the rest are calculable:

```
function isFibonacci(iValue, iIndex, aArray)
{
   switch (iIndex)
   {
      case 0:
         return (iValue == 0);
      break;

      case 1:
         return (iValue == 1);
      break;

      default:
         return (iValue == aArray[iIndex-2] + aArray[iIndex-1]);
   }
}

var a = new Array(0, 1, 1, 2, 3, 5, 8, 13, 21);
var bResult = a.every(isFibonacci);
// returns true

a = [0, 1, 1, 2, 3, 5, 8, 13, 22];
var bResult = a.every(isFibonacci);
// returns false because 22 isn't 8 + 13
```

This same functionality could be gotten from a `for()` loop. The advantage of using `every()` is the encapsulation of program logic in discrete functions that can be re-used in different places in your script.

To equip browsers that don't yet support the `every()` method of JavaScript 1.6, you can supply that functionality to the array object using code supplied by Mozilla.org:

```
if (!Array.prototype.every)
{
   Array.prototype.every = function(fun /*, thisp*/)
```

```
    {
       var len = this.length >>> 0;
       if (typeof fun != "function")
       {
          throw new TypeError();
       }

       var thisp = arguments[1];
       for (var i = 0; i < len; i++)
       {
          if (i in this &&
                !fun.call(thisp, this[i], i, this))
             return false;
       }

       return true;
    };
 }
```

Related Items: `array.filter()`,`array.forEach()`, `array.map()` methods

`array``.filter(`*`callback[, thisObject]`*`)`

Returns: Array

Compatibility: FF2+, Safari3+, Chrome1+, Opera 9.5+, WinIE−, MacIE−

Similar to `every()`, the `filter()` method lets you test every element of an array with a callback function of your choice, and then returns a new array consisting of every element that tested `true`. The original array is not modified unless your callback function explicitly modifies it.

The callback function examines a single array element and returns Boolean `true` or `false`. It takes three arguments: the value of the array element being examined, its index in the array, and a reference to the array object being traversed. Every element that tests true is added to the result array.

This example uses our previous callback function to check for positive numbers:

```
function isPositive(iValue, iIndex, aArray)
{
   return (iValue > 0);
}

var a = new Array(1, -2, -3, 4, 5, -6);
var aResult = a.filter(isPositive);
// result: [1, 4, 5]
```

Mozilla suggests the following code to compensate for browsers that don't yet support the `filter()` method of JavaScript 1.6:

```
if (!Array.prototype.filter)
{
   Array.prototype.filter = function(fun /*, thisp*/)
```

```
      {
         var len = this.length >>> 0;
         if (typeof fun != "function")
            throw new TypeError();

         var res = new Array();
         var thisp = arguments[1];
         for (var i = 0; i < len; i++)
         {
            if (i in this)
            {
               var val = this[i]; // in case fun mutates this
               if (fun.call(thisp, val, i, this))
                  res.push(val);
            }
         }

         return res;
      };
   }
```

Related Items: `array.every()`, `array.forEach()`, `array.map()`, `array.some()` methods

array.forEach(*callback[, thisObject]*)

Returns: Array

Compatibility: FF2+, Safari3+, Chrome1+, Opera 9.5+, WinIE−, MacIE−

`forEach()` executes a function on every element of an array. It does not return a value and the original array is not modified unless your callback function explicitly modifies it.

The callback function takes three arguments: the value of the array element being examined, its index in the array, and a reference to the array object being traversed.

The following example builds a message that displays the contents of an array. Note that in the callback function `buildMsg()`, the keyword `this` refers to the custom object `oMsg` that we're passing to `forEach()`:

```
function sendMessages(sValue, iIndex, aArray)
{
   sendMsg(sValue, this.text);  // where sendMsg() is defined elsewhere ...
}

var a = new Array('chris@example.com', 'pat@example.net', 'jessie@example.org');

var oMessage = {text: "Dear friend,\nYou are cordially invited..."};
a.forEach(sendMessages, oMessage);
```

Alternatively, the preceding example could be written using the more universally supported `for()` method:

```
var a = new Array('apple', 'banana', 'cardamom', 'dandelion');
var oMsg = {buffer: 'This array contains:'};
```

```
for (var i = 0; i < a.length; i++)
{
   oMsg.buffer += '\n' + i + ':' + a[i];
}
alert(oMsg.buffer);
```

Mozilla suggests the following code to equip browsers that don't yet support the `forEach()` method of JavaScript 1.6:

```
if (!Array.prototype.forEach)
{
   Array.prototype.forEach = function(fun /*, thisp*/)
   {
      var len = this.length >>> 0;
      if (typeof fun != "function")
         throw new TypeError();

      var thisp = arguments[1];
      for (var i = 0; i < len; i++)
      {
         if (i in this)
            fun.call(thisp, this[i], i, this);
      }
   };
}
```

Related Items: `array.every()`,`array.filter()`, `array.map()`, `array.some()` methods

array.indexOf(*searchString[, startFrom]*)
array.lastIndexOf(*searchString[, startFrom]*)

Returns: Index value of the array element whose value matches searchString, or −1 if not found

Compatibility: FF2+, Safari3+, Chrome1+, Opera 9.5+, WinIE−, MacIE−

These two methods search for an array element that matches `searchString` exactly and return the index of the first (or last) matching element, or return −1 if no match is found.

The principle difference between these two methods is that `indexOf()` searches from left to right, starting at the beginning of the array ascending and `lastIndexOf()` searches from right to left, from the end of the array descending. In either case, the value that's returned is the index of the found item — its offset from the beginning of the array. Therefore, if there's only one matching element in the array and `startFrom` is not specified, both `indexOf()` and `lastIndexOf()` will return the same value because they'll find the same matching element.

The optional `startFrom` argument tells the method how far from the beginning or end of the array to begin the search. If it's missing, it's assumed to be zero — that is, begin the search at the first (with `indexOf()`) or last (with `lastIndexOf()`) element. If `startFrom` is a positive number, it's used as a starting offset from the beginning of the array; if it's negative, it's used as an offset from the end of the array.

Examples:

```
var a = new Array('cat', 'dog', 'cat');

// indexOf searches left to right:
a.indexOf('cat');         // result = 0 (search elements 0, 1, and 2)
a.indexOf('cat', 1);      // result = 2 (search elements 1 and 2)
a.indexOf('cat', 2);      // result = 2 (search element 2)
a.indexOf('cat', 3);      // result = -1 (no elements to search)
a.indexOf('cat', -1);     // result = 2 (search element 2)
a.indexOf('cat', -2);     // result = 2 (search elements 1 and 2)
a.indexOf('cat', -3);     // result = 0 (search elements 0, 1, and 2)

// lastIndexOf searches right to left:
a.lastIndexOf('cat');     // result = 2 (search elements 2, 1, and 0)
a.lastIndexOf('cat', 1);  // result = 0 (search elements 1 and 0)
a.lastIndexOf('cat', 2);  // result = 2 (search elements 2, 1, and 0)
a.lastIndexOf('cat', 3);  // result = 2 (search whole array)
a.lastIndexOf('cat', -1); // result = 2 (search elements 2, 1, and 0)
a.lastIndexOf('cat', -2); // result = 0 (search elements 1 and 0)
a.lastIndexOf('cat', -3); // result = 0 (search element 0)
```

Mozilla suggests the following code to equip browsers that don't yet support the `indexOf()` method of JavaScript 1.6:

```
if (!Array.prototype.indexOf)
{
   Array.prototype.indexOf = function(elt /*, from*/)
   {
      var len = this.length >>> 0;

      var from = Number(arguments[1]) || 0;
      from = (from < 0)
             ? Math.ceil(from)
             : Math.floor(from);
      if (from < 0)
         from += len;

      for (; from < len; from++)
      {
         if (from in this &&
               this[from] === elt)
            return from;
      }
      return -1;
   };
}
```

And the following to equip browsers that don't yet support `lastIndexOf()`:

```
if (!Array.prototype.lastIndexOf)
{
   Array.prototype.lastIndexOf = function(elt /*, from*/)
   {
      var len = this.length;

      var from = Number(arguments[1]);
      if (isNaN(from))
      {
         from = len - 1;
      }
      else
      {
         from = (from < 0)
              ? Math.ceil(from)
              : Math.floor(from);
         if (from < 0)
            from += len;
         else if (from >= len)
            from = len - 1;
      }

      for (; from > -1; from--)
      {
         if (from in this &&
             this[from] === elt)
            return from;
      }
      return -1;
   };
}
```

array.join(*separatorString*)

Returns: String of entries from the array delimited by the *separatorString* value

Compatibility: WinIE4+, MacIE4+, NN3+, Moz+, Safari+, Opera+, Chrome+

You cannot directly view data that is stored in an array. Nor can you put an array into a form element for transmittal to a server-side program that expects a string of text. To make the transition from discrete array elements to string, the `array.join()` method handles what would otherwise be a nasty string manipulation exercise.

The sole parameter for this method is a string of one or more characters that you want to act as a delimiter between entries. For example, if you want commas between array items in their text version, the statement is

```
var arrayText = myArray.join(",");
```

***arrayObject*.join**

Invoking this method does not change the original array in any way. Therefore, you need to assign the results of this method to another variable or a value property of a form element.

Example

The script in Listing 18-6 converts an array of planet names into a text string. The page provides you with a field to enter the delimiter string of your choice and shows the results in a text area.

LISTING 18-6

Using the Array.join() Method

HTML: jsb-18-06.html

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Array.join()</title>
      <script type="text/javascript" src="../jsb-global.js"></script>
      <script type="text/javascript" src="jsb-18-06.js"></script>
   </head>
   <body>
      <h1>Array.join(): Converting arrays to strings</h1>
      <form>
         <p>This document contains an array of planets in our solar system.</p>
         <p>
            <label for="delimiter">Enter a string to act
                                   as a delimiter between entries:</label>
            <input type="text" id="delimiter" name="delim" value="," size="5">
         </p>
         <p>
            <input id="displayButton" type="button" value="Display as String">
            <input type="reset">
         </p>
         <p>
            <textarea id="output" name="output" rows="4" cols="40"
                         wrap="virtual"></textarea>
         </p>
      </form>
   </body>
</html>
```

JavaScript: jsb-18-06.js

```
// initialize when the page has loaded
addEvent(window, 'load', initialize);

function initialize()
{
   // do this only if the browser can handle DOM methods
```

```
   if (document.getElementById)
   {
      // point to critical elements
      oDelimiter = document.getElementById('delimiter');
      oOutput = document.getElementById('output');
      var oButton = document.getElementById('displayButton');

         // if they all exist...
         if (oDelimiter && oOutput && oButton)
         {
            // apply behavior to button
            oButton.onclick = convert;

            // set global array
            solarSys = new Array("Mercury", "Venus", "Earth", "Mars", "Jupiter",
                                 "Saturn", "Uranus", "Neptune");
         }
   }
}

// join array elements into a string
function convert(evt)
{
   var delimiter = oDelimiter.value;
   oOutput.value = decodeURIComponent(solarSys.join(delimiter));
}
```

Notice that this method takes the parameter very literally. If you want to include non-alphanumeric characters, such as a newline or tab, do so with URL-encoded characters (`%0D` for a carriage return; `%09` for a tab) instead of inline string literals. Coming up in Listing 18-7, the results of the `array.join()` method are subjected to the `decodeURIComponent()` function in order to display them in the `textarea`.

Related Items: `string.split()` method

array.map(*callback[, thisObject]*)

Returns: Array

Compatibility: FF2+, Safari3+, Chrome1+, Opera 9.5+, WinIE−, MacIE−

`map()` creates a new array by running a user-provided function on each element of the original array.

The callback function returns a value that is used to build the result array. It takes three arguments: the value of the array element being examined, its index in the array, and a reference to the array object being traversed.

If provided, `thisObject` will be used as the `this` value within the callback function.

Example:

```
function square(iValue, iIndex, aArray)
{
```

```
    return iValue * iValue;
}

var a = new Array(0, 1, 2, 3, 4, 5, 6, 7);
var b = a.map(square);
// result: b == [0, 1, 4, 9, 16, 25, 36, 49]
```

You could also write this by creating the callback function on-the-fly:

```
var a = new Array(0, 1, 2, 3, 4, 5, 6, 7);
var b = a.map(function(i) { return i * i; });
```

Mozilla suggests the following code to equip browsers that don't yet support the `map()` method of JavaScript 1.6:

```
if (!Array.prototype.map)
{
   Array.prototype.map = function(fun /*, thisp*/)
   {
      var len = this.length >>> 0;
      if (typeof fun != "function")
         throw new TypeError();

      var res = new Array(len);
      var thisp = arguments[1];
      for (var i = 0; i < len; i++)
      {
         if (i in this)
            res[i] = fun.call(thisp, this[i], i, this);
      }

      return res;
   };
}
```

Related Items: `array.every()`, `array.filter()`,`array.forEach()`, `array.some()` methods

array.pop()
array.push(*valueOrObject*)
array.shift()
array.unshift(*valueOrObject*)

Returns: One array entry value

Compatibility: WinIE5.5+, MacIE−, NN4+, Moz+, Safari+, Opera+, Chrome+

The notion of a *stack* is well known to experienced programmers, especially those who know about the inner workings of assembly language at the CPU level. Even if you've never programmed a stack before, you have encountered the concept in real life many times. The classic analogy is the spring-loaded pile of cafeteria trays. If the pile were created one tray at a time, each tray would be pushed onto the top of the stack of trays. When a customer comes along, the topmost tray (the last

one to be pushed onto the stack) gets popped off. The last one to be put on the stack is the first one to be taken off.

JavaScript in modern browsers lets you turn an array into one of these spring-loaded stacks. But instead of placing trays on the pile, you can place any kind of data at either end of the stack, depending on which method you use to do the stacking. Similarly, you can extract an item from either end.

Perhaps the most familiar terminology for this is *push* and *pop*. When you `push()` a value onto an array, the value is appended as the last entry in the array. When you issue the `array.pop()` method, the last item in the array is removed from the stack and is returned, and the array shrinks in length by one. In the following sequence of statements, watch what happens to the value of the array used as a stack:

```
var source = new Array("Homer","Marge","Bart","Lisa","Maggie");
var stack = new Array();
    // stack = <empty>
stack.push(source[0]);
    // stack = "Homer"
stack.push(source[2]);
    // stack = "Homer","Bart"
var Simpson1 = stack.pop();
    // stack = "Homer" ; Simpson1 = "Bart"
var Simpson2 = stack.pop();
    // stack = <empty> ; Simpson2 = "Homer"
```

While `push()` and `pop()` work at the end of an array, another pair of methods works at the front. Their names are not as picturesque as `push()` and `pop()`. To insert a value at the front of an array, use the `array.unshift()` method; to grab the first element and remove it from the array, use `array.shift()`. Of course, you are not required to use these methods in matching pairs. If you `push()` a series of values onto the back end of an array, you can `shift()` them off from the front end without complaint. It all depends on how you need to process the data.

Related Items: `array.concat()`, `array.slice()` method

array.reduce(*callback[, initialValue]*)
array.reduceRight(*callback[, initialValue]*)

Returns: A single value

Compatibility: FF3+, Safari4+

These methods reduce an array down to a single value. `reduce()` operates on the array elements starting with the first element, whereas `reduceRight()` begins with the last element. Both methods accept as arguments a callback function and an optional initial value (since the very first iteration will have only one array element to work on). The callback function has two arguments, (`firstValue`, `secondValue`), which it processes in some way to derive a single return value.

For example, the following code reduces an array of numbers down to their sum:

```
function addEmUp(a, b)
{
   return a + b;
}
```

```
var a = new Array(1, 2, 3, 4, 5);
var b = a.reduce(addEmUp);
// result: b == 15
```

For simple addition, it doesn't matter whether the reduction occurred frontward or backward, but look at what happens when we apply that same callback to text. (Remember that in JavaScript the plus sign adds numbers but concatenates text.)

```
var a = new Array('d', 'e', 's', 's', 'e', 'r', 't', 's');

var b = a.reduce(addEmUp);
// result: b == 'desserts'

var b = a.reduceRight(addEmUp);
// result: b == 'stressed'
```

Mozilla suggests the following code to equip browsers that don't yet support the `reduce()` method of JavaScript 1.8:

```
if (!Array.prototype.reduce)
{
  Array.prototype.reduce = function(fun /*, initial*/)
  {
    var len = this.length >>> 0;
    if (typeof fun != "function")
      throw new TypeError();

    // no value to return if no initial value and an empty array
    if (len == 0 && arguments.length == 1)
      throw new TypeError();

    var i = 0;
    if (arguments.length >= 2)
    {
      var rv = arguments[1];
    }
    else
    {
      do
      {
        if (i in this)
        {
          rv = this[i++];
          break;
        }

        // if array contains no values, no initial value to return
        if (++i >= len)
          throw new TypeError();
      }
      while (true);
```

```
      }

      for (; i < len; i++)
      {
        if (i in this)
          rv = fun.call(null, rv, this[i], i, this);
      }

      return rv;
    };
  }
```

... and likewise for `reduceRight()`:

```
  if (!Array.prototype.reduceRight)
  {
    Array.prototype.reduceRight = function(fun /*, initial*/)
    {
      var len = this.length >>> 0;
      if (typeof fun != "function")
        throw new TypeError();

      // no value to return if no initial value, empty array
      if (len == 0 && arguments.length == 1)
        throw new TypeError();

      var i = len - 1;
      if (arguments.length >= 2)
      {
        var rv = arguments[1];
      }
      else
      {
        do
        {
          if (i in this)
          {
            rv = this[i--];
            break;
          }

          // if array contains no values, no initial value to return
          if (--i < 0)
            throw new TypeError();
        }
        while (true);
      }

      for (; i >= 0; i--)
      {
        if (i in this)
```

```
                rv = fun.call(null, rv, this[i], i, this);
            }

            return rv;
        };
    }
```

array.reverse()

Returns: Array of entries in the opposite order of the original

Compatibility: WinIE4+, MacIE4+, NN3+, Moz+, Safari+, Opera+, Chrome+

Occasionally, you may find it more convenient to work with an array of data in reverse order. Although you can concoct repeat loops to count backward through index values, a server-side program may prefer the data in a sequence opposite to the way it was most convenient for you to script it.

You can have JavaScript switch the contents of an array for you: Whatever element was last in the array becomes the 0 index item in the array. Bear in mind that if you do this, you're restructuring the original array, not copying it, even though the method also returns a copy of the reversed version. A reload of the document restores the order as written in the HTML document.

Example

Listing 18-7 is an enhanced version of Listing 18-6, which includes another button and function that reverse the array and display it as a string in a text area.

LISTING 18-7

Array.reverse() Method

HTML: jsb-18-07.html

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Array.reverse()</title>
      <script type="text/javascript" src="../jsb-global.js"></script>
      <script type="text/javascript" src="jsb-18-07.js"></script>
   </head>
   <body>
      <h1>Array.reverse(): Reversing array element order</h1>
      <form>
         <p>This document contains an array of planets in our solar system.</p>
         <p>
            <label for="delimiter">Enter a string to act as
                                 a delimiter between entries:</label>
            <input type="text" id="delimiter" name="delim" value="," size="5">
         </p>
         <p>
            <input type="button" id="showAsIs" value="Array as-is">
```

```
            <input type="button" id="reverseIt" value="Reverse the array">
            <input type="reset">
            <input type="button" id="reload" value="Reload">
         </p>
         <p>
            <textarea id="output" name="output" rows="4" cols="40"
                      wrap="virtual"></textarea>
         </p>
      </form>
   </body>
</html>
```

JavaScript: jsb-18-07.js

```
// initialize when the page has loaded
addEvent(window, 'load', initialize);

function initialize()
{
   // do this only if the browser can handle DOM methods
   if (document.getElementById)
   {
      // point to critical elements
      oDelimiter = document.getElementById('delimiter');
      oOutput = document.getElementById('output');
      var oButtonShowAsIs = document.getElementById('showAsIs');
      var oButtonReverseIt = document.getElementById('reverseIt');
      var oReload = document.getElementById('reload');

         // if they all exist...
         if (oDelimiter && oOutput && oButtonShowAsIs && oButtonReverseIt)
         {
            // apply behaviors to buttons
            oButtonShowAsIs.onclick = showAsIs;
            oButtonReverseIt.onclick = reverseIt;
            oReload.onclick = function() { location.reload(); };

            // set global array
            solarSys = new Array("Mercury", "Venus", "Earth", "Mars", "Jupiter",
                                 "Saturn", "Uranus", "Neptune");
         }
   }
}

// show array as currently in memory
function showAsIs(evt)
{
   var delimiter = oDelimiter.value;
   oOutput.value = decodeURIComponent(solarSys.join(delimiter));
}

// reverse array order, then display as string
```

continued

LISTING 18-7 *(continued)*

```
function reverseIt(evt)
{
   var delimiter = oDelimiter.value;
   solarSys.reverse();   // reverses original array
   oOutput.value = decodeURIComponent(solarSys.join(delimiter));
}
```

Notice that the `solarSys.reverse()` method stands by itself (meaning, nothing captures the returned value) because the method modifies the `solarSys` array. You then run the now-inverted `solarSys` array through the `array.join()` method for your text display.

Related Items: `array.sort()` method

array.slice(*startIndex* [, *endIndex*])

Returns: Array

Compatibility: WinIE4+, MacIE4+, NN4+, Moz+, Safari+, Opera+, Chrome+

Behaving as its like-named string method, `array.slice()` lets you extract a contiguous series of items from an array. The extracted segment becomes an entirely new array object. Values and objects from the original array have the same kind of behavior as arrays created with the `array.concat()` method.

One parameter is required — the starting index point for the extraction. If you don't specify a second parameter, the extraction goes all the way to the end of the array; otherwise the extraction goes to, *but does not include,* the index value supplied as the second parameter. For example, extracting Earth's neighbors from an array of planet names looks like the following:

```
var solarSys = new Array("Mercury","Venus","Earth","Mars",
   "Jupiter","Saturn","Uranus","Neptune","Pluto");
var nearby = solarSys.slice(1,4);
    // result: new array of "Venus", "Earth", "Mars"
```

Related Items: `array.splice()`, `string.slice()` methods

array.sort([*compareFunction*])

Returns: Array of entries in the order as determined by the *compareFunction* algorithm

Compatibility: WinIE4+, MacIE4+, NN3+, Moz+, Safari+, Opera+, Chrome+

JavaScript array sorting is both powerful and a bit complex to script if you haven't had experience with this kind of sorting methodology. The purpose, obviously, is to let your scripts sort entries of an array by almost any kind of criterion that you can associate with an entry. For entries consisting of strings, the criterion may be their alphabetical order or their length; for numeric entries, the criterion may be their numerical order.

Look first at the kind of sorting you can do with the `array.sort()` method by itself (for example, without calling a comparison function). When no parameter is specified, JavaScript takes a snapshot of

the contents of the array and converts items to strings. From there, it performs a string sort of the values. ASCII values of characters govern the sort, which means that numbers are sorted by their string values, not their numeric values. This fact has strong implications if your array consists of numeric data: The value 201 sorts before 88, because the sorting mechanism compares the first characters of the strings ("2" versus "8") to determine the sort order. For simple alphabetical sorting of string values in arrays, the plain `array.sort()` method does the trick.

Fortunately, additional intelligence is available that you can add to array sorting. The key tactic is to define a function that helps the `sort()` method compare items in the array. A comparison function is passed two values from the array (what you don't see is that the `array.sort()` method rapidly sends numerous pairs of values from the array to help it sort through all entries). The comparison function lets the `sort()` method know which of the two items comes before the other, based on the value the function returns. Assuming that the function compares two values, `a` and `b`, the returned value reveals information to the `sort()` method, as shown in Table 18-1.

TABLE 18-1

Comparison Function Return Values

Return Value Range	Meaning
< 0	Value `b` should sort later than `a`.
0	The order of `a` and `b` should not change.
> 0	Value `a` should sort later than `b`.

Consider the following example:

```
myArray = new Array(12, 5, 200, 80);
function compare(a,b)
{
   return a - b;
}
myArray.sort(compare);
```

The array has four numeric values in it. To sort the items in numerical order, you define a comparison function (arbitrarily named `compare()`), which is called from the `sort()` method. Note that unlike invoking other functions, the parameter of the `sort()` method uses a reference to the function, which lacks parentheses.

When the `compare()` function is called, JavaScript automatically sends two parameters to the function in rapid succession until each element has been compared with the others. Every time `compare()` is called, JavaScript assigns two of the array's values to the parameter variables (`a` and `b`). In the preceding example, the returned value is the difference between `a` and `b`. If `a` is larger than `b`, then a positive value goes back to the `sort()` method, telling it to sort `a` later than `b` (that is, position `a` at a higher value index position than `b`). Therefore, `b` may end up at `myArray[0]`, whereas `a` ends up at a higher index-valued location. On the other hand, if `a` is smaller than `b`, the returned negative value tells `sort()` to put `a` in a lower index value spot than `b`.

Evaluations within the comparison function can go to great lengths, as long as some data connected with array values can be compared. For example, instead of numerical comparisons, as just shown,

you can perform string comparisons. The following function sorts alphabetically by the last character of each array string entry:

```
function compare(a,b)
{
   // last character of array strings
   var aComp = a.charAt(a.length - 1);
   var bComp = b.charAt(b.length - 1);
   if (aComp < bComp)
      return -1;
   if (aComp > bComp)
      return 1;
   return 0;
}
```

First, this function extracts the final character from each of the two values passed to it. Then, because strings cannot be added or subtracted like numbers, you compare the ASCII values of the two characters, returning the corresponding values to the `sort()` method to let it know how to treat the two values being checked at that instant.

When an array's entries happen to be objects, you can even sort by properties of those objects. If you bear in mind that the `a` and `b` parameters of the sort function are references to two array entries, then by extension you can refer to properties of those objects. For example, if an array contains objects whose properties define information about employees, one of the properties of those objects can be the employee's age as a string. You can then sort the array based on the numeric equivalent of the `age` property of the objects by way of the following comparison function:

```
function compare(a,b) {
   return parseInt(a.age) - parseInt(b.age);
}
```

Array sorting, unlike sorting routines you may find in other scripting languages, is not a stable sort. Not being stable means that succeeding sort routines on the same array are not cumulative. Also, remember that sorting changes the sort order of the original array. If you don't want the original array harmed, make a copy of it before sorting or reload the document to restore an array to its original order. Should an array element be `null`, the method sorts such elements at the end of the sorted array.

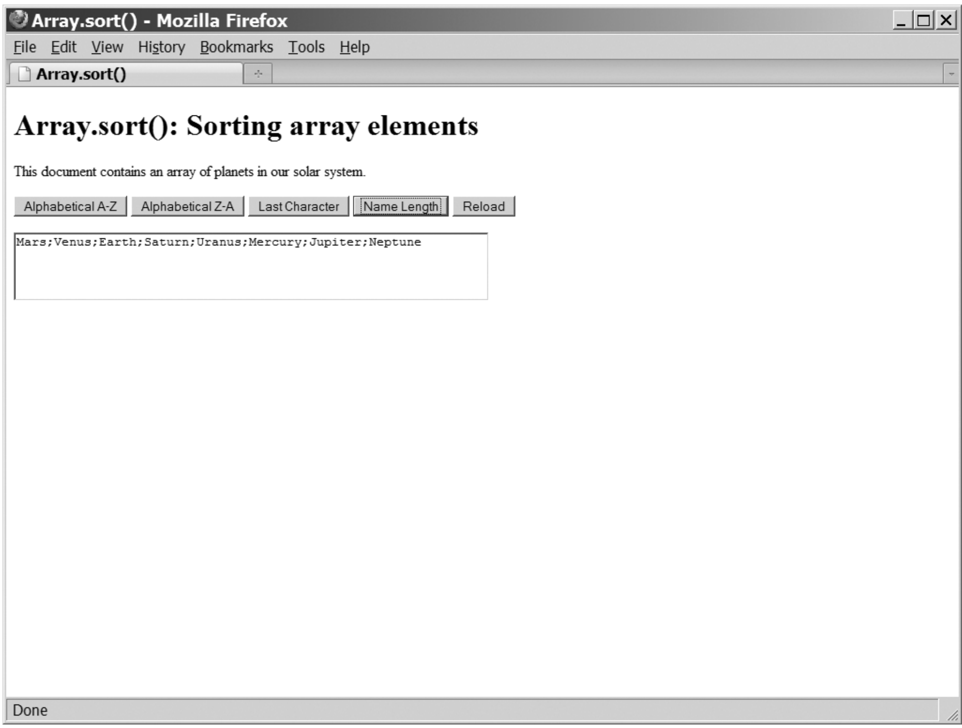
JavaScript array sorting is extremely powerful stuff. Array sorting is one reason why it's not uncommon to take the time during the loading of a page containing an IE XML data island, for example, to make a JavaScript copy of the data as an array of objects (see Chapter 60, "Application: Transforming XML Data," on the CD-ROM). Converting the XML to JavaScript arrays makes the job of sorting the data much easier and faster than cobbling together your own sorting routines on the XML elements.

Example

You can look to Listing 18-8 for a few examples of sorting an array of string values (see Figure 18-2). Four buttons summon different sorting routines, three of which invoke comparison functions. This listing sorts the planet array alphabetically (forward and backward) by the last character of the planet name and also by the length of the planet name. Each comparison function demonstrates different ways of comparing data sent during a sort.

FIGURE 18-2

Sorting an array of planet names alphabetically by name length.



LISTING 18-8

Array.sort() Possibilities

HTML: jsb-18-08.html

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Array.sort()</title>
      <script type="text/javascript" src="../jsb-global.js"></script>
      <script type="text/javascript" src="jsb-18-08.js"></script>
   </head>
   <body>
      <h1>Array.sort(): Sorting array elements</h1>
      <form>
         <p>This document contains an array of planets in our solar system.</p>
         <p>
            <input type="button" id="sortAsc" value="Alphabetical A-Z">
            <input type="button" id="sortDesc" value="Alphabetical Z-A">
```

continued

LISTING 18-8 *(continued)*

```
            <input type="button" id="sortLastChar" value="Last Character">
            <input type="button" id="sortLength" value="Name Length">
            <input type="button" id="reload" value="Reload">
         </p>
         <p>
            <textarea id="output" name="output" rows="3" cols="60"
                          wrap="virtual"></textarea>
         </p>
      </form>
   </body>
</html>
```

JavaScript: jsb-18-08.js

```
// initialize when the page has loaded
addEvent(window, 'load', initialize);

function initialize()
{
   // do this only if the browser can handle DOM methods
   if (document.getElementById)
   {
      // point to critical elements
      oOutput = document.getElementById('output');
      var oButtonSortAsc = document.getElementById('sortAsc');
      var oButtonSortDesc = document.getElementById('sortDesc');
      var oButtonSortLastChar = document.getElementById('sortLastChar');
      var oButtonSortLength = document.getElementById('sortLength');
      var oReload = document.getElementById('reload');

         // if they all exist...
         if (oOutput && oButtonSortAsc && oButtonSortDesc && oButtonSortLastChar
                     && oButtonSortLength && oReload)
         {
            // apply behaviors to buttons
            oButtonSortAsc.onclick = function() { sortIt(null) };
            oButtonSortDesc.onclick = function() { sortIt(compare1) };
            oButtonSortLastChar.onclick = function() { sortIt(compare2) };
            oButtonSortLength.onclick = function() { sortIt(compare3) };
            oReload.onclick = function() { location.reload(); };

            // set global array
            solarSys = new Array("Mercury", "Venus", "Earth", "Mars", "Jupiter",
                                 "Saturn", "Uranus", "Neptune");
         }
   }
}
```

```
// comparison functions
function compare1(a,b)
{
   // reverse alphabetical order
   if (a > b)
      return -1;
   if (b > a)
      return 1;
   return 0;
}

function compare2(a,b)
{
   // last character of planet names
   var aComp = a.charAt(a.length - 1);
   var bComp = b.charAt(b.length - 1);
   if (aComp < bComp)
      return -1;
   if (aComp > bComp)
      return 1;
   return 0;
}

function compare3(a,b)
{
   // length of planet names
   return a.length - b.length;
}

// sort and display array
function sortIt(compFunc)
{
   if (compFunc == null)
   {
         solarSys.sort();
   }
   else
   {
         solarSys.sort(compFunc);
   }

   // display results in field
   var delimiter = ";";
   oOutput.value = decodeURIComponent(solarSys.join(delimiter));
}
```

Related Items: `array.reverse()` method

Note

As we show you in Chapter 45, many regular expression object methods generate arrays as their result (for example, an array of matching values in a string). These special arrays have a custom set of named properties that assist your script in analyzing the findings of the method. Beyond that, these regular expression result arrays behave like all others. ■

array.splice(*startIndex* , *deleteCount*[, *item1*[, *item2*[,...*itemN*]]])

Returns: Array

Compatibility: WinIE5.5+, MacIE−, NN4+, Moz+, Safari+, Opera+, Chrome+

If you need to remove items from the middle of an array, the `array.splice()` method simplifies a task that would otherwise require assembling a new array from selected items of the original array. The first of two required parameters is a zero-based index integer that points to the first item to be removed from the current array. The second parameter is another integer that indicates how many sequential items are to be removed from the array. Removing array items affects the length of the array, and those items that are removed are returned by the `splice()` method as their own array.

You can also use the `splice()` method to replace array items. Optional parameters beginning with the third let you provide data elements that are to be inserted into the array in place of the items being removed. Each added item can be any JavaScript data type, and the number of new items does not have to be equal to the number of items removed. In fact, by specifying a second parameter of zero, you can use `splice()` to insert one or more items into any position of the array.

Example

Use The Evaluator (see Chapter 4) to experiment with the `splice()` method. Begin by creating an array with a sequence of numbers:

```
a = new Array(1,2,3,4,5)
```

Next, remove the center three items, and replace them with one string item:

```
a.splice(1, 3, "two/three/four")
```

The Results box shows a string version of the three-item array returned by the method. To view the current contents of the array, enter `a` into the top text box.

To put the original numbers back into the array, swap the string item with three numeric items:

```
a.splice(1, 1, 2, 3, 4)
```

The method returns the single string, and the `a` array now has five items in it again.

Related Item: `array.slice()` method

array.toLocaleString()

Returns: String

Compatibility: WinIE5.5+, MacIE−, NN6+, Moz+, Safari+, Opera+, Chrome+

array.toString()

Returns: String

Compatibility: WinIE4+, MacIE4+, NN3+, Moz+, Safari+, Opera+, Chrome+

The `array.toLocaleString()` and the older, more compatible `array.toString()` are methods to retrieve the contents of an array in string form. Browsers use the `toString()` method on their own whenever you attempt to display an array in text boxes, in which case the array items are comma-delimited.

The precise string conversion of the `toLocaleString()` is left up to the specific browser implementation. That browsers differ in some details is not surprising, even in the U.S. English versions of operating systems and browsers. For example, if the array contains integer values, the `toLocaleString()` method in IE5.5+ returns the numbers comma-and-space-delimited, formatted with two digits to the right of the decimal (as if dollars and cents). Mozilla-based browsers, on the other hand, return just the integers, but these are also comma-delimited.

If you need to convert an array to a string for purposes of passing array data to other venues (for example, as data in a hidden text box submitted to a server or as search string data conveyed to another page), use the `array.join()` method instead. `Array.join()` gives you more reliable and flexible control over the item delimiters, and you are assured of the same results regardless of locale.

Related Item: `array.join()` method

Array Comprehensions

Compatibility: FF2+

A feature of JavaScript 1.7 is an enhancement to JavaScript syntax called *array comprehensions*, a shortcut for building a new array based on another. Comprehensions can often replace the `map()` and `filter()` methods.

Here's an example that uses a comprehension to produce a modified version of every element in an array:

```
var aIntegers = new Array(0, 1, 2, 3, 4, 5, 6, 7);
var aResults = [i * i for each (i in aIntegers)];

// result: aResults == [0, 1, 4, 9, 16, 25, 36, 49]
```

This is the equivalent of the `map()` syntax:

```
function square(iValue)
{
   return iValue * iValue;
}

var aIntegers = new Array(0, 1, 2, 3, 4, 5, 6, 7);
var aResults = aIntegers.map(square);
```

Array comprehensions can also be used to select certain elements from an array:

```
var aNumbers = new Array(1, -2, -3, 4, 5, -6);
var aResult = [i for each (i in aNumbers) if (i > 0)];

// result: aResult == [1, 4, 5]
```

This is the equivalent of the `filter()` syntax:

```
function isPositive(iValue, iIndex, aArray)
{
   return (iValue > 0);
}

var aNumbers = new Array(1, -2, -3, 4, 5, -6);
var aResult = aNumbers.filter(isPositive);
```

The square brackets of an array comprehension introduce an implicit block for scoping purposes. New variables (such as `i` in the preceding examples) have local scope (are treated as if they had been declared using `let`) and are not available outside of the comprehension.

The input to an array comprehension does not itself need to be an array; *iterators* and *generators* can also be used (see Chapter 21).

Destructuring Assignment

Compatibility: FF2+, Opera 9.5 (partial)

Destructuring assignment (introduced with JavaScript 1.7 and corrected in 1.8) makes it easy to extract an array from another array, object, or function. You can view it as setting one array equal to another:

```
[a, b, c] = [1, 2, 3]
// result: a == 1, b == 2, and c == 3
```

This syntax provides a succinct way to swap the values of variables:

```
var x = 1;
var y = 2;

[x, y] = [y, x]
// result: x == 2 and y == 1
```

... replacing the more conventional technique that employs a third temporary variable to swap two values:

```
temp = x;
x = y;
y = temp;
```

Another example is receiving an array of values from a function:

```
function fun()
{
   return [3, 'aardvark', 0];
}

[a, b, c] = fun()
// result: a == 3, b == 'aardvark', and c == 0
```

As of this writing, destructuring assignment is supported by Firefox 2.0+ and partially by Opera 9.5 but not at all by Internet Explorer 8 and other browsers. We can't use JavaScript to test a browser's ability to execute this code because an older version will simply crash trying to compile it. We can "firewall" this code with a special script tag:

```
<script type="application/javascript;version=1.8" src="scriptname.js"></script>
```

However, we question the usefulness of this if we also have to provide more conventional syntax for other browsers. Unless we're in a special circumstance in which we can guarantee that our code will be running in a browser that incorporates JavaScript 1.8, we may just have to wait until destructuring assignment syntax is more universally adopted by the other browser manufacturers.

Compatibility with Older Browsers

Some of the features described in this chapter were introduced with JavaScript 1.6 and later. They run in Firefox 2.0+ or 3.0+ and occasional others but are not supported in many other browsers, including Internet Explorer 8. New array object methods such as `every()` and `filter()` can be used in our cross-browser scripts simply by adding them to the object prototype when they're not already there, as we've listed in this chapter. However, innovations of syntax, including array comprehensions and destructuring assignment, cannot be included in blocks of script run by legacy browsers because older versions of JavaScript will simply stop cold during the compile phase and will not run any of the script on the page.

Should we then refrain from using the new syntax until it's universally supported?

As mentioned in Chapter 4, there is a way to identify a block of code so that it will run only in a given version of JavaScript or greater — for example:

```
<script type="text/javascript" src="for-everyone.js"></script>
<script type="application/javascript;version=1.7" src="new-stuff.js"></script>
```

You can isolate sections of code in this way to execute only if the browser is capable, separated safely from older user agents.

The question then comes, what alternative code will you run if the browser is older? If you have to write code that performs the same operation as an array comprehension to support older browsers, why not simply use that code for all browsers?

As with any facet of web development, and JavaScript in particular, make sure that all basic functionality on a page is intact even in the absence of JavaScript or any specific features of the language; then add enhancements when scripting and any of its advanced features are available. This principle of progressive enhancement ensures that while your pages may work better or faster for those with fancier tools, they will work fundamentally for everyone.

CHAPTER 19

JSON — Native JavaScript Object Notation

JSON (JavaScript Object Notation) is a lightweight way to represent data structures that many programming languages can easily read and write. This makes it (similarly to XML) a useful data interchange format among applications written in a variety of languages, for example, between server-side PHP and client-side JavaScript. With the ECMAScript Fifth Edition standard, JSON is now formally part of the language and will be increasingly supported by more browsers over the coming years. In the meantime, there is freely available code to assist older browsers.

JSON was developed by Douglas Crockford and presented to the Internet community in 2006. It was formally made part of the ECMAScript standard, on which JavaScript is based, as of the Fifth Edition finalized in April 2010.

IN THIS CHAPTER

- **How JSON works**
- **Sending and receiving JSON data**
- **JSON object methods**
- **Security concerns**

How JSON Works

JSON combines two aspects of JavaScript syntax for representing arrays and objects to present data structures — simple array notation using brackets `[ ]`:

```
[ "item1", "item2", "item3" ]
```

and the notation for objects and associative arrays using curly braces `{ }`:

```
{ "key": "value" }
```

We have access to the embedded values by numerical index in the first case (remember that the first item has an index of zero):

```
var aItems = [ "item1", "item2", "item3" ];
// e.g., aItems[0] == "item1"
```

and by key name in the second:

```
var oExample = { "charlie": "horse" };
// oExample["charlie"] == "horse"
// oExample.charlie == "horse"
```

Using these two conventions, one can describe many data structures with sub-records having either named or numerically indexed keys:

```
var oNovelist =
{
   "firstName": "Joanna",
   "lastName": "Russ",
   "novels":
   [
      {
         "title": "And Choas Died",
         "year": "1970"
      },
      {
         "title": "The Female Man",
         "year": "1975"
      }
   ]
};
```

In this example, `firstName` and `lastName` have simple string values, while `novels` is a numerically indexed array. Here are two ways to refer to these values:

```
var sMsg = oNovelist['firstName'] + ' ' + oNovelist['lastName'] + "'s novel " +
 oNovelist['novels'][0]['title'] + ' was published in ' +
 oNovelist['novels'][0]['year'] + '.';

var sMsg = oNovelist.firstName + ' ' + oNovelist.lastName + "'s novel " +
 oNovelist.novels[0].title + ' was published in ' +
 oNovelist.novels[0].year + '.';

// result: Joanna Russ's novel And Chaos Died was published in 1970.
```

In the first example, we're treating the structure as an associative array with bracketed keys. In the second, we're using dot-notation to retrieve the object's properties. The first syntax will always work because the keys, when strings, are quoted. We got away with using dot-notation in the second example because none of these keys contained characters such as hyphens, periods, and other characters that are legal key values but would throw JavaScript a curve:

```
var sMsg = oNovelist['first-name'];   // this works
var sMsg = oNovelist.first-name;      // oh no! JavaScript tries to subtract
```

Cross-Reference

For more on this point, see the section "Simulating a Hash Table" in Chapter 18, "The Array Object." ■

Data types

Six types of data can occur as a JSON value:

- String (a collection of zero or more Unicode characters, wrapped in double quotes, using backslash escapes)
- Number (integer or real; no octal and hexadecimal formats)

- Object (a collection of key:value pairs, comma-separated and enclosed in curly braces)
- Array (an ordered sequence of values, comma-separated and enclosed in square brackets)
- Boolean (`true` or `false`)
- `null`

Whitespace can be inserted between any pair of tokens.

Sending and Receiving JSON Data

Because JSON grammar uses JavaScript's normal object and array syntax, we don't need any add-ons to the language in order to be able to build and read JSON data structures within a single script. It's when we need to share these structures between scripts that complications arise.

In order to read, write, send, and receive a JSON data object, we need to convert it to and from a string. The standard way of doing this is to read and write exactly the same syntax we'd use in JavaScript:

```
{"firstName":"Joanna","lastName":"Russ","novels":[{"title":"And Choas
Died","year":"1970"},{"title":"The Female Man","year":"1975"}]}
```

This string can then be imported into any of the programming languages that work with JSON.

Performing this conversion between object and string in JavaScript is simple enough. There are functions at `http://json.org` that we're encouraged to use for older browsers so that we don't have to reinvent the wheel.

Because the JSON data is stored in a string exactly as it is in JavaScript source code, on first glance it might seem that the most expedient way to convert it from a string back to an object is by using the JavaScript method `eval()`:

```
var sStringJSON = readJSONDataFromWhateverSource();

eval("var oJSON = " + sStringJSON);
// or:
var oJSON = eval('(' + sStringJSON + ')');
```

Now `oJSON.firstName` will yield 'Joanna' for the example above. (In the second use of `eval()`, note the necessary inclusion of parentheses around the JSON string to satisfy the JavaScript interpreter.)

However, we can't over-emphasize that `eval()` opens up a doorway for malicious exploits in JavaScript and should be avoided on any public page. A proper JSON parser will walk through the data string character by character to build the JSON object rather than yielding to the temptation to use `eval()`.

JSON goes native

The JSON object was made part of the language with the ECMAScript Fifth Edition standard, published in December 2009. Mozilla implemented this in JavaScript 1.8.1, in the Gecko layout engine version 1.9.1 that drives Firefox 3.5, Thunderbird 3, and SeaMonkey 2. Other browser development teams are sure to follow their lead.

The JSON object has two methods, `parse()` and `stringify()`. The `toJSON()` method is also added to the Date object.

JSON Object

Properties	Methods
	`parse()`
	`stringify()`

Methods

`JSON.parse(`*`string [, translator]`*`)`

Returns: JSON object reference

Compatibility: Moz1.9.1+, Firefox3.5+

Use the `JSON.parse()` method to convert a JSON string to a JSON object.

```
object = JSON.parse(string, translator)
```

The `string` parameter is the "stringified" JSON to be converted to an object.

Related Item: `JSON.stringify()` method

`JSON.stringify(`*`object [, replacer [, space]]`*`)`

Returns: JSON string

Compatibility: Moz1.9.1+, Firefox3.5+

`JSON.stringify()` converts a JSON object to a JSON string.

```
string = JSON.stringify(object, replacer, space)
```

The `object` parameter is the JSON object to be converted to a string.

`replacer` is an optional parameter that is either a function that alters the behavior of the stringification process or an array of String and Number objects that serve as a whitelist for selecting the properties of the object to be stringified. If this value is null or not provided, all properties of the object are included in the resulting JSON string.

`space` is a String or Number object that's used to insert white space into the output JSON string for readability purposes. If this is a Number, it indicates the number of space characters to use as white space; this number is capped at 10 if it's larger than that. Values less than 1 indicate that no space should be used. If this is a String, the string (or the first 10 characters of the string, if it's longer than that) is used as white space. If this parameter is not provided (or is null), no white space is used.

Related Item: `JSON.parse()` method

Security Concerns

Some concern has been expressed about the vulnerability JSON presents in web security. We feel that these issues can be dismissed with sufficiently careful use of the technology.

Using `eval()` to convert a JSON string to an object (or for any other purpose on a public page) is an easy pitfall to avoid. If a method (function) is embedded in serialized JSON and `eval()` is used to convert it, JavaScript code can be executed without the permission of the page author's script. The obvious solution is to use a non-`eval()` parsing algorithm, such as that published by Douglas Crockford, on older browsers and the new JSON object method `parse()` on newer browsers.

The DOM method `XMLHttpRequest()` is commonly used to exchange information between server and client. By default it permits exchanges only with the server from which the current page has originated. However, by injecting a `script` tag that links to an off-site JavaScript file, this protection can be circumvented. The solution here is to be vigilant when it comes to allowing JavaScript from third parties to execute on your page.

CHAPTER 20

E4X — Native XML Processing

IN THIS CHAPTER

Using the XML object

Adding elements and attributes

Selecting records

Serializing XML objects to text

With the rise in popularity of Ajax (Asynchronous JavaScript and XML) connectivity in recent years, it's become commonplace for web developers who might otherwise never stray beyond the bounds of HTML to work with XML (Extensible Markup Language). In fact, XML has been considered so fundamental to everyday web development that the ECMA has incorporated it into the very syntactical fabric of JavaScript. We can now write unquoted XML markup right into our JavaScript statements — as permitted by the most recent browsers.

This new feature of JavaScript syntax places XML uniquely on both sides of a major divide. Every other aspect of scripting is either part of the core JavaScript language or part of the DOM (Document Object Model). XML is now a member of both camps.

We discuss techniques for transmitting XML between client and server in Chapter 39, "Ajax, E4X, and XML" (on the CD).

XML

Briefly, XML is a markup language designed to contain and transmit textual data. It consists of nested elements marked up with three types of tag — matching begin- and end-tags

```
<hobby>music</hobby>
```

and empty-element tags

```
<breathe/>
```

Elements can contain attributes, which are name/value pairs that reside within the opening or empty tag:

```
<fax location="home" call-first="true">123-456-7890</fax>;
```

Unlike HTML and XHTML, in which element and attribute name choices are constrained by the Document Type Declaration or XMLSchema, XML element and attribute names are completely arbitrary and defined by the author of the data to suit the current purpose.

For a complete formal description of XML, see the W3C specification at `http://w3.org/TR/REC-xml/`.

ECMAScript for XML (E4X)

Prior to E4X, the only way to access XML documents with JavaScript was through the DOM at the object level. With the advent of E4X, XML is treated as a primitive, at the same level as strings, numbers, and Booleans. What does this mean to us? The short story is that XML access is now simpler for us as web developers, and faster for the end user. An `XML` object created with the XML primitive is not part of the DOM; neither is it a DOM representation of XML. Because it is a primitive, you use it in ways similar to how you use other JavaScript objects.

Using the XML object

With E4X you declare an `XML` object, in much the same way that you declare a string or numeric or Boolean object, with the `XML` constructor:

```
var oCopyright = new XML();
```

You can initialize the object using text:

```
var sXMLText = "<copyright>"
                + "<date>2010</date>"
                + "<author>Danny Goodman</author>"
                + "</copyright>";

var oCopyright = new XML(sXMLText);
```

Remarkably, you can also use unquoted XML markup in an assignment statement, which implicitly creates the XML object:

```
var oCopyright = <copyright>
                    <date>2010</date>
                    <author>Danny Goodman</author>
                 </copyright>;
```

This type of syntax, in which specially constructed text can be assigned without quotation marks, may be reminiscent of JavaScript's native regular expression syntax, as described in Chapter 45, "The Regular Expression and RegExp Objects."

XML elements

The XML elements within the `XML` object are its properties. You can use both the familiar `object.property` dot-notation and the `object[property]` bracket-notation to reference elements and their values:

```
var sText = "Copyright " + oCopyright.date + " by " + oCopyright.author;
var sText = "Copyright " + oCopyright['date'] + " by " + oCopyright['author'];
```

The outermost XML element is the root of the object, so the first top-level property within our `oCopyright` object above is the `date` element:

```
alert(oCopyright.name());           // result = 'copyright'
alert(oCopyright.child(0).name());  // result = 'date'
alert(oCopyright.date);             // result = '2010'
```

Of course, as with other JavaScript objects, you won't be able to use object dot-notation successfully when an element tag name contains any characters that would confuse JavaScript. For example:

```
var oXML = <author>
              <first-name>Danny</first-name>
           </author>;

var sText = oXML['first-name'];   // result: Danny
var sText = oXML.first-name;      // result: JavaScript stops cold!
```

In the preceding example, JavaScript attempts to subtract the value of a variable called `name` from the value of the XML element `oXML.first`. If either one doesn't exist or if they do exist but don't both have numerical values that can be subtracted, JavaScript fails. The moral of the story: if you want to use dot notation and you are guaranteed to have complete control over the elements' tag names, you can restrict their character set to alphanumerics and underscores. If you're working with data coming in from a source that you might not always control, use bracket-quote notation, which will always succeed.

Adding elements

The `appendChild()` method of the `XML` object works similarly to the DOM method of the same name, inserting a child element into the referent:

```
var oBook = <book/>;
var oAuthor = <author/>;
oBook.appendChild(oAuthor);

result: <book>
           <author/>
        </book>
```

Elements can also be appended to an XML element using the same concatenation syntax that we typically use with strings:

```
oCopyright += <addendum>All rights reserved.</addendum>;
```

This results in:

```
<copyright>
   <date>2010</date>
   <author>Danny Goodman</author>
</copyright>;
<addendum>All rights reserved.<addendum>
```

In order to insert the new element inside the `copyright` element, we need to write:

```
oCopyright.author += <addendum>All rights reserved.</addendum>;
```

resulting in:

```
<copyright>
   <date>2010</date>
   <author>Danny Goodman</author>
   <addendum>All rights reserved.<addendum>
</copyright>
```

Embedded JavaScript expressions

E4X becomes really exciting when you want the `XML` object to dynamically interact with JavaScript expressions. You do this with curly braces:

```
var oDate = new Date();

var oCopyright = <copyright>
                    <date>{oDate.getFullYear()}</date>
                    <author>Danny Goodman</author>
                 </copyright>;
```

Whatever is inside the curly braces is evaluated as a JavaScript expression and its result plugged into the XML stream. In the year 2020, the preceding example will resolve to:

```
<copyright>
   <date>2020</date>
   <author>Danny Goodman</author>
</copyright>;
```

XML element attributes

Just as in XML documents, the elements in your `XML` objects can have attributes. Those attribute values can be set dynamically with curly brace notation as well. You can extract the value of an element's attribute by using dot notation with the @ operator in front of the attribute name, as we do with the `location` attribute in this example:

```
var oPhonebook = <phonebook>
                    <entry>
                       <name>Janis Joplin</name>
                       <phone location="home">123-456-7890</phone>
                    </entry>
                 </phonebook>;

var sText = oPhonebook.entry.name + "'s "
            + oPhonebook.entry.phone.@location + " phone is "
            + oPhonebook.entry.phone + ".";
// or:
var sText = oPhonebook['entry']['name'] + "'s "
            + oPhonebook['entry']['phone']['@type'] + " phone is "
            + oPhonebook['entry']['phone'] + ".";

// result == Janis Joplin's home phone is 123-456-7890.
```

You can also use the @ notation to create and modify attribute values:

```
oPhonebook.entry.phone.@location = "work";
```

The XMLList object

Most XML documents contain repeating sets of elements. You can create such sets in your XML object simply by repeating a tag name for elements with the same parent:

```
var oBook = <book>
               <title>The JavaScript Bible 7th Edition</title>
               <author>Danny Goodman</author>
               <coauthor>Michael Morrison</coauthor>
               <coauthor>Paul Novitski</coauthor>
               <coauthor>Cynthia Gustaff Rayl</coauthor>
            </book>;
```

When you point to the repeating set, you get back an XMLList object. It can be processed almost the same as you would an array. There are two primary differences: length() is a method and not a property, and therefore requires the parentheses (); and other Array object methods are not supported by XMLList.

```
var sText = "The number of co-authors is " + oBook.coauthor.length() + ".";
```

In our current example XML, oBook.coauthor comprises a list within the overall data structure because there is more than one sibling element with the same tag name.

```
// extract the XMLList of coauthor elements
var oCoauthors = oBook.coauthor;

var sText = oBook.title + ' was written by ' + oBook.author + ' with ';

// loop through co-authors
var sConj = '';
var iCoauthors = oCoauthors.length();
var sLastAuthor = oCoauthors[iCoauthors-1];

for (var i=0; i < iCoauthors; i++)
{
      // comma before item if more than two items
      if (i > 0 && iCoauthors > 2)
      {
         sConj = ', ';
      }
      // 'and' before item if last in a series
      if (i == iCoauthors-1 && iCoauthors > 1)
      {
         sConj += ' and ';
      }

   // add item to list
   sText += sConj + oCoauthors[i];
}
```

```
sText += '.';

// result == The JavaScript Bible 7th Edition was written by Danny Goodman
             with Michael Morrison, Paul Novitski, and Cynthia Gustaff Rayl.
```

The asterisk (*) can be used to represent all items in a collection. Using the preceding XML:

```
var iCount = oBook.*.length();
```

The result is 5 — the root parent has a `title`, an `author`, and three `coauthor` elements.

Adding list items

To add a new item to an `XMLList`, use the same concatenation syntax as mentioned above and name the list by its repeating element:

```
oBook.coauthor += <coauthor>John Lennon</coauthor>;
```

resulting in:

```
<book>
   <title>The JavaScript Bible 7th Edition</title>
   <author>Danny Goodman</author>
   <coauthor>Michael Morrison</coauthor>
   <coauthor>Paul Novitski</coauthor>
   <coauthor>Cynthia Gustaff Rayl</coauthor>
   <coauthor>John Lennon</coauthor>
</book>
```

Note

A cautionary note: if you extract an `XMLList` **object from an** `XML` **object as we did above:**

```
// extract the XMLList of coauthor elements
var oCoauthors = oBook.c  oauthor;
```

the list is a static copy of the XML elements with no dynamic linkage. Any subsequent change to either the `XML` **object or the** `XMLList` **object will not be reflected in the other. In this example, adding a new** `coauthor` **element to the** `oCoauthors` **list does not modify the original** `XML` **object.** ■

Selecting list items

The power of E4X really shines in its ability to select `XMLList` items based on attributes and values. Consider the following example:

```
var oPhonebook = <phonebook>
                    <entry>
                       <name>Janis Joplin</name>
                       <phone location="home">123-456-7890</phone>
                       <phone location="work">234-567-8901</phone>
                       <phone location="cell">345-678-9012</phone>
                    </entry>
                    <entry>
```

```
        <name>John Lennon</name>
        <phone location="home">987-654-3210</phone>
        <phone location="work">876-543-2109</phone>
      </entry>
    </phonebook>;
```

We've seen that we can select a list of all elements of a matching name quite easily:

```
// extract a list of all the phone elements
var oPhones = oPhonebook.entry.phone;

result:
<phone location="home">123-456-7890</phone>
<phone location="work">234-567-8901</phone>
<phone location="cell">345-678-9012</phone>
<phone location="home">987-654-3210</phone>
<phone location="work">876-543-2109</phone>
```

Note that this finds the phone elements in more than one `entry` record. The selection statement essentially says, "Find all the `phone` elements whose parent is a root `entry` element." If we wanted to limit the search to a single `entry` element, we need only specify its index:

```
var oPhones = oPhonebook.entry[0].phone;
```

Similarly, we can select elements by attribute value:

```
var oWorkPhones = oPhonebook.entry.phone.(@location == 'work');

result:
<phone location="work">234-567-8901</phone>
<phone location="work">876-543-2109</phone>
```

This narrows the selection to just those phone elements that have a type attribute of "work" — when the expression in parentheses evaluates to `true`. Note that we're using the double equal sign == of an equality comparison, not the single = of a value assignment!

One of the handy aspects of this syntax is being able to extract the value of one element after searching for its sibling:

```
var oPhone = oPhonebook.entry.(name == 'Janis Joplin').phone.
  (@location == 'home');

result:
<phone location="home">123-456-7890</phone>
```

The selector says, "Get me the home phone number of the entry with the name 'Janis Joplin.'" The statement locates an `entry` with a child element name that has the value '`Janis Joplin`' but then returns a `phone` element that is the sibling of the `name` element. Is that cool or what? Of course, like so many dead rock stars, Janis currently has an unlisted number, but we can always dream.

XML selector expressions can contain JavaScript functions, and without having to use curly brace syntax:

```
function checkExchange(sExchange, sPhone)
{
```

```
      sExchange = '-' + sExchange + '-';
      return (sPhone.indexOf(sExchange) > 0);
   }
   var sPhone = oPhonebook.entry.(checkExchange('543', phone.
     (@location=='work'))).name;
```

This returns "John Lennon" since we've selected the entry in which the `checkExchange()` function returns true.

Serializing XML objects

XML can be a handy format for keeping track of information within a script, but its primary use is communicating structured data between programs. To send an XML structure to the server, we translate the JavaScript object into a text string. The `XML` object has two interchangeable methods for doing this — `toXMLString()` and `toString()`.These methods can operate on any subset of the object as well as the whole thing at once:

```
var sXMLText = oPhonebook.toXMLString();
var sXMLText = oPhonebook.entry.(name == 'John Lennon').toString();
```

Embedding E4X in HTML

Although you can use the standard MIME type with E4X

```
<script type="text/javascript">
```

you might end up with inexplicable syntax errors. This usually occurs for one of two reasons: you are using HTML comments to hide script from older browsers; or you are putting your scripts into XML CDATA sections. If either of these is the case, then you can add the E4X argument to the standard MIME type:

```
<script type="text/javascript;e4x=1">
```

Methods

`child()`

Returns: XML object

Compatibility: WinIE5-, MacIE-, NN-, FF3.5+, Safari-, Opera-, Chrome-

The `child()` method returns an XML element that is an immediate descendant of the referent:

```
var oPhonebook = <phonebook>
                    <person>
                       <name>Janis Joplin</name>
                       <phone location="home">123-456-7890</phone>
                    </person>
                 </phonebook>;

alert(oPhonebook.person.child(1));        // result = '123-456-7890'
```

Remember that `child()` is a method, even though we use it much as we would an array, so use parentheses around the index, not brackets. Like an array, the `child()` collection is a zero-based list, so `child(1)` finds the second child element. Unlike an array, we cannot use `forEach` syntax.

Related Item: `childNodes()` DOM property

copy()

Returns: XML object

Compatibility: WinIE5-, MacIE-, NN-, FF3.5+, Safari-, Opera-, Chrome-

The `copy()` method returns a deep copy of the selected portion of an `XML` object. For example:

```
var oPhonebook = <phonebook>
                    <person>
                       <name>Janis Joplin</name>
                       <phone location="home">123-456-7890</phone>
                    </person>
                 </phonebook>;

var oPerson = oPhonebook.person.copy();
```

The result of this operation is the following `XML` object:

```
<person>
   <name>Janis Joplin</name>
   <phone location="home">123-456-7890</phone>
</person>
```

Note that this can also be accomplished using a simple assignment statement:

```
var oPerson = oPhonebook.person;
```

name()

Returns: String

Compatibility: WinIE5-, MacIE-, NN-, FF3.5+, Safari-, Opera-, Chrome-

The `name()` method returns a string that is the XML element's tag name:

```
var oPhonebook = <phonebook>
                    <person>
                       <name>Janis Joplin</name>
                       <phone location="home">123-456-7890</phone>
                    </person>
                 </phonebook>;

alert(oPhonebook.child(0).name());          // result = 'person'
alert(oPhonebook.person.child(1).name());   // result = 'phone'
alert(oPhonebook.person.child(1));          // result = '123-456-7890'
```

Remember that `child()` is a method, even though we use it much as we would an array, so use parentheses around the index, not brackets. Like an array, the `child()` collection is a zero-based list, so `child(1)` finds the second child element.

Related Item: `childNodes()` DOM property

toString(), toXMLString()

Returns: String

Compatibility: WinIE5-, MacIE-, NN-, FF1.5+, Safari-, Opera-, Chrome-

The `toString()` and `toXMLString()` methods return a string that is the concatenated text of all the text nodes of the XML object.

Related Item: *string*.toString()

CHAPTER

21

Control Structures and Exception Handling

IN THIS CHAPTER

Branching script execution down multiple paths

Looping through ordered collections of data

Applying exception handling techniques

You get up in the morning, go about your day's business, and then end the day by walking the dog, changing into comfy footwear, and relaxing. That's not much different from what a program does from the time it starts to the time it ends. But along the way, both you and a program take lots of tiny steps, not all of which advance the processing in a straight line. At times, you have to control what's going on by making a decision or repeating tasks until the whole job is finished. Control structures are the facilities that make these tasks possible in JavaScript.

Control structures in JavaScript follow along the same lines as they do in many programming languages. Basic decision-making and looping constructions satisfy the needs of just about all programming tasks.

Another vital program control mechanism — error (or exception) handling — is formally addressed in the ECMA-262 language standard. The concept of exception handling was added to the JavaScript version introduced in IE5.5 and NN6, but it is well known to programmers in many other environments. Adopting exception-handling techniques in your code can greatly enhance recovery from processing errors beyond your control, such as those caused by errant user input or network glitches.

If and If. . .Else Decisions

Compatibility: WinIE3+, MacIE3+, NN2+, Moz+, Safari+, Opera+, Chrome+

JavaScript programs frequently have to make decisions based on the current values of variables or object properties. Such decisions can have only two possible outcomes at a time. The factor that determines the path that the program takes at these decision points is if a statement evaluates to true or false. For example, when you come back from walking the dog at night, the statement under test is something such as, "Is my comfy footwear in the living room?" If that statement is false, you look for your comfy footwear; if that statement is true, you put on your comfy footwear and relax.

Simple decisions

JavaScript syntax for this kind of simple decision always begins with the keyword `if`, followed by the condition to test, and then the statements that execute if the condition yields a true result. JavaScript uses no "then" keyword (as some other languages do); the keyword is implied by the way parentheses and braces surround the various components of this construction. The formal syntax is:

```
if (condition)
{
   statementsIfTrue
}
```

This construction means that if the condition is true, program execution takes a detour to execute statements inside the braces. No matter what happens, the program continues executing statements beyond the closing brace (`}`). If a comfy footwear search were part of the scripting language, the code would look like this:

```
if (comfyFootwearInLivingRoom == false)
{
   look for them
}
```

If you're not used to C/C++, the double equals sign may have caught your eye. You learn more about this type of operator in the next chapter, but for now, know that this operator compares the equality of items on either side of it. In other words, the `condition` statement of an `if` construction must always yield a Boolean (`true` or `false`) value. Some object properties, you may recall, are Booleans, so you can stick a reference to that property into the `condition` statement by itself. Otherwise, the `condition` statement consists of two values separated by a comparison operator, such as == (equals) or `!=` (does not equal).

Next, look at some real JavaScript. The following function receives a text input object:

```
function notTooHigh(oInput)
{
      if (parseInt(oInput.value) > 100)
      {
         alert("Sorry, the value you entered is too high. Try again.");
         return false;
      }
   return true;
}
```

The condition (in parentheses) tests the contents of the field against a hard-wired value of `100`. If the entered value is larger than that, the function alerts you and returns a `false` value to the calling statement elsewhere in the script. But if the value is less than `100`, all intervening code is skipped and the function returns `true`.

About (*condition*) expressions

A lot of condition testing for control structures compares a value against some very specific condition, such as a string being empty or a value being `null`. You can use a couple of shortcuts to take care of many circumstances. Table 21-1 details the values that evaluate to `true` or `false` (or equivalent) to satisfy a control structure's *condition* expression.

TABLE 21-1

Condition Value Equivalents

True	False
Nonempty string	Empty string
Nonzero number	0
Nonnull value	`null`
Object exists	Object doesn't exist
Property is defined	Undefined property

Instead of having to spell out an equivalency expression for a condition involving these kinds of values, you can simply supply the value to be tested. For example, if a variable named `myVal` is capable of reaching an `if` construction with a value of `null`, an empty string, or a string value for further processing, you can use the following shortcut:

```
if (myVal)
{
   // do processing on myVal
}
```

All `null` or empty string conditions evaluate to `false`, so that only the cases of `myVal` being a processable value get inside the `if` construction. This mechanism is the same that you have seen elsewhere in this book to employ object detection for browser branching. For example, the code nested inside the following code segment executes only if the `document` object has an `images` array property:

```
if (document.images)
{
   // do processing on image objects
}
```

Complex decisions

The simple type of `if` construction described earlier is fine when the decision is to take a small detour before returning to the main path. But not all decisions — in programming or in life — are like that. To present two alternate paths in a JavaScript decision, you can add a component to the construction. The syntax is

```
if (condition)
{
   // statementsIfTrue
}
else
{
   // statementsIfFalse
}
```

By appending the else keyword, you give the if construction a path to follow in case the condition evaluates to false. The statementsIfTrue and statementsIfFalse do not have to be balanced in any way: One statement can be one line of code, the other 100 lines. But when either one of those branches completes, execution continues after the last closing brace. To demonstrate how this construction can come in handy, the following example is a script fragment that assigns the number of days in February based on whether or not the year (the Date object is explained in Chapter 17) is a leap year (using modulo arithmetic, explained in Chapter 22, to determine if the year is evenly divisible by four, and setting aside all other leap year calculation details for the moment):

```
var howMany = 0;
var currentTime = new Date();
var theYear = currentTime.getFullYear();
   if (theYear % 4 == 0)
   {
      howMany = 29;
   }
   else
   {
      howMany = 28;
   }
```

Here is a case where execution has to follow only one of two possible paths to assign the number of days to the howMany variable. Had the else portion not been used, as in

```
var howMany = 0;
var currentTime = new Date();
var theYear = currentTime.getFullYear();
   if (theYear % 4 == 0)
   {
       howMany = 29;
   }
howMany = 28;
```

then the variable would always be set to 28, occasionally after momentarily being set to 29. The else construction is essential in this case. However, by initializing your variable to 28, it is possible to achieve the correct end without the else portion, as in

```
var howMany = 28;
var currentTime = new Date();
var theYear = currentTime.getFullYear();
   if (theYear % 4 == 0)
   {
       howMany = 29;
   }
```

Nesting if. . .else statements

Designing a complex decision process requires painstaking attention to the logic of the decisions your script must process and the statements that must execute for any given set of conditions. The need for repetitive logic disappeared with the advent of switch construction in version 4 browsers (described later in this chapter), but there may still be times when you must fashion complex decision behavior

out of a series of nested `if. . .else` statements. Without a JavaScript-aware text editor to help keep everything properly indented and properly terminated (with closing braces), you have to monitor the authoring process very carefully. Moreover, the error messages that JavaScript provides when a mistake occurs (see Chapter 48, “Debugging Scripts,” on the CD-ROM) may not point directly to the problem line, but only to the region of difficulty.

To demonstrate a nested set of `if. . .else` constructions, Listing 21-1 presents a simple user interface to a complex problem. A single text object asks the user to enter one of three letters — A, B, or C. The script behind that field processes a different message for each of the following conditions:

- The user enters no value.
- The user enters A.
- The user enters B.
- The user enters C.
- The user enters something entirely different.

Note

The function to assign event handlers throughout the code in this chapter and much of the book is `addEvent()`, a cross-browser event handler explained in detail in Chapter 32: See “A cross-browser event binding solution.” ■

What’s with the Formatting?

Indentation of the `if` construction, and the further indentation of the statements executed on a `true` condition, are not required by JavaScript. What you see here, however, is a convention that most JavaScript scripters follow. As you write the code in your text editor, you can use the Tab key to establish each level of indentation; some developers prefer using a setting in their editor that converts tabs to spaces, which guarantees that indentations are consistent across different editors. The browser ignores these tab characters (and/or spaces) when loading your scripts.

There are many styles of indenting. The style we use most often in this book is known as Allman — in which opening and closing braces occupy their own lines — while many programmers prefer 1TBS (The One True Brace Style), in which the opening brace is on the same line as the control statement.

LISTING 21-1

Nested if. . .else Constructions

HTML: jsb-21-01.html

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Nested if...else</title>
      <script type="text/javascript" src="../jsb-global.js"></script>
```

continued

LISTING 21-1 *(continued)*

```
      <script type="text/javascript" src="jsb-21-01.js"></script>
   </head>
   <body>
      <h1>Nested if...else</h1>
      <form id="theForm" action="validate.php" method="get">
         <fieldset>
            <label for="theText">Please enter A, B, or C: </label>
            <input id="theText" type="text">
            <input type="submit">
         </fieldset>
      </form>
   </body>
</html>
```

JavaScript: jsb-21-01.js

```
// initialize when the page has loaded
addEvent(window, 'load', initialize);

function initialize()
{
   // add an event to the form
   addEvent(document.getElementById('theForm'), 'submit',
     suppressFormSubmission);

   // add an event to the input box
   addEvent(document.getElementById('theText'), 'change', testLetter);
}

function suppressFormSubmission(evt)
{
      // consolidate event handling
      if (!evt) evt = window.event;

   // cancel default behavior
      // W3C DOM method (hide from IE)
      if (evt.preventDefault) evt.preventDefault();

   // IE method
   return false;
}

function testLetter()
{
   // assign to shorter variable name
   var inpVal = this.value;

      if (inpVal != "")
      {  // if entry is not empty then dive in...
```

```
      if (inpVal == "A")
      {  // Is it an "A"?
         alert("Thanks for the A.");
      }
      else if (inpVal == "B")
      {  // No.  Is it a "B"?
         alert("Thanks for the B.");
      }
      else if (inpVal == "C")
      {  // No.  Is it a "C"?
         alert("Thanks for the C.");
      }
      else
      { // Nope.  None of the above
         alert("Sorry, wrong character or case.");
      }
   }
   else
   {   // value was empty, so skipped all other stuff above
      alert("You did not enter anything.");
   }
}
```

Each condition executes only the statements that apply to that particular condition, even if it takes several queries to find out what the entry is. You do not need to break out of the nested construction, because when a `true` response is found, the relevant statement executes, and no other statements occur in the execution path to run.

You might have noticed the `action` on the `form` element identifies a PHP script called `validate.php`. In Chapter 7, "Scripts and HTML Documents," we talked about the advisability of always specifying a server-side program in case the user has turned off JavaScript.

Conditional Expressions

Compatibility: WinIE3+, MacIE3+, NN2+, Moz+, Safari+, Opera+, Chrome+

While we're showing you decision-making constructions in JavaScript, now is a good time to introduce a special type of expression that you can use in place of an `if...else` control structure for a common type of decision — the instance where you want to assign one of two values to a variable, depending on the outcome of some condition. The formal definition for the conditional expression is as follows:

```
variable = (condition) ? val1 : val2;
```

This expression means that if the Boolean result of the `condition` statement is `true`, JavaScript assigns `val1` to the variable; otherwise, it assigns `val2` to the variable. Like other instances of conditional expressions, this one must also be written inside parentheses. The question mark is key here, as is the colon separating the two possible values.

A conditional expression, though not particularly intuitive or easy to read inside code, is very compact. Compare an `if...else` version of an assignment decision that follows

```
var collectorStatus;
   if (CDCount > 500)
   {
      collectorStatus = "fanatic";
   }
   else
   {
      collectorStatus = "normal";
   }
```

with the conditional expression version:

```
var collectorStatus = (CDCount > 500) ? "fanatic" : "normal";
```

The latter saves a few code lines, although the internal processing is the same as that of an `if...else` construction. Of course, if your decision path contains more statements than just one setting the value of a variable, the `if...else` or `switch` construction is preferable. This shortcut, however, is a handy one to remember if you need to perform binary actions, such as setting a true-or-false flag in a script.

The switch Statement

Compatibility: WinIE4+, MacIE4+, NN4+, Moz+, Safari+, Opera+, Chrome+

In some circumstances, a binary — true or false — decision path is not enough to handle the processing in your script. An `object` property or `variable` value may contain any one of several values, and a separate execution path is required for each one. The most obvious way to establish such a decision path is with a series of `if...else` constructions. However, in addition to quickly getting unwieldy with nested code, the more conditions you must test, the less efficient the processing is, because each condition must be tested. The end result is a sequence of clauses and braces that can get very confusing.

Starting in version 4 browsers, a control structure in use by many languages was introduced to JavaScript. The implementation is similar to that of PHP, Java, and C/C++, which use the `switch` and `case` keywords. The basic premise is that you can create any number of execution paths based on the value of some expression. At the beginning of the structure, you identify what that expression is and then, for each execution path, assign a label matching a particular value.

The formal syntax for the `switch` statement is as follows:

```
switch (expression)
{
   case label1:
      [statements]
      [break;]
   case label2:
      [statements]
      [break;]
   ...
```

```
    [default:
       statements]
}
```

For example:

```
switch (TimeOfDay)
{
   case 'morning':
      meal = 'breakfast';
      break;

   case 'midday':
      meal = 'lunch';
      break;

   case 'afternoon':
   case 'evening':
      meal = 'dinner';
      break;

   default:
      meal = 'fast';
}
```

The `expression` parameter of the `switch` statement can evaluate to any string or number value. Labels are surrounded by quotes when the labels represent string values of the expression. Notice that the `break` statements are optional. A `break` statement forces the switch expression to bypass all other checks of succeeding labels against the expression value. It's important to understand that without a `break` statement at the end of each `case`, every line of code in the `switch` expression will get executed. However, since Boolean operators are not allowed (each label must be for one and only one value), you can use careful placement of the `break` statement. (In the preceding example, if the time of day is either afternoon or evening, the meal is dinner.) Another option is the `default` statement, which provides a catchall execution path when the expression value does not match any of the `case` statement labels. If you'd rather not have any execution take place with a non-matching expression value, omit the default part of the construction.

To demonstrate the syntax of a working `switch` statement, Listing 21-2 provides the skeleton of a larger application of this control structure. The page contains two separate arrays of different product categories. Each product has its name and price stored in its respective array. A `select` list displays the product names. After a user chooses a product, the script looks up the product name in the appropriate array and displays the price.

The trick behind this application is the values assigned to each product in the select list. While the displayed text is the product name, the `value` attribute of each `<option>` tag is the array category for the product. That value is the expression used to decide which branch to follow. Notice, too, that we assign a label to the entire switch construction. The purpose of that is to let the deeply nested repeat loops for each case completely bail out of the switch construction (via a labeled `break` statement) whenever a match is made. You can extend this example with any number of product category arrays with additional `case` statements to match.

LISTING 21-2

The switch Construction in Action

HTML: jsb-21-02.html

```
<!DOCTYPE html>
<html>
  <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Switch Statement and Labeled Break</title>
      <script type="text/javascript" src="../jsb-global.js"></script>
      <script type="text/javascript" src="jsb-21-02.js"></script>
   </head>
   <body>
      <h1>Switch Statement and Labeled Break</h1>
      <p>Select a chip for lookup in the chip price table:</p>
      <form id="theForm" action="validate.php" method="get">
         <p>
            <label for="chips">Chip:</label>
            <select id="chips">
               <option></option>
               <option value="ICs">Septium 900MHz</option>
               <option value="ICs">Septium Pro 1.0GHz</option>
               <option value="ICs">Octium BFD 750MHz</option>
               <option value="snacks">Rays Potato Chips</option>
               <option value="snacks">Cheezey-ettes</option>
               <option value="snacks">Tortilla Flats</option>
               <option>Poker Chipset</option>
            </select>
            <label for="cost">  Price:</label>
            <input type="text" id="cost" size="10">
         </p>
      </form>
   </body>
</html>
```

JavaScript: jsb-21-02.js

```
// initialize when the page has loaded
addEvent(window, 'load', initialize);

function initialize()
{
   // add an event to the drop down list
   addEvent(document.getElementById('chips'), 'change', getPrice);
}

// build two product arrays with a custom object, simulating two database tables
```

```
function product(name, price)
{
   this.name = name;
   this.price = price;
}

var ICs = new Array();
ICs[0] = new product("Septium 900MHz", "$149");
ICs[1] = new product("Septium Pro 1.0GHz", "$249");
ICs[2] = new product("Octium BFD 750MHz", "$329");

var snacks = new Array();
snacks[0] = new product("Rays Potato Chips", "$1.79");
snacks[1] = new product("Cheezey-ettes", "$1.59");
snacks[2] = new product("Tortilla Flats", "$2.29");

// lookup in the 'table' the cost associated with the product
function getPrice()
{
   var chipName = this.options[this.selectedIndex].text;
   var chipType = this.options[this.selectedIndex].value;
   var outField = document.getElementById('cost');

   master:
   switch(chipType)
   {
      case "ICs":
         for (var i = 0; i < ICs.length; i++)
         {
            if (ICs[i].name == chipName)
            {
               outField.value = ICs[i].price;
               break master;
            }
         }
         break;
      case "snacks":
         for (var i = 0; i < snacks.length; i++)
         {
            if (snacks[i].name == chipName)
            {
               outField.value = snacks[i].price;
               break master;
            }
         }
         break;
      default:
         outField.value = "Not Found";
   }
}
```

Repeat (for) Loops

Compatibility: WinIE3+, MacIE3+, NN2+, Moz+, Safari+, Opera+, Chrome+

As you have seen in numerous examples throughout other chapters, the capability to cycle through every entry in an array or through every item of a form element is vital to many JavaScript scripts. Perhaps the most typical operation is inspecting a property of many similar items in search of a specific value, such as to determine which radio button in a group is selected. One JavaScript structure that allows for these repetitious excursions is the `for` loop, so-named after the keyword that begins the structure. Two other structures, the `while` loop and the `do-while` loop, are covered in the following sections.

The JavaScript `for` loop repeats a series of statements any number of times and includes an optional loop counter that can be used in the execution of the statements. The following is the formal syntax definition:

```
for ( [initial expression]; [condition]; [update expression])
{
   statements
}
```

The three statements inside the parentheses (parameters to the `for` statement) play a key role in the way a `for` loop executes.

An initial expression in a `for` loop is executed one time, the first time the `for` loop begins to run. The most common application of the initial expression is to assign a name and starting value to a loop counter variable. Thus, seeing a `var` statement that both declares a variable name and assigns an initial value (generally `0` or `1`) to it is not uncommon. An example is

```
var i = 0;
```

You can use any variable name, but conventional usage calls for the letter `i`, which is short for *index*. If you prefer the word `counter` or another word that reminds you of what the variable represents, that's fine, too. In any case, the important point to remember about this statement is that it executes once at the outset of the `for` loop.

The second statement is a *condition*, precisely like the `condition` statement you saw in `if` constructions earlier in this chapter. When a loop-counting variable is established in the initial expression, the `condition` statement usually defines how high the loop counter should go before the looping stops. Therefore, the most common statement here is one that compares the loop counter variable against some fixed value — is the loop counter less than the maximum allowed value? If the condition is false at the start, the body of the loop is not executed. But if the loop does execute, then when execution comes back around to the top of the loop, JavaScript reevaluates the condition to determine the current result of the expression. If the loop counter increases with each loop, eventually the counter value goes beyond the value in the `condition` statement, causing the `condition` statement to yield a Boolean value of `false`. The instant that happens, execution drops out of the `for` loop entirely.

The final statement, the *update expression*, is executed at the end of each loop execution — after all statements nested inside the `for` construction have run. Again, the loop counter variable can be a factor here. If you want the counter value to increase by one the next time through the loop (called incrementing the value), you can use the JavaScript operator that makes that happen: the ++ operator appended to the variable name. That task is the reason for the appearance of all those `i++` symbols in the `for` loops that you've already seen in this book. You're not limited to incrementing by one. You

can increment by any multiplier you want, or even drive a loop counter backward by decrementing the value (i--).

Now, take this knowledge and beef up the formal syntax definition with one that takes into account a typical loop-counting variable, i, and the common ways to use it:

```
//incrementing loop counter
for (var i = minValue; i <= maxValue; i++)
{
   // statements
}
//decrementing loop counter
for (var i = maxValue; i >= minValue; i--)
{
   // statements
}
```

In the top loop counter, the variable, i, is initialized at the outset to a value equal to that of minValue. Variable i is immediately compared against maxValue. If i is less than or equal to maxValue, processing continues into the body of the loop. At the end of the loop, the update expression executes. The value of i is incremented by 1. Therefore, if i is initialized as 0, then the first time through the loop, the i variable maintains that 0 value during the first execution of statements in the loop. The next time around, the variable has the value of 1.

As you may have noticed in the formal syntax definition, each of the parameters to the for statement is optional. For example, the statements that execute inside the loop may control the value of the loop counter based on data that gets manipulated in the process. Therefore, the update statement would probably interfere with the intended running of the loop. But we suggest that you use all three parameters until such time as you feel absolutely comfortable with their roles in the for loop. If you omit the condition statement, for instance, and you don't program a way for the loop to exit on its own, your script may end up in an infinite loop — which does your users no good.

Putting the loop counter to work

Despite its diminutive appearance, the i loop counter (or whatever name you want to give it) can be a powerful tool for working with data inside a repeat loop. For example, examine a version of the classic JavaScript function that creates an array while initializing entries to a value of 0:

```
// initialize array with n entries
function MakeArray(n)
{
   this.length = n;
   for (var i = 0; i < n; i++)
   {
      this[i] = 0;
   }
   return this;
}
```

The loop counter, i, is initialized to a value of 0 because the first element of any JavaScript array is element zero. In the condition statement, the loop continues to execute as long as the value of the counter is less than the number of entries being created (n). (For example, if n is 10, the array

is initialized with ten elements, 0–9.) After each loop, the counter increments by 1. In the nested statement that executes within the loop, you use the value of the i variable to substitute for the index value of the assignment statement:

```
this[i] = 0;
```

The first time the loop executes, the value expression evaluates to

```
this[0] = 0;
```

The next time, the expression evaluates to

```
this[1] = 0;
```

and so on, until all n entries are created and stuffed with 0.

Recall the HTML page in Listing 18-2, where a user chooses a regional office from a select list (triggering a script to look up the manager's name and sales quota for that region). Because the regional office names are stored in an array, the page could be altered so that a script populates the select element's options from the array. That way, if there is ever a change to the alignment of regional offices, there need be only one change to the array of offices, and the HTML doesn't have to be modified. As a reminder, here is the definition of the regional offices array, created while the page loads:

```
var aRegionalOffices = ["New York", "Chicago", "Houston", "Portland"];
```

Code inside the external JavaScript file can be used to dynamically generate the select list as follows:

```
for (var i = 0; i < aRegionalOffices.length; i++)
{
   oSelect.options[i] = new Option(aRegionalOffices[i]);
}

// pre-select the first option
oSelect.options[0].selected = true;
```

Notice one important point about the condition statement of the for loop: JavaScript extracts the length property from the array to be used as the loop counter boundary. From a code maintenance and stylistic point of view, this method is preferable to hard-wiring a value there. If the company added a new regional office, you would make the addition to the array "database," whereas everything else in the code would adjust automatically to those changes, including creating a longer pop-up menu in this case.

Notice, too, that the operator for the condition statement is less-than (<): The zero-based index values of arrays means that the maximum index value we can use is one less than the actual count of items in the array. This is vital information, because the index counter variable (i) is used as the index to the regionalOffices array each time through the loop to read the string for each item's entry. You also use the index of the first option (0) to add the selected attribute to the first option's definition.

The utility of the loop counter in for loops often influences the way you design data structures, such as two-dimensional arrays (see Chapter 18, "The Array Object") for use as databases. Always keep "the loop-counter mechanism" in the back of your mind when you begin writing JavaScript script that relies on collections of data that you embed in your documents.

Breaking out of a loop

Some loop constructions perform their job as soon as a certain condition is met, at which point they have no further need to continue looping through the rest of the values in the loop counter's range. A common scenario for this is the cycling of a loop through an entire array in search of a single entry that matches some criterion. That criterion test is set up as an `if` construction inside the loop. If that criterion is met, you break out of the loop and let the script continue with the more meaningful processing of succeeding statements in the main flow. To accomplish that exit from the loop, use the `break` statement. The following schematic shows how the `break` statement may appear in a `for` loop:

```
for (var i = 0; i < array.length; i++)
{
   if (array[i].property == magicValue)
   {
      // statements that act on entry array[i]
      break;
   }
}
```

The `break` statement tells JavaScript to bail out of the nearest `for` loop (in case you have nested `for` loops). Script execution then picks up immediately after the closing brace of the `for` statement. The variable value of `i` remains whatever it was at the time of the break, so that you can use that variable later in the same script to access, say, that same array entry.

A construction similar to this was used in Chapter 35, "Button Objects." There, the discussion of radio buttons demonstrates this construction, where, in Listing 35-6, you see a set of radio buttons whose `value` attributes contain screen sizes, in pixels. A function uses a `for` loop to find out which button was selected and then uses that item's index value — after the `for` loop breaks out of the loop — to alert the user. Listing 21-3 shows the relevant function.

LISTING 21-3

Breaking Out of a for Loop

JavaScript: jsb-21-03.js

```
function showMegapixels()
{
   var theForm = document.getElementById('sizesForm');
      if (theForm)
      {
         var sizeChoices = theForm.choices;
         for (var i = 0; i < sizeChoices.length; i++)
         {
            if (sizeChoices[i].checked)
            {
               break;
            }
         }
```

continued

LISTING 21-3 *(continued)*

```
         alert("That image size requires " + sizeChoices[i].value +
            " megapixels.");
      }
}
```

In this case, breaking out of the `for` loop is more than a matter of mere efficiency; the value of the loop counter (frozen at the break point) is used to summon a different property outside of the `for` loop. Starting back in version 4 browsers, the `break` statement gained additional powers in cooperation with the new `label` feature of control structures. This subject is covered later in this chapter.

Directing loop traffic with continue

One other possibility in a `for` loop is that you may want to skip execution of the nested statements for just one condition. In other words, as the loop goes merrily on its way round and round, executing statements for each value of the loop counter, one value of that loop counter may exist for which you don't want those statements to execute. To accomplish this task, the nested statements need to include an `if` construction to test for the presence of the value to skip. When that value is reached, the `continue` command tells JavaScript to immediately skip the rest of the body, execute the `update` statement, and loop back around to the top of the loop.

To illustrate this construction, take a look at an artificial example that skips over execution when the counter variable is the superstitious person's unlucky 13:

```
for (var i = 0; i <= 20; i++)
{
   if (i == 13)
   {
      continue;
   }
   // statements
}
```

In this example, the `statements` part of the loop executes for all values of `i` except `13`. The `continue` statement forces execution to jump to the `i++` part of the loop structure, incrementing the value of `i` for the next time through the loop. In the case of nested `for` loops, a `continue` statement affects the `for` loop in whose immediate scope the `if` construction falls. The `continue` statement was enhanced in version 4 browsers with the `label` feature of control structures, which is covered later in this chapter.

The while Loop

Compatibility: WinIE3+, MacIE3+, NN2+, Moz+, Safari+, Opera+, Chrome+

The `for` loop is not the only kind of repeat loop you can construct in JavaScript. Another statement, called a `while` statement, sets up a loop in a slightly different format. Rather than providing a mechanism for modifying a loop counter, a `while` repeat loop assumes that your script statements will reach a condition that forcibly exits the repeat loop.

The basic syntax for a `while` loop is

```
while (condition)
{
   statements
}
```

The `condition` expression is the same kind that you saw in `if` constructions and in the middle parameter of the `for` loop. You introduce this kind of loop if some condition exists in your code (evaluates to `true`) before reaching this loop. The loop then performs some action, which affects that condition repeatedly until that condition becomes `false`. At that point, the loop exits, and script execution continues with statements after the closing brace. If the statements inside the `while` loop do not somehow affect the values being tested in `condition`, your script never exits, and it becomes stuck in an infinite loop.

Many loops can be rendered with either the `for` or `while` loops. In fact, Listing 21-4 shows a `while` loop version of the `for` loop from Listing 21-3.

LISTING 21-4

A while Loop Version of Listing 21-3

JavaScript: jsb-21-04.js

```
function showMegapixels()
{
   var theForm = document.getElementById('sizesForm');
      if (theForm)
      {
         var sizeChoices = theForm.choices;
         var i = 0;
         while (i < sizeChoices.length && !sizeChoices[i].checked)
            {
               i++;
            }
         }
         alert("That image size requires " + sizeChoices[i].value +
            " megapixels.");
      }
}
```

One point you may notice is that if the condition of a `while` loop depends on the value of a loop counter, the scripter is responsible for initializing the counter prior to the `while` loop construction and managing its value within the `while` loop.

Should you need their powers, the `break` and `continue` control statements work inside `while` loops just as they do in `for` loops. But because the two loop styles treat their loop counters and conditions differently, be extra careful (do lots of testing) when applying `break` and `continue` statements to both kinds of loops.

No hard-and-fast rules exist for which type of loop construction to use in a script. Generally, use `while` loops only when the data, or object, to loop through is already part of the script before the loop. In other words, by virtue of previous statements in the script, the values for any condition or loop counting (if needed) are already initialized. But if cycling through an object's properties, or an array's entries, to extract some piece of data for use later in the script is what's needed, favor the `for` loop. The `for` loop is also generally preferred when the looping involves a simple counter from one value to another.

Another point of style, particularly with the `for` loop, is where a scripter should declare the `i` variable. Some programmers prefer to declare (or initialize, if initial values are known) all variables in the opening statements of a script or function. That is why you tend to see a lot of `var` statements in those positions in scripts. If you have only one `for` loop in a function, for example, nothing is wrong with declaring and initializing the `i` loop counter in the initial expression part of the `for` loop (as demonstrated frequently in the previous sections). But if your function utilizes multiple `for` loops that reuse the `i` counter variable (that is, the loops run completely independently of one another), then you can declare the `i` variable once at the start of the function and simply assign a new initial value to `i` in each `for` construction.

The do-while Loop

Compatibility: WinIE4+, MacIE4+, NN4+, Moz+, Safari+, Opera+, Chrome+

JavaScript brings you one more looping construction, called the `do-while` loop. The formal syntax for this construction is as follows:

```
do
{
   statements
} while (condition)
```

An important difference distinguishes the `do-while` loop from the `while` loop. In the `do-while` loop, the statements in the construction always execute at least one time before the condition can be tested; in a `while` loop, the statements may never execute if the condition tested at the outset evaluates to `false`. So, just think of the `do-while` loop as a `while` loop where the statements get executed at least once, no matter what.

Use a `do-while` loop when you know for certain that the looped statements are free to run at least one time. If the condition may not be met the first time, use the `while` loop. For many instances, the two constructions are interchangeable, although only the `while` loop is compatible with legacy browsers.

Looping through Properties (for-in)

Compatibility: WinIE3+, MacIE3+, NN2+, Moz+, Safari+, Opera+, Chrome+

JavaScript includes a variation of the `for` loop, called a `for-in` loop, which has special powers of extracting the names and values of any object property currently in the browser's memory. The syntax looks like this:

```
for (varName in object)
{
```

```
        // statements
    }
```

The `object` parameter is not the string name of an object but a reference to the object itself. JavaScript delivers an object reference if you provide the name of the object as an unquoted string, such as `window` or `document`, or via `getElementById`. Recall that although a legal name in HTML is case-insensitive, a legal name in JavaScript is case-sensitive, so remember to use the camelCase convention even in your HTML code. Using the `varName` variable, you can create a script that extracts and displays the range of properties for any given object.

Listings 21-5 shows a page containing a utility function that you can insert into your external JavaScript files during the authoring and debugging stages of designing a JavaScript-enhanced page. In the example, the current `window` object is examined, and its properties are presented in the page (note that Safari 1.0 doesn't expose `window` object properties).

LISTING 21-5

Property Inspector Function

HTML: jsb-21-05.html

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Property Inspector Function</title>
      <script type="text/javascript" src="../jsb-global.js"></script>
      <script type="text/javascript" src="jsb-21-05.js"></script>
      <link rel="stylesheet" href="jsb-21-05.css" type="text/css">
   </head>
   <body>
      <h1>Property Inspector Function</h1>
      <p>Here are the properties of the current window:</p>
      <ul id="propertyList"></ul>
   </body>
</html>
```

JavaScript: jsb-21-05.js

```
// initialize when the page has loaded
addEvent(window, 'load', showProps);

// display all the properties of the object
function showProps()
{
   var oOutput = document.getElementById('propertyList');
      if (oOutput)
      {
         var newElem;
         var newText;
         for (var i in window)
```

continued

LISTING 21-5 *(continued)*

```
        {
          newElem = document.createElement('li');
          newElem.className = 'objProperty';
          newText = document.createTextNode('window.' + i + ' = ' + window[i]);

          // insert the text node into the new paragraph
          newElem.appendChild(newText);

          // insert the completed paragraph into propertyList
          oOutput.appendChild(newElem);
        }
      }
}
```

Stylesheet: jsb-21-05.css

```
ul#propertyList li
{
   white-space: pre;
}
```

For debugging purposes, you can revise the function slightly to display the results in an alert dialog box or a `textarea` element. Just use the `\n` carriage return character after each property for a nicely formatted display. If the `showProps()` function looks familiar to you, it is because it closely resembles the property inspector routines of The Evaluator (see Chapter 4). In Chapter 48 (on the CD-ROM), you can see how to embed functionality of The Evaluator into a page under construction so that you can view property values while debugging your scripts.

The with Statement

Compatibility: WinIE3+, MacIE3+, NN2+, Moz+, Safari+, Opera+, Chrome+

The `with` statement enables you to preface any number of statements by advising JavaScript on precisely which object your scripts will be talking about, so that you don't have to use full, formal addresses to access properties or invoke methods of the same object. The formal syntax definition of the `with` statement is as follows:

```
with (object)
{
   statements
}
```

The object reference is a reference to any valid object currently in the browser's memory. An example of this appears in Chapter 16, "The Math, Number, and Boolean Objects," when the `Math` object is discussed. By embracing several `Math` statements inside a `with` construction, your scripts can call the

properties and methods without having to make the object part of every reference to those properties and methods. Here's an example:

```
with (Math)
{
   var randInt = round(random() * 100);
}
```

This example uses the `round()` and `random()` methods of the `Math` object to obtain a random integer between 0 and 100. The significance of the code is how the `with` statement allows you to forego using the full notation of `Math.round()` and `Math.random()`.

An advantage of the `with` structure is that it can make heavily object-dependent statements easier to read and understand. Consider this long version of a function that requires multiple calls to the same object (but different properties):

```
function seeColor(form)
{
   var newColor = (form.colorsList.options[form.colorsList.selectedIndex]
   .text);
   return newColor;
}
```

Using the `with` structure, you can shorten the long statement:

```
function seeColor(form)
{
   with (form.colorsList)
   {
      newColor = (options[selectedIndex].text);
   }
   return newColor;
}
```

When JavaScript encounters an otherwise unknown identifier inside a `with` statement, it tries to build a reference out of the object specified as its parameter and that unknown identifier. You cannot, however, nest `with` statements that build on one another. For instance, in the preceding example, you cannot have a `with (colorsList)` nested inside a `with (form)` statement and expect JavaScript to create a reference to options out of the two object names.

As clever as the `with` statement may seem, be aware that there are some cautions associated with its use. It introduces some inherent performance penalties in your script (because of the way the JavaScript interpreter must artificially generate references). You probably won't notice degradation with occasional use of this construction, but if it's used inside a loop that must iterate many times, processing speed will almost certainly be affected negatively. Additionally, some people feel that the `with` statement introduces ambiguity in your code.

Labeled Statements

Compatibility: WinIE4+, MacIE4+, NN4+, Moz+, Safari+, Opera+, Chrome+

Crafting multiple nested loops can sometimes be difficult when the final condition your script is looking for is met deep inside the nests. The problem is that the `break` or `continue` statement by itself

has scope only to the nearest loop level. Therefore, even if you break out of the inner loop, the outer loop(s) continue to execute. If all you want to do is exit the function after the condition is met, a simple `return` statement performs the same job as some other languages' exit command. But if you also need some further processing within that function after the condition is met, you need the JavaScript facility supported in modern browsers that lets you assign labels to blocks of statements. Your `break` and `continue` statements can then alter their scope to apply to a labeled block other than the one containing the statement.

A *label* is any identifier (that is, a name starting with a letter and containing no spaces or odd punctuation other than underscores) followed by a colon preceding a logical block of executing statements, such as an `if...then` or loop construction. The formal syntax looks like the following:

```
labelID:
   statements
```

For a `break` or `continue` statement to apply itself to a labeled group, the label is added as a kind of parameter to each statement, as in

```
break labelID;
continue labelID;
```

Note

If you're arriving at JavaScript from another programming language that has a bit more structure, such as C++ or Java, the thought of labeled code may have you worried about the risks of creating code that is impossible to manage. This worry is likely rooted in the `goto` statement that is found in some languages, such as BASIC, and which is seriously frowned upon in modern structured languages. Labels in JavaScript are much more limited than the infamous `goto` statement in other languages; you can only use the `break` statement to target a label in which your code is nested. So, although we don't necessarily encourage the heavy usage of labels, you can rest easy knowing that they aren't on par with the much maligned `goto` statement. ■

To demonstrate how valuable labels can be in the right situation, Listing 21-6 contains two versions of the same nested loop construction. The goal of each version is to loop through two different index variables until both values equal the target values set outside the loop. When those targets are met, the entire nested loop construction should break off and continue processing afterward. To help you visualize the processing that goes on during the execution of the loops, the scripts output intermediate and final results to a `textarea`.

In the version without labels, only the simple `break` statement is issued when the targets are met. This breaks the inner loop at that point, but the outer loop picks up on the next iteration. By the time the entire construction has ended, a lot of wasted processing has gone on. Moreover, the values of the counting variables max themselves out, because the loops execute in their entirety several times after the targets are met.

But in the labeled version, the inner loop breaks out of the labeled outer loop as soon as the targets are met. Far fewer lines of code are executed, and the loop counting variables are equal to the targets, as desired. Experiment with Listing 21-6 by changing the `break` statements to `continue` statements. Then closely analyze the two results in the Results `textarea` to see how the two versions behave.

LISTING 21-6

Labeled Statements

HTML: jsb-21-06.html

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Breaking Out of Nested Labeled Loops</title>
      <script type="text/javascript" src="../jsb-global.js"></script>
      <script type="text/javascript" src="jsb-21-06.js"></script>
   </head>
   <body>
      <h1>Breaking Out of Nested Labeled Loops</h1>
      <p>Look in the Results field for traces of these button scripts:</p>
      <form id="theForm" action="validate.php" method="get">
        <p><input id="withoutLabelButton" type="button" value="Execute WITHOUT↩
              Label"></p>
        <p><input id="withLabelButton" type="button" value="Execute WITH↩
              Label"></p>
        <p>
           <label for="output">Results:</label>
           <textarea id="output" rows="43" cols="60"></textarea>
        </p>
      </form>
   </body>
</html>
```

JavaScript: jsb-21-06.js

```
// initialize when the page has loaded
addEvent(window, 'load', initialize);

function initialize()
{
   // locate output field
   out = document.getElementById('output');
      // if found...
      if (out)
      {
         // add an event to the button button labeled Execute WITHOUT Label
         addEvent(document.getElementById('withoutLabelButton'), 'click', run1);

         // add an event to the button button labeled Execute WITH Label
         addEvent(document.getElementById('withLabelButton'), 'click', run2);
      }
}
```

continued

LISTING 21-6 *(continued)*

```
var out;   // global variable for output field
var targetA = 2;
var targetB = 2;
var range = 5;

function run1()
{
   out.value = "Running WITHOUT labeled break\n";
   for (var i = 0; i <= range; i++)
   {
      out.value += "Outer loop #" + i + "\n";
      for (var j = 0; j <= range; j++)
      {
         out.value += "  Inner loop #" + j + "\n";
         if (i == targetA && j == targetB)
         {
            out.value += "**BREAKING OUT OF INNER LOOP**\n";
            break;
         }
      }
   }
   out.value += "After looping, i = " + i + ", j = " + j + "\n";
}

function run2()
{
   out.value = "Running WITH labeled break\n";
   outerLoop:
   for (var i = 0; i <= range; i++)
   {
      out.value += "Outer loop #" + i + "\n";
      innerLoop:
      for (var j = 0; j <= range; j++)
      {
         out.value += "  Inner loop #" + j + "\n";
         if (i == targetA && j == targetB)
         {
            out.value += "**BREAKING OUT OF OUTER LOOP**\n";
            break outerLoop;
         }
      }
   }
   out.value += "After looping, i = " + i + ", j = " + j + "\n";
}
```

Exception Handling

The subject of exception handling is relatively new to JavaScript. Formalized in Edition 3 of ECMA-262, parts of the official mechanism were implemented in IE5, with more complete implementations in IE6 and NN6, and, of course, in Mozilla, Firefox, Camino, Safari, Opera, and Chrome.

Exceptions and errors

If you've done any scripting, you are certainly aware of JavaScript errors, whether they be from syntax errors in your code, or what are known as *runtime* errors — errors that occur while scripts are processing information. Ideally, a program should be aware of when an error occurs and handle it as gracefully as possible. This self-healing can prevent lost data and help keep users from seeing ugly error messages. Chapter 27, "Window and Frame Objects," covers the `onerror` event handler and `window.onerror` property, which were early attempts at letting scripts gain a level of control over runtime errors. This event-driven mechanism works on a global level (that is, in the `window` object) and processes every error that occurs throughout the page. This event handler ends up being used primarily as a last-ditch defense against displaying any error message to the user, and is a long way from what programmers consider to be proper exception handling.

In the English language, the term "exception" can mean something out of the ordinary, or something abnormal. This definition seems quite distant from the word "error," which usually means a mistake. In the realm of programming languages, however, the two words tend to be used interchangeably, and the difference between the two depends primarily on one's point of view.

Consider, for example, a simple script whose job is to multiply numbers that the user enters into two text fields on the page. The script is supposed to display the results in a third text box. If the script contains no data entry validation, JavaScript will attempt to multiply whatever values are entered into the text boxes. If the user enters two numbers, JavaScript is smart enough to recognize that even though the `value` properties of the two input text fields are strings, the strings contain numbers that can be converted to number types for the proper multiplication. Without complaint, the product of the two numbers gets calculated and displayed into the results.

But what if the user types a letter into one of the text boxes? Again, without any entry validation in the script, JavaScript has a fixed way of responding to such a request: The result of the multiplication operation is the `NaN` (not a number) constant. If you are an untrained user, you have no idea what `NaN` means, but your experience with computers tells you that some kind of error has occurred. You may blame the computer or you may blame yourself — the accurate response may in fact be to blame the JavaScript developer!

To shift the point of view to the programmer, however, the script was designed to be run by a user who never makes a typing mistake, intentional or not. That, of course, is not very good programming practice. Users make mistakes. Therefore, anticipating user input that is not what would be expected is the programmer's job — input that is an exception to the rules your program wants to operate by. You must include some additional code that handles the exceptions gracefully so as to not confuse the user with unintelligible output, and perhaps even to help the user repair the input to get a result. This extra programming code handles the undesirable and erroneous input and makes your scripts considerably more user-friendly and robust.

As it turns out, JavaScript and the W3C Document Object Model liberally mix terms of exception and error within the vocabulary used to handle exceptions. As you see shortly, an exception creates an

error object, which contains information about the exception. It is safe to say that you can think of exceptions and errors as the same things.

The exception mechanism

Newcomers to JavaScript (or any programming environment, for that matter) may at first have a difficult time creating a mental model of how all this exception stuff runs within the context of the browser. It may be easy enough to understand how pages load and create object models, and how event handlers (or listeners, in the W3C DOM terminology) cause script functions to run. But a lot of action also seems to be going on in the background. For example, the event object that is generated automatically with each event action (see Chapter 32, "Event Objects") seems to sit "somewhere," while event handler functions run so that they can retrieve details about the event. After the functions finish their processing, the event object disappears, without even leaving behind a Cheshire Cat smile. Mysterious.

Browsers equipped for exception handling have more of this "stuff" running in the background, ready for your scripts when you need it. Because you have certainly viewed the details of at least one scripting error, you have already seen some of the exception-handling mechanism that is built into browsers. If a script error occurs, the browser creates in its memory an error object, whose properties contain details about the error. The precise details (described later in this chapter) vary from one browser brand to the next, but what you see in the error details readout is the default way the browser handles exceptions/errors. As browsers have matured, their makers have gone to great lengths to tone down the intrusion of script errors. For example, in NN4+, errors appeared in a separate JavaScript Console window (which must be invoked in NN4 by typing `javascript:` into the Location field; or opened directly via the Tools menu in NN6+ and Mozilla-based browsers, including Firefox and Camino). In IE4+ for Windows, the status bar comes into play again, as the icon at the bottom-left corner turns into an alert icon: Double-clicking the icon displays more information about the error. MacIE users can turn off scripting error alerts altogether. Safari 1.0 didn't divulge any script errors but a JavaScript console was added as of version 1.3.

True exception handling, however, goes further than just displaying error messages. It also provides a uniform way to let scripts guard against unusual occurrences. Ideally, the mechanism makes sure that *all* runtime errors get funneled through the same mechanism to help simplify the scripting of exception handling. The mechanism is also designed to be used intentionally as a way for your own code to generate errors in a uniform way so that other parts of your scripts can handle them quietly and intelligently. In other words, you can use the exception handling mechanism as a kind of "back channel" to communicate from one part of your scripts to another.

The JavaScript exception handling mechanism is built around two groups of program execution statements. The first group consists of the `try-catch-finally` statement triumvirate; the second group is the single `throw` statement.

Using try-catch-finally Constructions

The purpose of the `try-catch-finally` group of related statements is to provide a controlled environment in which script statements that may encounter runtime errors can run, such that your scripts can act upon exceptions without alarming the rest of the browser's error mechanisms. Each of the three statements precedes a block of code in the following syntax:

```
try
{
   statements to run
```

```
}

catch (errorInfo)
{
   statements to run if exception occurs in try block
}
finally
{
   statements to run whether or not an exception occurred [optional]
}
```

Each `try` block must be mated with a `catch` and/or `finally` block at the same nesting level, with no intervening statements. For example, a function can have a one-level `try-catch` construction inside it as follows:

```
function myFunc()
{
   try
   {
      // statements
   }
   catch (e)
   {
      // statements
   }
}
```

But if there were another `try` block nested one level deeper, a balancing `catch` or `finally` block would also have to be present at that deeper level:

```
function myFunc()
{
   try
   {
      // statements
      try
      {
         // statements
      }
      catch (e)
      {
         // statements
      }
   }
   catch (e)
   {
      // statements
   }
}
```

The statements inside the `try` block include statements that you believe are capable of generating a runtime error because of user input errors, the failure of some page component to load, or a similar

error. The presence of the `catch` block prevents errors from appearing in the browser's regular script error reporting system (for example, the JavaScript Console of Safari1.3+, NN6+, and Mozilla-based browsers).

An important term to know about exception handling of this type is *throw*. The convention is that when an operation or method call triggers an exception, it is said to "throw an exception." For example, if a script statement attempts to invoke a method of a string object, but that method does not exist for the object (perhaps you mistyped the method name), JavaScript throws an exception. Exceptions have names associated with them — a name that sometimes, but not always, reveals important information about the exception. In the mistyped method example just cited, the name of that exception is a `TypeError` (yet more evidence of how "exception" and "error" become intertwined).

The JavaScript language supported in modern browsers is not the only entity that can throw exceptions. The W3C DOM also defines categories of exceptions for DOM objects. For example, according to the Level 2 specification, the `appendChild()` method (see Chapter 26, "Generic HTML Element Objects") can throw (or *raise*, in the W3C terminology) one of three exceptions:

Exception Name	When Thrown
`HIERARCHY_REQUEST_ERR`	If the current node is of a type that does not allow children of the type of the `newChild` node, or if the node to append is one of this node's ancestors
`WRONG_DOCUMENT_ERR`	If `newChild` was created from a different document than the one that created the current node
`NO_MODIFICATION_ALLOWED_ERR`	If the current node is read-only

Because the `appendChild()` method is capable of throwing exceptions, a JavaScript statement that invokes this method should ideally be inside a `try` block. If an exception is thrown, then script execution immediately jumps to the `catch` or `finally` block associated with the `try` block. Execution does not come back to the `try` block.

A `catch` block has special behavior. Its format looks similar to a function in a way, because the `catch` keyword is followed by a pair of parentheses and an arbitrary variable that is assigned a reference to the error object whose properties are filled by the browser when the exception occurs. One of the properties of that error object is the name of the error. Therefore, the code inside the `catch` block can examine the name of the error and perhaps include some branching code to take care of a variety of different errors that are caught.

To see how this construction may appear in code, look at a hypothetical generic function whose job is to create a new element and append it to some other node. Both the type of element to be created and a reference to the parent node are passed as parameters. To take care of potential misuses of this function through the passage of improper parameter values, it includes extra error handling to treat all possible exceptions from the two DOM methods: `createElement()` and `appendChild()`. Such a function looks like Listing 21-7.

LISTING 21-7

A Hypothetical try-catch Routine

```
// generic appender
function attachToEnd(theNode, newTag)
{
   try
   {
      var newElem = document.createElement(newTag);
      theNode.appendChild(newElem);
   }
   catch (e)
   {
      switch (e.name)
      {
         case "INVALID_CHARACTER_ERR" :
            // statements to handle this createElement() error
            break;
         case "HIERARCHY_REQUEST_ERR" :
            // statements to handle this appendChild() error
            break;
         case "WRONG_DOCUMENT_ERR" :
            // statements to handle this appendChild() error
            break;
         case "NO_MODIFICATION_ALLOWED_ERR" :
            // statements to handle this appendChild() error
            break;
         default:
            // statements to handle any other error
      }
      return false;
   }
   return true;
}
```

The single `catch` block in Listing 21-7 executes only if one of the statements in the `try` block throws an exception. The exceptions may be not only one of the four specific ones named in the `catch` block, but also syntax or other errors that could occur inside the `try` block. That's why you have a last-ditch case to handle truly unexpected errors. Your job as scripter is to anticipate errors and provide clean ways for the exceptions to be handled, whether it be through judiciously worded alert dialog boxes or through self-repair. For example, in the case of the invalid character error for `createElement()`, your script may attempt to salvage the data passed to the `attachToEnd()` function and reinvoke the method, passing `theNode` value as-is and the repaired value originally passed to `newTag`. If your repairs were successful, the `try` block would execute without error and carry on with the user's being completely unaware that a nasty problem had been averted. And that's really the goal of exception handling — to save the day when something "unexpected" goes wrong so that the user isn't left confused or frustrated.

A `finally` block contains code that always executes after a `try` block, whether or not the `try` block succeeds without throwing an error. Unlike the `catch` block, a `finally` block does not receive an error object as a parameter, so it operates very much in the dark about what transpires inside the `try` block. If you include both `catch` and `finally` blocks after a `try` block, the execution path depends on whether an exception is thrown. If no exception is thrown, the `finally` block executes after the last statement of the `try` block runs. But if the `try` block throws an exception, program execution runs first to the `catch` block. After all processing within the `catch` block finishes, the `finally` block executes. In development environments that give programmers complete control over resources, such as memory allocation, a `finally` block may be used to delete some temporary items generated in the `try` block, whether or not an exception occurs in the `try` block. Currently, JavaScript's automatic memory management system reduces the need for that kind of maintenance, but you should be aware of the program execution possibilities of the `finally` block in the `try-catch-finally` context.

Real-life exceptions

The example shown in Listing 21-7 is a bit idealized. The listing assumes that the browser dutifully reports every W3C DOM exception precisely as defined in the formal specification. Unfortunately, even the latest browsers have yet to fully comply with the DOM when it comes to exception reporting. Most browsers implement additional error naming conventions and layers between actual DOM exceptions and what gets reported with the error object at the time of the exception.

If you think these discrepancies make cross-browser exception handling difficult, you're right. Even simple errors are reported differently among the two major browser brands (IE and Mozilla) and the W3C DOM specification. Until the browsers exhibit a greater unanimity in exception reporting, the smoothest development road will be for those scripters who have the luxury of writing for just one of the browser platforms, such as IE for Windows or Safari for Mac.

That said, however, one aspect of exception handling can still be used in all modern browsers without a hitch. You can take advantage of `try-catch` constructions to throw your own exceptions — a practice that is quite common in advanced programming environments.

Throwing Exceptions

The last exception handling keyword not yet covered — `throw` — makes it possible to utilize exception-handling facilities for your own management of processes, such as data entry validation. At any point inside a `try` block, you can manually throw an exception that gets picked up by the associated `catch` block. The details of the specific exception are up to you.

Syntax for the `throw` statement is as follows:

```
throw value;
```

The value you throw can be of any type, but good practice suggests that the value be an error object (described more fully later in this chapter). Whatever value you throw is assigned to the parameter of the `catch` block. Look at the following two examples. In the first, the value is a string message; in the second, the value is an error object.

Listing 21-8 presents one input text box for a number between 1 and 5. Clicking a button looks up a corresponding letter in an array and displays the letter in a second text box. The lookup script has two simple data validation routines to make sure the entry is a number and is in the desired range.

Error checking here is done manually by script. If either of the error conditions occurs, `throw` statements force execution to jump to the `catch` block. The `catch` block assigns the incoming `string` parameter to the variable `e`. The design here assumes that the message being passed is text for an alert dialog box. Not only does a single `catch` block take care of both error conditions (and conceivably any others to be added later), but the `catch` block runs within the same variable scope as the function, so that it can use the reference to the input text box to focus and select the input text if there is an error.

LISTING 21-8

Throwing String Exceptions

HTML: jsb-21-08.html

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Throwing a String Exception</title>
      <script type="text/javascript" src="../jsb-global.js"></script>
      <script type="text/javascript" src="jsb-21-08.js"></script>
   </head>
   <body>
      <h1>Throwing a String Exception</h1>
      <form id="theForm" action="validate.php" method="get">
         <p>
            <label for="visitorInput">Enter a number from 1 to 5:</label>
            <input type="text" id="visitorInput" size="5">
            <input type="button" id="getLetterButton" value="Get Letter">
            <label for="output">Matching Letter is:</label>
            <input type="text" id="output" size="5">
         </p>
      </form>
   </body>
</html>
```

JavaScript: jsb-21-08.js

```
// initialize when the page has loaded
addEvent(window, 'load', initialize);

function initialize()
{
   // add an event to the Get Letter button
   addEvent(document.getElementById('getLetterButton'), 'click', getLetter);
}

var letters = new Array("A","B","C","D","E");

function getLetter()
{
```

continued

LISTING 21-8 *(continued)*

```
    try
    {
       var inputField = document.getElementById('visitorInput');
       var inp = parseInt(inputField.value, 10);
       var outputField = document.getElementById('output');

          if (isNaN(inp))
          {
             throw "Entry was not a number.";
          }

          if (inp < 1 || inp > 5)
          {
             throw "Enter only 1 through 5.";
          }

       outputField.value = letters[inp - 1];
    }
    catch (e)
    {
       alert(e);
       outputField.value = "";
       inputField.focus();
       inputField.select();
    }
}
```

The flaw with Listing 21-8 is that if some other kind of exception were thrown inside the `try` block, the value passed to the `catch` block would be an error object, not a string. The alert dialog box displayed to the user would be meaningless. Therefore, it is better to be uniform in your `throw-catch` constructions and pass an error object.

Listing 21-9 is an updated version of Listing 21-8, demonstrating how to create an error object that gets sent to the `catch` block via `throw` statements. The HTML code is exactly the same as for listing 21-8, so that is not repeated here.

LISTING 21-9

Throwing an Error Object Exception

JavaScript: jsb-21-09.js

```
// initialize when the page has loaded
addEvent(window, 'load', initialize);

function initialize()
{
   // add an event to the Get Letter button
```

```
   addEvent(document.getElementById('getLetterButton'), 'click', getLetter);
}

var letters = new Array("A","B","C","D","E");

function getErrorObj(msg)
{
   var err = new Error(msg);
   return err;
}

function getLetter()
{
   try
   {
      var inputField = document.getElementById('visitorInput');
      var inp = parseInt(inputField.value, 10);
      var outputField = document.getElementById('output');

      if (isNaN(inp))
      {
         throw getErrorObj("Entry was not a number.");
      }

      if (inp < 1 || inp > 5)
      {
         throw getErrorObj("Enter only 1 through 5.");
      }

      outputField.value = letters[inp - 1];
   }
   catch (e)
   {
      alert(e.message);
      outputField.value = "";
      inputField.focus();
      inputField.select();
   }
}
```

The only difference to the `catch` block is that it now reads the `message` property of the incoming error object. This means that if some other exception is thrown inside the `try` block, the browser-generated message will be displayed in the alert dialog box.

In truth, however, the job really isn't complete. In all likelihood, if a browser-generated exception is thrown, the message in the alert dialog box won't mean much to the user. The error message will probably be some kind of syntax or type error — the kind of meaningless error message you often get from your favorite operating system. A better design is to branch the `catch` block so that while "intentional" exceptions thrown by your code are handled through the alert dialog box messages you've put there, other types are treated differently. To accomplish this, you can take over one of the other properties of the error object — `name` — so that your `catch` block treats your custom messages separately.

In Listing 21-10, the `getErrorObj()` function adds a custom value to the `name` property of the newly created error object. The name you assign can be any name, but you want to avoid exception names used by JavaScript or the DOM. Even if you don't know what all of those are, you can probably conjure up a suitably unique name for your error. Down in the `catch` block, a `switch` construction branches to treat the two classes of errors differently. In this simplified example, about the only possible problem other than the ones being trapped for explicitly in the `try` block would be some corruption to the page during downloading. Therefore, for this example, the branch for all other errors simply asks that the user reload the page and try again. The point is, however, that you can have as many classifications of custom and system errors as you want and handle them in a single `catch` block accordingly. Again, the HTML code is exactly the same as for listing 21-8, so it is not repeated here.

LISTING 21-10

A Custom Object Exception

JavaScript: jsb-21-10.js

```
// initialize when the page has loaded
addEvent(window, 'load', initialize);

function initialize()
{
   // add an event to the Get Letter button
   addEvent(document.getElementById('getLetterButton'), 'click', getLetter);
}

var letters = new Array("A","B","C","D","E");

function getErrorObj(msg)
{
   var err = new Error(msg);
   err.name = "MY_ERROR";
   return err;
}

function getLetter()
{
   try
   {
      var inputField = document.getElementById('visitorInput');
      var inp = parseInt(inputField.value, 10);
      var outputField = document.getElementById('output');

      if (isNaN(inp))
      {
         throw getErrorObj("Entry was not a number.");
      }

      if (inp < 1 || inp > 5)
      {
         throw getErrorObj("Enter only 1 through 5.");
```

```
      }

      outputField.value = letters[inp - 1];
   }
   catch (e)
   {
      switch (e.name)
      {
         case 'MY_ERROR' :
            alert(e.message);
            outputField.value = "";
            inputField.focus();
            inputField.select();
            break;

         default :
            alert('Reload the page and try again.');
      }
   }
}
```

If you want to see how the alternative branch of Listing 21-10 looks, copy the listing file from the CD-ROM to your hard disk and modify the last line of the `try` block so that one of the characters is dropped from the name of the array:

```
outputField.value = letter[inp - 1];
```

This may simulate the faulty loading of the page. If you enter one of the allowable values, the reload alert appears, rather than the actual message of the error object: letter is undefined. Your users will thank you.

All that's left now on this subject are the details of the error object.

Error Object

Properties	Methods
errorObject.prototype	errorObject.toString()
errorObject.constructor	
errorObject.description	
errorObject.filename	
errorObject.lineNumber	
errorObject.message	
errorObject.name	
errorObject.number	

Syntax

Creating an error object:

```
var myError = new Error("message");
var myError = Error("message");
```

Accessing static `Error` object property:

```
Error.property
```

Accessing error object properties and methods:

```
errorObject.property | method([parameters])
```

Compatibility: WinIE5+, MacIE−, NN6+, Moz+, Safari+, Opera+, Chrome+

About this object

An error object instance is created whenever an exception is thrown or when you invoke either of the constructor formats for creating an error object. Properties of the error object instance contain information about the nature of the error so that `catch` blocks can inspect the error and process error handling accordingly.

IE5 implemented an error object in advance of the ECMA-262 formal error object, and the IE5 version ended up having its own set of properties that are not part of the ECMA standard. Those proprietary properties are still part of IE5.5+, which includes the ECMA properties as well. NN6, on the other hand, started with the ECMA properties and adds two proprietary properties of its own. The browser uses these additional properties in its own script error reporting. The unfortunate bottom line for cross-browser developers is that no properties in common among all browsers support the error object. However, there are two common denominators (`name` and `message`) between IE5.5+ and other browsers.

As described earlier in this chapter, you are encouraged to create an error object whenever you use the `throw` statement for your own error control.

Properties

`constructor`

(See Chapter 15, "The String Object.")

`description`

Value: String Read/Write

Compatibility: WinIE5+, MacIE-, NN-, Moz-, Safari-, Opera-, Chrome-

The `description` property contains a descriptive string that provides some level of detail about the error. For errors thrown by the browser, the description is the same text that appears in the script error dialog box in IE. Although this property continues to be supported, the `message` property is preferred.

Related Item: `message` property

fileName
lineNumber

Value: String Read/Write

Compatibility: WinIE-, MacIE-, NN6+, Moz+, Safari-, Opera-, Chrome-

The NN6 browser uses the `fileName` and `lineNumber` properties of an error object for its own internal script error processing — these values appear as part of the error messages that are listed in the JavaScript Console. The `fileName` is the URL of the document causing the error; the `lineNumber` is the source code line number of the statement that threw the exception. These properties are exposed to JavaScript, as well, so that your error processing may use this information if it is meaningful to your application.

See the discussion of the `window.error` property in Chapter 27 for further ideas on how to use this information for bug reporting from users.

Related Item: `window.error` property

message

Value: String Read/Write

Compatibility: WinIE5.5+, MacIE-, NN6+, Moz+, Safari+, Opera+, Chrome+

The `message` property contains a descriptive string that provides some level of detail about the error. For errors thrown by the browser, the message is the same text that appears in the script error dialog box in IE and the JavaScript Console in Mozilla. By and large, these messages are more meaningful to scripters than to users. Unfortunately, there are no standards for the wording of a message for a given error. Therefore, it is hazardous at best to use the message content in a `catch` block as a means of branching to handle particular kinds of errors. You may get by with this approach if you are developing for a single browser platform, but you have no assurances that the text of a message for a particular exception may not change in future browser versions.

Custom messages for errors that your code explicitly throws can be in user-friendly language if you intend to display such messages to users. See Listings 21-8 through 21-10 for examples of this usage.

Related Item: `description` property

name

Value: String Read/Write

Compatibility: WinIE5.5+, MacIE-, NN6+, Moz+, Safari+, Opera+, Chrome+

The `name` property generally contains a word that identifies the type of error that has been thrown. The most general kind of error (and the one that is created via the `new Error()` constructor) has a name `Error`. But JavaScript errors can be of several varieties: `EvalError`, `RangeError`, `ReferenceError`, `SyntaxError`, `TypeError`, and `URIError`. Some of these error types are not necessarily intended for exposure to scripters (they're used primarily in the inner workings of the JavaScript engine), but some browsers do expose them. Unfortunately, there are some discrepancies as to the specific name supplied to this property for script errors.

When JavaScript is being used in a W3C-compatible browser, some DOM exception types are returned via the `name` property. But browsers frequently insert their own error types for this property, and, as is common in this department, little uniformity exists among browser brands.

For custom exceptions that your code explicitly throws, you can assign names as convenient. As shown in Listings 21-9 and 21-10, this information can assist a `catch` block in handling multiple categories of errors.

Related Item: `message` property

number

Value: Number Read/Write

Compatibility: WinIE5+, MacIE-, NN-, Moz-, Safari-, Opera-, Chrome-

IE5+ assigns unique numbers to each error description or message. The value of the `number` property must be massaged somewhat to retrieve a meaningful error description. Following is an example of how you must apply binary arithmetic to an error number to arrive at a meaningful result:

```
var errNum = errorObj.number & 0xFFFF;
```

Related Item: `description` property

Methods

toString()

Returns: String (see text)

Compatibility: WinIE5+, MacIE-, NN6+, Moz+, Safari+, Opera+, Chrome+

The `toString()` method for an error object should return a string description of the error. In IE5+, however, the method returns a reference to the very same error object. In Mozilla-based browsers, the method returns the `message` property string, preceded by the string `Error:` (with a space after the colon). Most typically, if you want to retrieve a human-readable expression of an error object, read its `message` (or, in IE5+, `description`) property.

Related Item: `message` property

CHAPTER

22

JavaScript Operators

IN THIS CHAPTER

Understanding operator categories

Exploring the role of operators in script statements

Recognizing operator precedence

JavaScript is rich in operators: words and symbols in expressions that perform operations on one or two values to arrive at another value. Any value on which an operator performs some action is called an operand. An expression may contain one operand and one operator (called a unary operator), as in a++, or two operands separated by one operator (called a binary operator), as in a + b. Many of the same symbols are used in a variety of operators. The combination and order of those symbols are what distinguish their powers.

Note

The vast majority of JavaScript operators have been in the language since the very beginning. But, as you may expect from an evolving language, some entries were added to the lexicon as the language matured and gained wider usage. In the rest of this chapter, compatibility charts typically govern an entire category of operator. If there are version anomalies for a particular operator within a category, they are covered in the text. The good news is that modern browsers support the entire set of JavaScript operators. ■

Operator Categories

To help you grasp the range of JavaScript operators, we group them into seven categories. We assign a wholly untraditional name (connubial) to the second group — but a name that we believe correctly identifies its purpose in the language. Table 22-1 shows the operator types.

Any expression that contains an operator evaluates to a value of some kind, meaning that a value always results from an operation. Sometimes the operator changes the value of one of the operands; other times the result is a new value. Even this simple expression

```
5 + 5
```

shows two integer operands joined by the addition operator. This expression evaluates to 10. The operator (+) is what provides the instruction for JavaScript to follow in its never-ending drive to evaluate every expression in a script.

TABLE 22-1

JavaScript Operator Categories

<table>
<tr><th>Type</th><th>What It Does</th></tr>
<tr><td>Comparison</td><td>Compares the values of two operands, deriving a result of either <code>true</code> or <code>false</code> (used extensively in conditional statements for <code>if...else</code> and <code>for</code> loop constructions)
<code>== != === !== > >= < <=</code></td></tr>
<tr><td>Connubial</td><td>Joins together two operands to produce a single value that is a result of an arithmetical or other operation on the two
<code>+ - * / % ++ -- +val -val</code></td></tr>
<tr><td>Assignment</td><td>Stuffs the value of the expression of the right-hand operand into a variable name on the left-hand side, sometimes with minor modification, as determined by the operator symbol
<code>= += -= *= /= %= <<= >= >>= >>>= &= |= ^= []</code></td></tr>
<tr><td>Boolean</td><td>Performs Boolean arithmetic on one or two Boolean operands
<code>&& || !</code></td></tr>
<tr><td>Bitwise</td><td>Performs arithmetic or column-shifting actions on the binary (base-2) representations of two operands
<code>& | ^ ~ << >> >>></code></td></tr>
<tr><td>Object</td><td>Helps scripts examine the heritage and capabilities of a particular object before they need to invoke the object and its properties or methods
<code>. [] () delete in instanceOf new this</code></td></tr>
<tr><td>Miscellaneous</td><td>A handful of operators that have special behaviors
<code>, ?: typeof void</code></td></tr>
</table>

Doing an equality comparison on two operands that, on the surface, look very different is not at all uncommon. JavaScript doesn't care what the operands look like — only how they evaluate. Two very dissimilar-looking values can, in fact, be identical when they are evaluated. Thus, an expression that compares the equality of two values, such as

```
fred == 25
```

does, in fact, evaluate to `true` if the variable `fred` has the number `25` stored in it from an earlier statement.

Comparison Operators

Compatibility: WinIE3+, MacIE3+, NN2+, Moz+, Safari+, Opera+, Chrome+

Anytime you compare two values in JavaScript, the result is a Boolean `true` or `false` value. You have a wide selection of comparison operators to choose from, depending on the kind of test you want to apply to the two operands. Table 22-2 lists all comparison operators.

TABLE 22-2

JavaScript Comparison Operators

Syntax	Name	Operand Types	Results
==	Equals	All	Boolean
!=	Does not equal	All	Boolean
===	Strictly equals	All	Boolean (IE4+, NN4+, Moz+, W3C)
!==	Strictly does not equal	All	Boolean (IE4+, NN4+, Moz+, W3C)
>	Is greater than	All	Boolean
>=	Is greater than or equal to	All	Boolean
<	Is less than	All	Boolean
<=	Is less than or equal to	All	Boolean

For numeric values, the results are the same as those you'd expect from your high school algebra class. Some examples follow, including some that may not be obvious.

```
10 == 10            // true
10 == 10.0          // true
9 != 10             // true
9 > 10              // false
9.99 <= 9.98        // false
```

Strings can also be compared on all of these levels:

```
"Fred" == "Fred"    // true
"Fred" == "fred"    // false
"Fred" > "fred"     // false
"Fran" < "Fred"     // true
```

To calculate string comparisons, JavaScript converts each character of a string to its ASCII value. Each letter, beginning with the first of the left-hand operand, is compared to the corresponding letter in the right-hand operand. With ASCII values for uppercase letters being less than those of their lowercase counterparts, an uppercase letter evaluates to being less than its lowercase equivalent. JavaScript takes case-sensitivity very seriously.

Values for comparison can also come from object properties or from values passed to functions via event handlers or other functions. A common string comparison used in data-entry validation is the one that checks whether the string has anything in it:

```
form.entry.value != ""    // true if something is in the field
```

Equality of Disparate Data Types

In versions of JavaScript before 1.2 (legacy browsers), when your script tries to compare string values consisting of numerals and real numbers (for example, `"123" == 123` or `"123" != 123`),

JavaScript anticipates that you want to compare apples to apples. Internally, it does some data type conversion that does not affect the data type of the original values (for example, if the values are in variables). But the entire situation is more complex, because other data types, such as objects, need to be dealt with. Therefore, prior to JavaScript 1.2, the rules of comparison are as shown in Table 22-3.

TABLE 22-3

Equality Comparisons for JavaScript 1.0 and 1.1

Operand A	Operand B	Internal Comparison Treatment
Object reference	Object reference	Compare object reference evaluations
Any data type	Null	Convert non-null to its object type and compare against null
Object reference	String	Convert object to string and compare strings
String	Number	Convert string to number and compare numbers

Understanding the logic of the equality comparisons in Table 22-3 requires a lot of thought on the scripter's part, because you have to be very conscious of the particular way data types may or may not be converted for equality evaluation (even though the values themselves are not converted). Supplying the proper conversion where necessary in the comparison statement is best. This ensures that what you want to compare — for example, the string versions of two values or the number versions of two values — is compared, rather than leaving the conversion up to JavaScript.

Backward-compatible conversion from a number to string entails concatenating an empty string to a number:

```
var a = "09";
var b = 9;
a == "" + b;  // result: false, because "09" does not equal "9"
```

For converting strings to numbers, you have numerous possibilities. The simplest is subtracting zero from a numeric string:

```
var a = "09";
var b = 9;
a-0 == b;  // result: true because number 9 equals number 9
```

You can also use the `parseInt()` and `parseFloat()` functions to convert strings to numbers:

```
var a = "09";
var b = 9;
parseInt(a, 10) == b;  // result: true because number 9 equals number 9
```

Of course, the other solution is to reasonably assume that your user base has a modern web browser that supports JavaScript 1.2+. To clear up the ambiguity of JavaScript's equality internal conversions, in version 1.2 JavaScript added two more operators to force the equality comparison to be extremely literal. The strictly equals (===) and strictly does not equal (!==) operators compare both the data type and value. The only time the === operator returns true is if the two operands are of the same

data type (for example, both are numbers) and the same value. Therefore, no number is ever automatically equal to a string version of that same number. Data and object types must match before their values are compared.

JavaScript 1.2+ also provides some convenient global functions for converting strings to numbers and vice versa: `String()` and `Number()`. In the following examples, the `typeof` operator shows the data type of expressions using these functions:

```
typeof 9;              // result: "number"
typeof String(9);      // result: "string"
typeof "9";            // result: "string"
typeof Number("9");    // result: "number"
```

Again, none of these functions alters the data type of the value being converted. The value of the function is what gets compared in an equality comparison:

```
var a = "09";
var b = 9;
a == String(b);    // result: false, because "09" does not equal "9"
typeof b;          // result: still "number"
Number(a) == b;    // result: true, because 9 equals 9
typeof a;          // result: still "string"
```

This discussion should impress upon you the importance of considering data types when testing the equality of two values.

Connubial Operators

Compatibility: WinIE3+, MacIE3+, NN2+, Moz+, Safari+

Connubial operators is our terminology for those operators that join two operands to yield a value related to the operands. Table 22-4 lists the connubial operators in JavaScript.

The four basic arithmetic operators for numbers are straightforward. The plus operator also works on strings to *concatenate* them (join them together), as in

```
"Scooby " + "Doo" // result = "Scooby Doo"
```

In object-oriented programming terminology, the plus sign is considered *overloaded*, meaning that it performs a different action depending on its context. Remember, too, that string concatenation does not do anything on its own to monitor or insert spaces between words. In the preceding example, the space between the names is part of the first string.

Modulo arithmetic is helpful for those times when you want to know if one number divides evenly into another. You used it in an example in Chapter 21, "Control Structures and Exception Handling," to figure out if a particular year was a leap year. Although some other leap year considerations exist for the turn of each century, the math in the example simply checked whether the year was evenly divisible by four. The result of the modulo math is the remainder after dividing the two values: When the remainder is 0, one divides evenly into the other. Here are some samples of years divided by four:

```
2002 % 4    // result = 2
2003 % 4    // result = 3
2004 % 4    // result = 0 (Bingo! Leap year!)
```

TABLE 22-4

JavaScript Connubial Operators

Syntax	Name	Operand Types	Results
+	Plus	Integer, float, string	Integer, float, string
-	Minus	Integer, float	Integer, float
*	Multiply	Integer, float	Integer, float
/	Divide	Integer, float	Integer, float
%	Modulo	Integer, float	Integer, float
++	Increment	Integer, float	Integer, float
--	Decrement	Integer, float	Integer, float
+*val*	Positive	Integer, float, string	Integer, float
-*val*	Negation	Integer, float, string	Integer, float

Thus, we used this modulo operator in a conditional statement with an `if. . .else` structure:

```
var howMany = 0;
today = new Date();
var theYear = today.getYear();
   if (theYear % 4 == 0)
   {
       howMany = 29;
   }
   else
   {
       howMany = 28;
   }
```

The modulo operator is also handy in special cases where you need to carry out some action in a loop at certain intervals, such as every third time through the loop. Here's an example of a loop that increments a counter every third time through, while looping to 100:

```
for (var i = 1; i < 100; i++)
{
   if (i % 3 == 0)
      threeCounter++;
}
```

Just as the modulo operator gives you the remainder of a division operation, some other languages offer an operator that results in the integer part of a division: integral division, or `div`. Although JavaScript does not have an explicit operator for this behavior, you can re-create it reliably if you know that your operands are always positive numbers. Use the `Math.floor()` or `Math.ceil()` methods with the division operator, as in

```
Math.floor(4/3);    // result = 1
```

In this example, `Math.floor()` replicates integral division only with values greater than or equal to `0`; `Math.ceil()` with values less than `0`.

The increment operator (++) is a *unary* operator (only one operand) and displays two different behaviors, depending on the side of the operand on which the symbols lie. Both the increment and decrement (`--`) operators can be used in conjunction with assignment operators, which we cover next.

As its name implies, the increment operator increases the value of its operand by one. But in an assignment statement, you have to pay close attention to precisely when that increase takes place. An assignment statement stuffs the value of the right operand into a variable on the left. If the ++ operator is located in front of the right operand (prefix), the right operand is incremented before the value is assigned to the variable; if the ++ operator is located after the right operand (postfix), the previous value of the operand is sent to the variable before the value is incremented. Follow this sequence to get a feel for these two behaviors:

```
var a = 10;    // initialize a to 10
var z = 0;     // initialize z to zero
z = a;         // a = 10, so z = 10
z = ++a;       // a becomes 11 before assignment, so a = 11 and z becomes 11
z = a++;       // a is still 11 before assignment, so z = 11; then a becomes 12
z = a++;       // a is still 12 before assignment, so z = 12; then a becomes 13
```

The decrement operator behaves the same way, except that the value of the operand decreases by one.

Increment and decrement operators are used most often with loop counters in `for` and `while` loops. The simpler ++ or `--` symbology is more compact than reassigning a value by adding `1` to it (such as, `z = z + 1` or `z += 1`). Because these are unary operators, you can use the increment and decrement operators without an assignment statement to adjust the value of a counting variable within a loop:

```
function doNothing()
{
   var i = 1;
   while (i < 20)
   {
      ++i;
   }
   alert(i); // loop ends with i = 20
}
```

The last pair of connubial operators are also unary operators (operating on one operand). Both the positive and negation operators can be used as shortcuts to the `Number()` global function, converting a string operand consisting of number characters to a number data type. The string operand is not changed, but the operation returns a value of the number type, as shown in the following sequence:

```
var a = "123";
var b = +a;       // b is now 123
typeof a;         // result: "string"
typeof b;         // result: "number"
```

The negation operator (`-val`) has additional power. By placing a minus sign in front of any numeric value (no space between the symbol and the value), you instruct JavaScript to evaluate a positive value as its corresponding negative value, and vice versa. The operator does not change the operand's value,

but the expression returns the modified value. The following example provides a sequence of statements to demonstrate:

```
var x = 2;
var y = 8;
var z = -x;          // z equals -2, but x still equals 2
z = -(x + y);        // z equals -10, but x still equals 2 and y equals 8
z = -x + y;          // z equals 6, but x still equals 2 and y equals 8
```

To negate a Boolean value, see the `Not (!)` operator in the discussion of Boolean operators.

Assignment Operators

Compatibility: WinIE3+, MacIE3+, NN2+, Moz+, Safari+, Opera+, Chrome+

Assignment statements are among the most common statements you write in your JavaScript scripts. These statements appear everywhere you copy a value or the results of an expression into a variable for further manipulation of that value.

You assign values to variables for many reasons, even though you could probably use the original values or expressions several times throughout a script. Here is a sampling of reasons why you should assign values to variables:

- Variable names are usually shorter
- Variable names can be more descriptive
- You may need to preserve the original value for later in the script
- The original value is a property that cannot be changed
- Invoking the same method several times in a script is not efficient

Newcomers to scripting often overlook the last reason. For instance, if a script is writing a long paragraph to a new document, it would be more efficient to assemble the paragraph as one long string of sentences before invoking the `document.createTextNode()` method to add that text to the document than to add each sentence at a time.

Table 22-5 shows the range of assignment operators in JavaScript.

As clearly demonstrated in the top group (see "Bitwise Operators" later in the chapter for information on the bottom group), assignment operators beyond the simple equals sign can save some characters in your typing, especially when you have a series of values that you're trying to bring together in subsequent statements. You've seen plenty of examples in previous chapters, where you used the add-by-value operator (+=) to work wonders with strings as you assemble a long string variable. Here's a bit of script that builds a paragraph listing regional offices:

```
var aRegionalOffices = ['New York', 'Paris', 'Istanbul', 'Lusaka'];
var sText = 'Our offices are located in ';
var sSuffix = ', ';

for (var i = 0; i < aRegionalOffices.length; i++)
{
     // doing penultimate office?
     if (i == aRegionalOffices.length-2)
```

```
            {
               sSuffix += 'and ';
            }
             // doing final office?
            else if (i == aRegionalOffices.length-1)
            {
               sSuffix = '.';
            }

          // add office to paragraph
          sText += aRegionalOffices[i] + sSuffix;
       }

       // create text node and add to paragraph element
       var oNewText = document.createTextNode(sText);
       var oParagraph = document.getElementById("p1");
       oParagraph.appendChild(oNewText);
```

TABLE 22-5

JavaScript Assignment Operators

Syntax	Name	Example	Means
=	Equals	x = y	x = y
+=	Add by value	x += y	x = x + y
-=	Subtract by value	x -= y	x = x - y
*=	Multiply by value	x *= y	x = x * y
/=	Divide by value	x /= y	x = x / y
%=	Modulo by value	x %= y	x = x % y
<<=	Left shift by value	x <<= y	x = x << y
>=	Right shift by value	x >= y	x = x > y
>>=	Zero fill by value	x >>= y	x = x >> y
>>>=	Right shift by value	x >>>= y	x = x >>> y
&=	Bitwise AND by value	x &= y	x = x & y
|=	Bitwise OR by value	x |= y	x = x | y
^=	Bitwise XOR by value	x ^= y	x = x ^ y
[]=	Destructuring assignment	[a,b]=[c,d]	a=c, b=d

The result is:

```
<p id="p1">Our offices are located in New York, Paris, Istanbul, and Lusaka.</p>
```

The script segment starts with a plain equals assignment operator to initialize the `sText` variable to the beginning of the paragraph. In the loop to add offices, you use the add-by-value operator to tack additional string values onto whatever is in the `sText` variable at the time. Without the add-by-value operator, you would use the plain equals assignment operator for each line of code to concatenate new string data to the existing string data, like this:

```
var sText = 'Our offices are located in ';
...
   sText = sText + aRegionalOffices[i] + sSuffix;
```

These enhanced assignment operators are excellent shortcuts that you should use at every turn.

One more bit of assignment syntax was added to JavaScript with version 1.7:

[ ] *(square brackets to indicate destructuring assignment)*

Compatibility: FF2+, Opera 9.5 (partial)

Square brackets can assign an array of values to a set of variables, including an array returned from a function:

```
[a, b, c] = [12, 23, 34];

function moveSoutheast(x, y)
{
   return [x+1, y+1];
}
[x, y] = moveSoutheast(x, y);
```

See also: Object Operators: [ ] square brackets to enumerate an object member

Boolean Operators

Compatibility: WinIE4+, MacIE4+, NN3+, Moz+, Safari+

Because a great deal of programming involves logic, it is no accident that the arithmetic of the logic world plays an important role. You've already seen dozens of instances where programs make all kinds of decisions based on whether a statement or expression is the Boolean value `true` or `false`. What you haven't seen much of yet is how to combine multiple Boolean values and expressions — a capacity that scripts with slightly above average complexity may need to have.

In the various conditional expressions required throughout JavaScript (such as in an `if` construction), the condition that the program must test for may be more complicated than, say, whether a variable value is greater than a certain fixed value or whether a field is not empty. Look at the case of validating a text field entry for whether the entry contains all the numbers that your script may want. Without some magical JavaScript function to tell you whether or not a string consists of all numbers, you have to break apart the entry, character by character, and examine whether each character falls within the range of 0 through 9. But that examination actually comprises two tests: You can test for any character whose ASCII value is less than `0` or greater than `9` (i.e., not a digit). Alternatively, you can test whether the character is greater than or equal to `0` and is less than or equal to `9` (i.e., is a digit). What you need is the bottom-line evaluation of both tests.

Boolean math

That's where the wonder of Boolean math comes into play. With just two values — `true` and `false` — you can assemble a string of expressions that yield Boolean results and then let Boolean arithmetic figure out whether the bottom line is `true` or `false`.

But you don't add or subtract Boolean values the same way you add or subtract numbers. Instead, you use one of three JavaScript Boolean operators at your disposal. Table 22-6 shows the three operator symbols. In case you're unfamiliar with the characters in the table, the symbols for the `OR` operator are created by typing Shift+\ (backslash).

TABLE 22-6

JavaScript Boolean Operators

Syntax	Name	Operands	Results
`&&`	`AND`	Boolean	Boolean
`||`	`OR`	Boolean	Boolean
`!`	`NOT`	One Boolean	Boolean

Using Boolean operators with Boolean operands gets tricky if you're not used to it, so we have you start with the simplest Boolean operator: `NOT`. This operator requires only one operand. The `NOT` operator precedes any Boolean value to switch it back to the opposite value (from `true` to `false`, or from `false` to `true`). For instance:

```
!true        // result = false
!(10 > 5)    // result = false
!(10 < 5)    // result = true
!("cat" == "bat")    // result = true
```

As shown here, enclosing the operand of a `NOT` expression inside parentheses is always a good idea. This forces JavaScript to evaluate the expression inside the parentheses before flipping it around with the `NOT` operator. Otherwise, you may accidentally perform the operation on only part of the intended expression, resulting in unexpected consequences.

The `AND` (`&&`) operator joins two Boolean values to reach a `true` or `false` value based on the results of both values. This brings up something called a *truth table*, which helps you visualize all the possible outcomes for each value of an operand. Table 22-7 is a truth table for the `AND` operator.

Only one condition yields a `true` result: Both operands must evaluate to `true`. Which side of the operator a `true` or `false` value lives on doesn't matter. Here are examples of each possibility:

```
5 > 1 && 50 > 10    // result = true
5 > 1 && 50 < 10    // result = false
5 < 1 && 50 > 10    // result = false
5 < 1 && 50 < 10    // result = false
```

Note

You may be wondering why parentheses aren't being used in this code to separate the comparison and Boolean expressions. The reason has to do with operator precedence, which you learn a great deal more about later in

the chapter. The short answer is that comparison operators are evaluated before Boolean operators. Even so, it's never a bad idea to use parentheses to group sub-expressions and make absolutely sure you're getting the desired result and to make your code easier for humans to read. ■

TABLE 22-7

Truth Table for the And Operator

Left Operand	AND Operator	Right Operand	Result
True	&&	True	True
True	&&	False	False
False	&&	True	False
False	&&	False	False

In contrast, the OR (||) operator is more lenient about what it evaluates to true. The reason is that if one or the other (or both) operands is true, the operation returns true. The OR operator's truth table is shown in Table 22-8.

TABLE 22-8

Truth Table for the Or Operator

Left Operand	OR Operator	Right Operand	Result
True	||	True	True
True	||	False	True
False	||	True	True
False	||	False	False

Therefore, if a true value exists on either side of the operator, a true value is the result. Take the previous examples and swap the AND operators with OR operators so that you can see the OR operator's impact on the results:

```
5 > 1 || 50 > 10    // result = true
5 > 1 || 50 < 10    // result = true
5 < 1 || 50 > 10    // result = true
5 < 1 || 50 < 10    // result = false
```

Only when both operands are false does the OR operator return false.

Boolean operators at work

Learning to apply Boolean operators to JavaScript just takes a little effort and some sketches on a pad of paper to help you figure out the logic of the expressions. Earlier, we talked about using a Boolean

operator to see whether a character fell within a range of ASCII values for data-entry validation. Listing 22-1 is a function discussed in more depth in Chapter 46, “Data-Entry Validation” (on the CD). This function accepts any string and sees whether each character of the string has an ASCII value less than 0 or greater than 9 — meaning that the input string is not a number.

LISTING 22-1

Is the Input String a Number?

```
function isNumber(inputStr)
{
   for (var i = 0; i < inputStr.length; i++)
   {
      var oneChar = inputStr.substring(i, i + 1);
         if (oneChar < "0" || oneChar > "9")
         {
            alert("Please make sure entries are numerals only.");
            return false;
         }
   }
   return true;
}
```

This code snippet combines a number of JavaScript powers to read individual characters (substrings) from a `string` object within a `for` loop. The statement that you’re interested in is the condition of the `if` construction:

```
(oneChar < "0" || oneChar > "9")
```

In one conditional statement, you use the `OR` operator to test for both possibilities. If you check the `OR` truth table (Table 22-8), you see that this expression returns `true` if either one or both tests returns `true`. If that happens, the rest of the function alerts the user about the problem and returns a `false` value to the calling statement. Only if both tests within this condition evaluate to `false` for all characters of the string does the function return a `true` value.

From the simple `OR` operator, we go to the opposite extreme, where the function checks — in one conditional statement — whether a number falls within several numeric ranges. The script in Listing 22-2 comes from the array lookup application in Chapter 53, “Application: A Lookup Table” (on the CD-ROM), in which a user enters the first three digits of a U.S. Social Security number.

LISTING 22-2

Is a Number within Discontiguous Ranges?

```
// function to determine if value is in acceptable range for this application
function inRange(inputStr)
{
   num = parseInt(inputStr)
      if (num < 1 || (num > 586 && num < 596) || (num > 599 && num < 700) ||
```

continued

LISTING 22-2 *(continued)*

```
         num > 728)
      {
         alert("Sorry, the number you entered is not part of our database.  Try
            another three-digit number.");
         return false;
      }
   return true;
}
```

By the time this function is called, the user's data entry has been validated enough for JavaScript to know that the entry is a number. Now the function must check whether the number falls outside of the various ranges for which the application contains matching data. The conditions that the function tests here are whether the number is

- Less than 1
- Greater than 586 and less than 596 (using the `AND` operator)
- Greater than 599 and less than 700 (using the `AND` operator)
- Greater than 728

Each of these tests is joined by an `OR` operator. Therefore, if any one of these conditions proves `true`, the whole `if` condition is `true`, and the user is alerted accordingly.

The alternative to combining so many Boolean expressions in one conditional statement would be to nest a series of `if` constructions. But such a construction requires not only a great deal more code but also much repetition of the alert dialog box message for each condition that could possibly fail. The combined Boolean condition is, by far, the best way to go.

Bitwise Operators

Compatibility: WinIE3+, MacIE3+, NN2+, Moz+, Safari+, Opera+, Chrome+

For scripters, bitwise operations are an advanced subject. Unless you're dealing with external processes on server-side applications or the connection to Java applets, it's rare that you will use bitwise operators. Experienced programmers who concern themselves with more specific data types (such as long integers) are quite comfortable in this arena, so we simply provide an explanation of JavaScript capabilities. Table 22-9 lists JavaScript bitwise operators.

The numeric value operands can appear in any of the JavaScript language's three numeric literal bases (decimal, octal, or hexadecimal). As soon as the operator has an operand, the value is converted to binary representation (32 bits long). For the first three bitwise operations, the individual bits of one operand are compared with their counterparts in the other operand. The resulting value for each bit depends on the operator:

- **Bitwise** `AND`**:** 1 if both digits are 1
- **Bitwise** `OR`**:** 1 if either digit is 1
- **Bitwise** `Exclusive OR`**:** 1 if only one digit is a 1

TABLE 22-9

JavaScript's Bitwise Operators

Operator	Name	Left Operand	Right Operand
&	Bitwise AND	Integer value	Integer value
|	Bitwise OR	Integer value	Integer value
^	Bitwise XOR	Integer value	Integer value
~	Bitwise NOT	(None)	Integer value
<<	Left shift	Integer value	Shift amount
>>	Right shift	Integer value	Shift amount
>>>	Zero fill right shift	Integer value	Shift amount

Bitwise `NOT`, a unary operator, inverts the value of every bit in the single operand. The bitwise shift operators operate on a single operand. The second operand specifies the number of positions to shift the value's binary digits in the direction of the arrows of the operator symbols.

Example

For example, the left shift (<<) operator has the following effect:

```
4 << 2 // result = 16
```

The reason for this shifting is that the binary representation for decimal 4 is 00000100 (to eight digits, anyway). The left shift operator instructs JavaScript to shift all digits two places to the left, giving the binary result 00010000, which converts to 16 in decimal format. If you're interested in experimenting with these operators, use The Evaluator to evaluate sample expressions for yourself (see Chapter 4, "JavaScript Essentials"). More advanced books on C and C++ programming are also of help.

Object Operators

The next group of operators concern themselves with objects (including native JavaScript, DOM, and custom objects) and data types. Most of these have been implemented after the earliest JavaScript browsers, so each one has its own compatibility rating.

. (dot or period)

Compatibility: WinIE3+, MacIE3+, NN2+, Moz+, Safari+, Opera+, Chrome+

The dot operator indicates that the object to its left owns or contains the resource to its right, as in `object.property` and `object.method()`. Examples:

`native JavaScript object`

```
var s = new String('syzygy');
```

```
var len = s.length;
var pos = s.indexOf('zyg'); // result: pos == 2
```

```
custom JavaScript object

function myMethod()
{
   // (custom code goes here)
}

function myCustomObject()
{
   this.myProperty = 'something';
   this.myMethod = myMethod;
}

var o = new myCustomObject();
var p = o.newProperty;
o.myMethod();
```

```
Document Object Model object

var sTitle = document.title;
var oThing = document.getElementById("elementID");
```

[ ] *(square brackets to enumerate object member)*

Compatibility: WinIE3+, MacIE3+, NN2+, Moz+, Safari+, Opera+, Chrome+

Square brackets enumerate a member of an object, such as an array element:

```
define an array:

var a = ['homo erectus', , 'homo sapiens'];

enumerate an array element:

a[5] = 'sixth element';

enumerate an object property:

a['color'] = 'puce';
```

See also: Assignment Operators: [ ] square brackets to indicate destructuring assignment

Delete

Compatibility: WinIE4+, MacIE4+, NN4+, Moz+, Safari+, Opera+, Chrome+

Array objects do not contain a method to remove an element from the collection, nor do custom objects offer a method to remove a property. You can always empty the data in an array item or property by setting its value to an empty string or `null`, but the array element or property remains in the object. With the `delete` operator, you can completely remove the element or property.

There is special behavior about deleting an array item that you should bear in mind. If your array uses numeric indexes, a deletion of a given index removes that index value from the total array, but without collapsing the array (which would alter index values of items higher than the deleted item).

Example

Consider the following simple dense array:

```
var oceans = new Array("Atlantic", "Pacific", "Indian","Arctic");
```

This kind of array automatically assigns numeric indices to its entries for addressing later in constructions such as `for` loops:

```
for (var i = 0; i < oceans.length; i++)
{
   if (oceans[i] == form.destination.value)
   {
      // statements
   }
}
```

If you then issue the statement

```
delete oceans[2];
```

the array undergoes significant changes. First, the third element is removed from the array. Note that the length of the array does not change. Even so, the index value `(2)` is removed from the array, such that, schematically, the array looks like the following:

```
oceans[0] = "Atlantic";
oceans[1] = "Pacific";
oceans[3] = "Arctic";
```

If you try to reference `oceans[2]` in this collection, the result is `undefined`.

The `delete` operator works best on arrays that have named indexes since there is less confusion due to deleted numeric indexes. Your scripts will have more control over the remaining entries and their values, because they don't rely on what could be a missing entry of a numeric index sequence.

One aspect of this deletion action that JavaScript doesn't provide is absolute control over memory utilization. All garbage collection is managed by the JavaScript interpreter engine, which tries to recognize when items occupying memory are no longer needed, at which time the unused browser's application memory may be recovered. But you cannot force the browser to perform its garbage collection task. So, deleting an entry from an array doesn't guarantee an immediate release of its associated memory.

`in`

Compatibility: WinIE5.5+, MacIE−, NN6+, Moz+, Safari+, Opera+, Chrome+

The `in` operator lets a script statement inspect an object to see if it has a named property or method. The operand to the left of the operator is a string reference to the property or method (just the method name, without parentheses); the operand to the right of the operator is the object being inspected. If the object knows the property or method, the expression returns `true`. Thus, you can use the `in` operator in expressions used for conditional expressions.

Example

You can experiment with this operator in The Evaluator (see Chapter 4). For example, to prove that the `write()` method is implemented for the `document` object, the expression you type into the top text box of The Evaluator is:

```
"write" in document
```

But compare the implementation of the W3C DOM `document.defaultView` property in IE5.5+ and modern W3C browsers:

```
"defaultView" in document
```

In NN6+, Mozilla (including Firefox and Camino), and Safari, the result is `true`, while in IE5.5 and IE6, the result is `false`.

Having this operator around for conditional expressions lets you go far beyond simple object detection for branching code. For example, if you intend to use `document.defaultView` in your script, you can make sure that the property is supported before referencing it (assuming your users all have browsers that know the `in` operator).

instanceof

Compatibility: WinIE5+, MacIE-, NN6+, Moz+, Safari+, Opera+, Chrome+

The `instanceof` operator lets a script test whether an object is an instance of a particular JavaScript native object or of a DOM object. The operand to the left side of the operator is the value under test; the value to the right of the operand is a reference to the root class from which the value is suspected of being constructed.

For native JavaScript classes, the kinds of object references to the right of the operator include such static objects as `Date`, `String`, `Number`, `Boolean`, `Object`, `Array`, and `RegExp`. You need to be mindful of how native JavaScript classes can sometimes be children of other native classes, which means that a value may be an instance of two different static objects.

Example

Consider the following sequence (which you can follow along in The Evaluator):

```
a = new Array(1,2,3);
a instanceof Array;
```

The second statement yields a result of `true`, because the `Array` constructor was used to generate the object. But the JavaScript `Array` is, itself, an instance of the root `Object` object. Therefore, both of the following statements evaluate to `true`:

```
a instanceof Object;
Array instanceof Object;
```

new

Compatibility: WinIE3+, MacIE3+, NN2+, Moz+, Safari+, Opera+, Chrome+

Most JavaScript core objects have constructor functions built into the language. To access those functions, you use the `new` operator along with the name of the constructor. The function returns a reference to the object instance, which your scripts can then use to get and set properties or invoke object methods. For example, creating a `new` date object requires invoking the `Date` object's constructor, as follows:

```
var today = new Date();
```

Some object constructor functions require parameters to help define the object. Others, as in the case of the `Date` object, can accept a number of different parameter formats, depending on the format of the date information with which you set the initial object. The `new` operator can be used with the following core language objects, as of each specified JavaScript version:

JavaScript 1.0	JavaScript 1.1	JavaScript 1.2	JavaScript 1.5
`Date`	`Array`	`RegExp`	`Error`
`Object`	`Boolean`		
(Custom object)	`Function`		
	`Image`		
	`Number`		
	`String`		

this

Compatibility: WinIE3+, MacIE3+, NN2+, Moz+, Safari+, Opera+, Chrome+

JavaScript includes an operator that allows script statements to refer to the very object in which they are located. The self-referential operator is `this`.

The most common application of the `this` operator is in event handlers that pass references of themselves to functions for further processing.

A function receiving the value assigns it to a variable that can be used to reference the sender, its properties, and its methods.

Example

Because the `this` operator references an object, that object's properties can be exposed with the aid of the operator. For example:

```
oInputField.onchange = validateInput;

function validateInput(evt)
{
   var sInputvalue = this.value;
}
```

When the visitor changes the contents of the input field — triggering the `onchange` event, which, in turn, executes the `validateInput()` function — the code inside that function recognizes `this` as being the input field object to which it belongs. In that context, `this.value` represents the value entered by the visitor.

In the bad old days of inline JavaScript, this would look like:

```
<input type="text" name="entry" onchange="validateInput(this)" />
```

The `this` operator also works inside other objects, such as custom objects. When you define a constructor function for a custom object, using the `this` operator to define properties of the object and assign values to those properties is common practice. Consider the following example of an object creation sequence:

```
function bottledWater(brand, ozSize, flavor)
{
   this.brand = brand;
   this.ozSize = ozSize;
   this.flavor = flavor;
}
var myWater = new bottledWater("Eau de Odio", 16, "original");
```

When the new object is created via the constructor function, the `this` operators define each property of the object and then assign the corresponding incoming value to that property. Using the same names for the properties and parameter variables is perfectly fine and makes the constructor easy to maintain.

By extension, if you assign a function as an object's property (to behave as a method for the object), the `this` operator inside that function refers to the object invoking the function, offering an avenue to the object's properties. For example, if we add the following function definition and statement to the `myWater` object created just above, the function can directly access the `brand` property of the object:

```
function adSlogan()
{
   return "Drink " + this.brand + ", it's wet and wild!";
}
myWater.getSlogan = adSlogan;
```

When a statement invokes the `myWater.getSlogan()` method, the object invokes the `adSlogan()` function, but all within the context of the `myWater` object. Thus, the `this` operator applies to the surrounding object, making the brand property available via the `this` operator (`this.brand`).

Miscellaneous Operators

The final group of operators doesn't fit into any of the previous categories, but they are no less important.

,

Compatibility: WinIE3+, MacIE3+, NN2+, Moz+, Safari+, Opera+, Chrome+

The comma operator indicates a series of expressions that are to be evaluated in left-to-right sequence. Most typically, this operator is used to permit multiple variable initializations. For example, you can combine the declaration of several variables in a single `var` statement, as follows:

```
var name, address, serialNumber;
```

Another situation where you could use this operator is within the expressions of a `for` loop construction. In the following example, two different counting variables are initialized and incremented at different rates. When the loop begins, both variables are initialized at zero (they don't have to be, but this example starts that way); for each subsequent trip through the loop, one variable is incremented by one, while the other is incremented by 10:

```
for (var i=0, j=0; i < someLength; i++, j+10)
{
   ...
}
```

Don't confuse the comma operator with the semicolon delimiter between statements.

?:

Compatibility: WinIE3+, MacIE3+, NN2+, Moz+, Safari+, Opera+, Chrome+

The conditional operator is a shortcut way of expressing the `if. . .else` conditional construction covered in Chapter 21, "Control Structures and Exception Handling." This operator is typically used in concert with an assignment operator to assign one of two values to a variable based on the result of a conditional expression. The formal syntax for the conditional operator is:

```
condition ? expressionIfTrue : expressionIfFalse
```

If used with an assignment operator, the syntax is:

```
var = condition ? expressionIfTrue : expressionIfFalse;
```

No matter how you use the operator, the important point to remember is that an expression that contains this operator evaluates to one of the two expressions following the question mark symbol. In truth, either expression could invoke any JavaScript, including calling other functions, or even nesting further conditional operators within one of the expressions to achieve the equivalent of nested `if. . .else` constructions. To assure proper resolution of nested conditionals, surround inner expressions with parentheses to make sure that they evaluate before the outer expression evaluates. As an example, the following statement assigns one of three strings to a variable depending on the date within a month:

```
var monthPart = (dateNum <= 10) ? "early" : ((dateNum <= 20) ?
   "middle" : "late");
```

When the statement is evaluated, the inner conditional expression at the right of the first colon is evaluated, returning either `middle` or `late`; then the outer conditional expression is evaluated, returning either `early` or the result of the inner conditional expression.

typeof

Compatibility: WinIE3+, MacIE3+, NN3+, Moz+, Safari+, Opera+, Chrome+

Unlike most other operators, which are predominantly concerned with arithmetic and logic, the unary `typeof` operator defines the kind of value to which a variable or expression evaluates. Typically, this operator is used to identify whether a variable value is one of the following types: `number`, `string`, `boolean`, `object`, `function`, or `undefined`.

Example

Having this investigative capability in JavaScript is helpful because variables cannot only contain any one of those data types but can change their data type on the fly. Your scripts may need to handle a value differently based on the value's type. The most common use of the `typeof` property is as part of a condition. For example:

```
if (typeof myVal == "number")
{
   myVal = parseInt(myVal);
}
```

The evaluated value of the `typeof` operation is, itself, a string.

void

Compatibility: WinIE3+, MacIE3+, NN3+, Moz+, Safari+, Opera+, Chrome+

In all scriptable browsers you can use the `javascript:` pseudo-protocol to supply the parameter for `href` and `src` attributes in HTML tags, such as links. In the process, you have to be careful that the function or statement being invoked by the URL does not return or evaluate to any values. If a value comes back from such an expression, then that value, or sometimes the directory of the client's hard disk, often replaces the page content. To avoid this possibility, use the `void` operator in front of the function or expression being invoked by the `javascript:` URL.

Example

The best way to use this construction is to place the operator before the expression or function and separate them by a space, as in

```
javascript: void doSomething();
```

On occasion, you may have to wrap the expression inside parentheses after the `void` operator. Using parentheses is necessary only when the expression contains operators of a lower precedence than the `void` operator (see the following section, "Operator Precedence"). But don't automatically wrap all expressions in parentheses, because some browsers can experience problems with these. Even so, it is common practice to assign the following URL to the `href` attribute of an `a` link whose `onclick` event handler does all of the work:

```
href="javascript: void (0)"
```

The `void` operator makes sure the function or expression returns no value that the HTML attribute can use. Such a link's `onclick` event handler should also inhibit the natural behavior of a clicked link (for example, by evaluating to `return false`).

Operator Precedence

When you start working with complex expressions that hold a number of operators (for example, Listing 22-2), knowing the order in which JavaScript evaluates those expressions is vital. JavaScript assigns different priorities or weights to types of operators in an effort to achieve uniformity in the way it evaluates complex expressions.

In the following expression

```
10 + 4 * 5 // result = 30
```

JavaScript uses its precedence scheme to perform the multiplication before the addition — regardless of where the operators appear in the statement. In other words, JavaScript first multiplies 4 by 5 and then adds that result to 10 to get a result of 30. That may not be the way you want this expression to evaluate. Perhaps your intention was to add the 10 and 4 first, and then to multiply that sum by 5. To make that happen, you have to override JavaScript's natural operator precedence. To do that, you must use parentheses to enclose an operator with lower precedence. The following statement shows how you adjust the previous expression to make it behave differently:

```
(10 + 4) * 5 // result = 70
```

That one set of parentheses has a great impact on the outcome. Parentheses have the highest precedence in JavaScript, and if you nest parentheses in an expression, the innermost set evaluates first.

For help in constructing complex expressions, refer to Table 22-10 for JavaScript's operator precedence. Our general practice: When in doubt about complex precedence issues, we build the expression with lots of parentheses according to the way we want the internal expressions to evaluate.

This precedence scheme is devised to help you avoid being faced with two operators from the same precedence level that often appear in the same expression. When it does happen (such as with addition and subtraction), JavaScript begins evaluating the expression from left to right.

One related fact involves a string of Boolean expressions strung together for a conditional statement (see Listing 22-2). JavaScript follows what is called *short-circuit evaluation*. As the nested expressions are evaluated left to right, the fate of the entire condition can sometimes be determined before all expressions are evaluated. Anytime JavaScript encounters an `AND` operator, if the left operand evaluates to `false`, the entire expression evaluates to `false` without JavaScript's even bothering to evaluate the right operand. For an `Or` operator, if the left operand is `true`, JavaScript short-circuits that expression to `true`. This feature can trip you up if you don't perform enough testing on your scripts: If a syntax error or other error exists in a right operand, and you fail to test the expression in a way that forces that right operand to evaluate, you may not know that a bug exists in your code. Users of your page, of course, will find the bug quickly. Do your testing to head bugs off at the pass.

TABLE 22-10

JavaScript Operator Precedence

Precedence Level	Operator	Notes
1	`()`	From innermost to outermost
	`[]`	Array index value
	`function()`	Any remote function call
2	`!`	Boolean `NOT`
	`~`	Bitwise `NOT`
	`-`	Negation
	`++`	Increment
	`--`	Decrement
	`new`	
	`typeof`	
	`void`	
	`delete`	Delete array or object entry
3	`*`	Multiplication
	`/`	Division
	`%`	Modulo
4	`+`	Addition
	`-`	Subtraction
5	`<<`	Bitwise shifts
	`>`	
	`>>`	
6	`<`	Comparison operators
	`<=`	
	`>`	
	`>=`	
7	`==`	Equality
	`!=`	
8	`&`	Bitwise `AND`
9	`^`	Bitwise `XOR`
10	`|`	Bitwise `OR`
11	`&&`	Boolean `AND`
12	`||`	Boolean `OR`

Precedence Level	Operator	Notes
13	?	Conditional expression
14	= += -= *= /= %= <<= >= >>= &= ^= |=	Assignment operators
15	,	Comma (parameter delimiter)

Note

Notice, too, that all math and string concatenation is performed prior to any comparison operators. This enables all expressions that act as operands for comparisons to evaluate fully before they are compared. ■

The key to working with complex expressions is to isolate individual expressions and to try them out by themselves, if you can. See additional debugging tips in Chapter 48, “Debugging Scripts,” on the CD-ROM.

CHAPTER 23

Function Objects and Custom Objects

IN THIS CHAPTER

Creating function blocks

Passing parameters to functions

Creating your own objects

The Object object

By now, you've seen dozens of JavaScript functions in action and probably have a pretty good feel for the way they work. This chapter provides the `Function` object specification and delves into the fun prospect of creating objects in your JavaScript code. If you've missed out on the object-oriented (OO) programming revolution, now is your chance to join. JavaScript is surprisingly full-featured when it comes to supporting OO and allowing you to develop scripts that rely heavily on custom objects.

Function Object

Properties	Methods	Event Handlers
`arguments`	`apply()`	
`arity`	`call()`	
`caller`	`toString()`	
`constructor`	`valueOf()`	
`length`		
`prototype`		

Syntax

Creating a `Function` object:

```
function functionName([arg1,...[,argN]])
{
   statement(s)
}
```

```
var funcName = new Function(["argName1",...[,"argNameN"],
        "statement1;...[;statementN]"])object.eventHandlerName =
        function([arg1,...[,argN]]) {statement(s)}
```

Accessing `Function` object properties and methods:

```
functionObject.property | method([parameters])
```

Compatibility: WinIE3+, MacIE3+, NN2+, Moz+, Safari+

About this object

JavaScript accommodates what other languages might call procedures, subroutines, and functions all in one type of structure: the *custom function*. A function may return a value (if programmed to do so with the `return` keyword), but it does not have to return any value. With the exception of JavaScript code that executes as the document loads, all deferred processing takes place in functions.

Although you can create functions that are hundreds of lines long, we recommend you break up longer processes into shorter functions. Among the reasons for doing so: smaller chunks are easier to write and debug; building blocks make it easier to visualize the entire script; you can make functions generalizable and reusable for other scripts; and other parts of the script or other open frames can use the functions.

Learning how to write good, reusable functions takes time and experience. But the earlier you understand the importance of this concept, the more you will be on the lookout for good examples in other people's scripts on the web.

Creating functions

The standard way of defining a function in your script involves following a simple pattern and then filling in the details. The formal syntax definition for a function is:

```
function functionName( [arg1] ... [, argN])
{
   statement(s)
}
```

The task of assigning a function name helps you determine the precise scope of activity of the function. If you find that you can't reduce the planned task for the function to a simple one- to three-word name (which is then condensed into one contiguous sequence of characters for the `functionName`), perhaps you're asking the function to do too much. A better idea may be to break the job into two or more functions. As you start to design a function, be on the lookout for functions that you can call from the one you're writing. If you find yourself copying and pasting lines of code from one part of a function to another because you're performing the same operation in different spots within the function, it may be time to break that segment out into its own function.

Here's a quick example of a simple function that accepts a single argument, a name, and then returns a string greeting that includes the name:

```
function sayHello(name)
{
   return ("Hello, " + name + ".");
}
```

You can also create what is called an *anonymous function* using the `new Function()` constructor. In reality, you assign a name to this anonymous function as follows:

```
var funcName = new Function(["argName1",...[,"argNameN"],
                            "statement1;...[;statementN]"]);
```

This other way of building a function is particularly helpful when your scripts need to create a function after a document loads. All the components of a function are present in this definition. Each function parameter name is supplied as a string value, separated from the others by commas. The final parameter string consists of the statements that execute whenever the function is called. (Recall that quoted strings cannot be broken onto several lines; the continuation character at the end of the second line in the code segment below is used only because this book has a finite width to its pages.) Separate each JavaScript statement with a semicolon, and enclose the entire sequence of statements inside quotes, as in the following:

```
var willItFit = new Function("width","height",
                "var sx = screen.availWidth; var sy = screen.availHeight;↩
                return (sx >= width && sy >= height)");
```

The `willItFit()` function takes two parameters; the body of the function defines two local variables (`sx` and `sy`) and then returns a Boolean value of `true` if the incoming parameters are smaller than the local variables. In traditional form, this function is defined as follows:

```
function willItFit(width, height)
{
   var sx = screen.availWidth;
   var sy = screen.availHeight;
   return (sx >= width && sy >= height);
}
```

When this function exists in the browser's memory, you can invoke it like any other function:

```
if (willItFit(400,500))
{
   // statements to load image
}
```

One last function creation format is available in IE4+, NN4+, Moz, and other W3C DOM browsers. This advanced technique, called a *lambda expression,* provides a shortcut for creating a reference to an anonymous function (truly anonymous because the function has no name that you can reference later). The common application of this technique is to assign function references to event handlers when an event object also must be passed. The following is an example of how to assign an anonymous function to an `onchange` event handler for a form control:

```
document.forms[0].age.onchange = function(event)
     {isNumber(document.forms[0].age)}
```

Because an anonymous function evaluates to a reference to a function object, you can use either form of anonymous function in situations where a function reference is called for, including parameters of methods or other functions.

Nesting functions

Modern browsers also provide for nesting functions inside one another. In the absence of nested functions, each function definition is defined at the global level, whereby every function is exposed and available to all other script code. With nested functions, you can encapsulate the exposure of a function inside another and make that nested function private to the enclosing function. Of course, we don't recommend reusing function names with this technique, although you can create nested functions with the same name inside multiple global-level functions, as the following skeletal structure shows:

```
function outerA()
{
   // statements
   function innerA()
   {
      // statements
   }
   // statements
}
function outerB()
{
   // statements
   function innerA()
   {
      // statements
   }
   function innerB()
   {
      // statements
   }
   // statements
}
```

A good time to apply a nested function is when a sequence of statements needs to be invoked in multiple places within a large function, and those statements have meaning only within the context of the larger function. In other words, rather than break out the repeated sequence as a separate global function, you keep it all within the scope of the larger function.

Note

The premise behind nesting a function is to isolate the function and make it private from the overall script. This results in cleaner script code because nothing is exposed globally without a good reason. As an admittedly strange analogy, the water heater in your house could feasibly be placed outside (global), but it's generally safer and more organized to place it inside (local). Unlike your mailbox, which must interact with the outside (global) world, your water heater plays an internal (local) role, and therefore lives inside your house. And after all, we don't want to make our home's hot water globally available. ■

You can access a nested function only from statements in its containing function (but in any order). Moreover, all variables defined in the outer function (including parameter variables) are accessible to the inner function; but variables defined in an inner function are not accessible to the outer function. See the section "Variable scope: Globals and locals" later in this chapter for details on how variables are visible to various components of a script.

Function parameters

The function definition requires a set of parentheses after the `functionName`. If the function does not rely on any information arriving with it when invoked, the parentheses can be empty. But when some kind of data is arriving with a call to the function, you need to assign names to each parameter. Virtually any kind of value can be a parameter: strings, numbers, Boolean operators, and even complete object references such as a form or form element. Choose names for these variables that help you remember the content of those values; also, avoid reusing existing object names as variable names because it's easy to get confused when objects and variables with the same name appear in the same statements. You must avoid using JavaScript keywords (including the reserved words listed in Appendix C, on the CD) and any global variable name defined elsewhere in your script. (See more about global variables in the following sections.)

JavaScript is forgiving about matching the number of parameters in the function definition with the number of parameters passed along from the calling statement. If you define a function with three parameters and the calling statement specifies only two, the third parameter variable value in that function is assigned a `null` value. For example:

```
function saveWinners(first, second, third)
{
   // statements
}
oneFunction("George","Gracie");
```

In the preceding example, the values of `first` and `second` inside the function are `"George"` and `"Gracie"`, respectively; the value of `third` is `null`.

At the opposite end of the spectrum, JavaScript also doesn't balk if you send more parameters from the calling statement than the number of parameter variables specified in the function definition. In fact, the language includes a mechanism — the `arguments` property — that you can add to your function to gather any extraneous parameters that should read your function.

Properties

arguments (deprecated)

Value: Array of arguments Read-Only

Compatibility: WinIE4+, MacIE4+, NN3+, Moz+, Safari+, Chrome+

When a function receives parameter values from the statement that invoked the function, those parameter values are silently assigned to the `arguments` property of the function object. This property is an array of the values, with each parameter value assigned to a zero-based index entry in the array — whether or not parameters are defined for it. You can find out how many parameters are sent by extracting `functionName.arguments.length`. For example, if four parameters are passed, `functionName.arguments.length` returns 4. Then, you can use array notation (`functionName.arguments[i]`) to extract the values of any parameter(s) you want.

Theoretically, you never have to define parameter variables for your functions because you can extract the desired arguments array entry instead. Well-chosen parameter variable names, however, are much more readable, so we recommend them over the `arguments` property in most cases. But you may run into situations in which a single function definition needs to handle multiple calls to the function

when each call may have a different number of parameters. The function knows how to handle any arguments over and above the ones given names as parameter variables.

Note

It is necessary in some cases to create a function that deliberately accepts a varied number of arguments, in which case the `arguments` property is the only way to access and process the arguments. For example, if you wanted to create a function that averages test scores, there's a good chance the number of scores may vary, in which case you would write the function so that it loops through the `arguments` array, adding up the scores as it carries out the calculation. ■

Example

See Listings 23-1 and 23-2 for a demonstration of both the `arguments` and `caller` properties.

arity (deprecated)

Value: Integer Read-Only

Compatibility: WinIE−, MacIE−, NN4+, Moz−, Safari−, Chrome+

See the discussion of the `length` property later in this chapter.

caller

Value: Function object reference Read-Only

Compatibility: WinIE4+, MacIE4+, NN3+, Moz+, Safari−, Chrome+

When one function invokes another, a chain is established between the two, primarily so that a returned value knows where to go. Therefore, a function invoked by another maintains a reference to the function that called it. Such information is automatically stored in a `Function` object as the `caller` property. This relationship reminds me a bit of a subwindow's `opener` property, which points to the window or frame responsible for the subwindow's creation. The value is valid only while the called function is running at the request of another function; when a function isn't running, its `caller` property is `null`.

The value of the `caller` property is a reference to a `Function` object, so you can inspect its `arguments` and `caller` properties (in case it was called by yet another function). Thus, a function can look back at a calling function to see what values it was passed.

The `functionName.caller` property reveals the contents of an entire function definition if the current function was called from another function (including an event handler). If the call for a function comes from a regular JavaScript statement not originating inside a function, the `functionName.caller` property is `null`.

Example

To help you grasp all that these two properties yield, study Listing 23-1.

Note

The function to assign event handlers throughout the code in this chapter and much of the book is `addEvent()`, a cross-browser event handler explained in detail in Chapter 32, "Event Objects." ■

LISTING 23-1

A Function's arguments and caller Properties

HTML: jsb-23-01.html

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Function call information</title>
      <script type="text/javascript" src="../jsb-global.js"></script>
      <script type="text/javascript" src="jsb-23-01.js"></script>
   </head>
   <body>
      <h1>Function call information</h1>
      <div id="placeHolder"></div>
   </body>
</html>
```

JavaScript: jsb-23-01.js

```
// initialize when the page has loaded
addEvent(window, "load", showArgCaller);

var newElem;
var newText;

function showArgCaller ()
{
   // Call the hansel function twice: once directly,
   // second indirectly from the gretel function.
   hansel(1, "two", 3);
   gretel(4, "five", 6, "seven");
}

function hansel(x,y)
{
   var args = hansel.arguments;
   var placeHolderElement = document.getElementById("placeHolder");

      if (placeHolderElement)
      {
         // a header line
         newElem = document.createElement("h3");
         newElem.className = "objProperty";
         newText = document.createTextNode("The caller and the arguments");
         // insert the 1st line (text node) into the new h3
         newElem.appendChild(newText);
         // insert the completed h3 into placeholder
```

continued

functionObject.caller()

LISTING 23-1 *(continued)*

```
         placeHolderElement.appendChild(newElem);

         // the next line is the caller
         newElem = document.createElement("div");
         newElem.className = "objProperty";
         newText = document.createTextNode("hansel.caller is " + hansel.caller);
         newElem.appendChild(newText);
         placeHolderElement.appendChild(newElem);

         // the next line is the number or arguments
         newElem = document.createElement("div");
         newElem.className = "objProperty";
         newText = document.createTextNode("hansel.arguments.length is " +
                                           hansel.arguments.length);
         newElem.appendChild(newText);
         placeHolderElement.appendChild(newElem);

         // the rest of the lines are the arguments
         for (var i = 0; i < args.length; i++) {
            newElem = document.createElement("div");
            newElem.className = "objProperty";
            newText = document.createTextNode("argument " + i + " is " + args[i]);
            newElem.appendChild(newText);
            placeHolderElement.appendChild(newElem);
         }
      }
}

function gretel(x,y,z)
{
   today = new Date();
   thisYear = today.getFullYear();
   hansel(x,y,z,thisYear);
}
```

When you load this page, the following JavaScript results appear in the browser window:

```
The caller and the arguments
hansel.caller is function showArgCaller()
{
  hansel(1, "two", 3);
  gretel(4, "five", 6, "seven");
}
hansel.arguments.length is 3
argument 0 is 1
argument 1 is two
argument 2 is 3
The caller and the arguments
```

```
hansel.caller is function gretel(x, y, z)
{
  today = new Date();
  thisYear = today.getFullYear();
  hansel(x,y,z,thisYear);
}
hansel.arguments.length is 4
argument 0 is 4
argument 1 is five
argument 2 is 6
argument 3 is 2010 (or whatever the current year is)
```

As the document loads, the `hansel()` function is called directly in the onload event handler in the external script. It passes three arguments, even though the `hansel()` function defines only two. The `hansel.arguments` property picks up all three arguments just the same. Then the onload event handler invokes the `gretel()` function, which, in turn, calls `hansel()` again. But when `gretel()` makes the call, it passes four parameters. The `gretel()` function picks up only three of the four arguments sent by the calling statement. It also inserts another value from its own calculations as an extra parameter to be sent to `hansel()`. The `hansel.caller` property reveals the entire content of the `gretel()` function, whereas `hansel.arguments` picks up all four parameters, including the year value introduced by the `gretel()` function.

constructor

(See `object.constructor` later in this chapter.)

length

Value: Integer Read-Only

Compatibility: WinIE4+, MacIE4+, NN4+, Moz+, Safari+

As the `arguments` property of a function proves, JavaScript is very forgiving about matching the number of parameters passed to a function with the number of parameter variables defined for the function. But a script can examine the `length` property of a `Function` object to see precisely how many parameter variables are defined for the function. A reference to the property starts with the function name representing the object. For example, consider the following function definition shell:

```
function identify(name, rank, serialNum)
{
   // ...
}
```

A script statement anywhere outside of the function can read the number of parameters with the reference:

```
identify.length
```

The value of the property in the preceding example is 3. The `length` property supercedes the NN-only `arity` property.

prototype

(See Chapter 18.)

Methods

apply(*[thisObj[, argumentsArray]]*)
call(*[thisObj[, arg1[, arg2[, ... argN]]]]*)

Returns: Nothing

Compatibility: WinIE5.5+, MacIE−, NN6+, Moz+, Safari+, Chrome+

The `apply()` and `call()` methods of a `Function` object invoke the function. This may seem redundant to the normal way in which script statements invoke functions by simply naming the function, following it with parentheses, passing parameters, and so on. The difference with these methods is that you can invoke the function through the `apply()` and `call()` methods using only a reference to the function. For example, if your script defines a function through the `new Function()` constructor (or other anonymous shortcut supported by the browser), you receive a reference to the function as a result of the constructor. To invoke the function later using only that reference (presumably preserved in a global variable), use either the `apply()` or `call()` method. Both of these methods achieve the same result, but choosing one method over the other depends on the form in which the function's parameters are conveyed (more about that in a moment).

The first parameter of both methods is a reference to the object that the function treats as the current object. For garden-variety functions defined in your script, use the keyword `this`, which means that the function's context becomes the current object (just like a regular function). In fact, if there are no parameters to be sent to the function, you can omit parameters to both methods altogether.

The object reference comes into play when the function being invoked is one that is normally defined as a method to a custom object. (Some of these concepts are covered later in this chapter, so you may need to return here after you are familiar with custom objects.)

Example

Consider the following code that generates a custom object and assigns a method to the object to display an alert about properties of the object:

```
// function to be invoked as a method from a 'car' object
function showCar()
{
   alert(this.make + " : " + this.color);
}
// 'car' object constructor function
function Car(make, color)
{
   this.make = make;
   this.color = color;
   this.show = showCar;
}
// create instance of a 'car' object
var myCar = new Car("Ford", "blue");
```

The normal way of getting the `myCar` object to display an alert about its properties is:

```
myCar.show();
```

At that point, the `showCar()` function runs, picking up the current `Car` object as the context for the `this` references in the function. In other words, when the `showCar()` function runs as a method of the object, the function treats the object as the *current object*.

With the `call()` or `apply()` methods, however, you don't have to bind the `showCar()` function to the `myCar` object. You can omit the statement in the `Car()` constructor that assigns the `showCar` function to a method name for the object. Instead, a script can invoke the `showCar()` method and instruct it to treat `myCar` as the current object:

```
showCar.call(myCar);
```

The `showCar()` function operates just as before, and the object reference in the `call()` method's first parameter slot is treated as the current object for the `showCar()` function.

As for succeeding parameters, the `apply()` method's second parameter is an array of values to be passed as parameters to the current function. The order of the values must match the order of parameter variables defined for the function. The `call()` method, on the other hand, enables you to pass individual parameters in a comma-delimited list. Your choice depends on how the parameters are carried along in your script. If they're already in array form, use the `apply()` method; otherwise, use the `call()` method. The (ECMA) recommended way to invoke a function through this mechanism when no parameters need to be passed is via the `call()` method.

Note

Remember, ECMA is the standards organization that oversees the official JavaScript language standard, which is formally known as ECMAScript. ECMAScript is the language specification, whereas JavaScript is an actual implementation. ■

toString()
valueOf()

Returns: String

Compatibility: WinIE4+, MacIE4+, NN4+, Moz+, Safari+

Scripts rarely, if ever, summon the `toString()` and `valueOf()` methods of a `Function` object. These methods work internally to allow debugging scripts to display a string version of the function definition. For example, when you enter the name of a function defined in The Evaluator (see Chapter 4) into the top text box, JavaScript automatically converts the function to a string so that its value can be displayed in the Results box. Using these methods or parsing the text they return has little, if any, practical application.

Function Application Notes

Understanding the ins and outs of JavaScript functions is the key to successful scripting, especially for complex applications. Additional topics covered in this section include the ways to invoke functions, variable scope in and around functions, recursion, and the design of reusable functions.

Invoking functions

A function doesn't perform any work until a script calls it by name or reference. Scripts invoke functions (that is, get functions to do something) through four routes: document object event

handlers; JavaScript statements; `href` attributes pointing to a `javascript:` URL; and the more modern `call()` and `apply()` methods of the `Function` object. The one approach not yet discussed at length in this book is the `javascript:` URL (some say pseudo-URL).

Several HTML tags have `href` attributes that normally point to Internet URLs for navigating to another page or loading a MIME file that requires a helper application or plug-in. These HTML tags are usually tags for clickable objects, such as links and client-side image map areas.

A JavaScript-enabled browser has a special built-in URL pseudo-protocol — `javascript:` — that lets the `href` attribute point to a JavaScript function or method rather than to a URL on the Net. For example, it is common practice to use the `javascript:` URL to change the contents of two frames from a single link. Because the `href` attribute is designed to point to only a single URL, you'd be out of luck if it weren't for the convenient way that JavaScript gives you access to multiframe navigation. You implement multiframe navigation by writing a function that sets the `location.href` properties of the two frames; then invoke that function from the `href` attribute. The following example shows what the script may look like:

```
function loadPages()
{
   parent.frames[1].location.href = "page2.html";
   parent.frames[2].location.href = "instrux2.html";
}
// ...
<a href="javascript:loadPages()">Next</a>
```

These kinds of function invocations can include parameters, and the functions can do anything you want. One potential side effect to watch out for occurs when the function returns a value (perhaps the function is also invoked from other script locations where a returned value is expected). Because the `href` attribute sets the `target` window to whatever the attribute evaluates to, the returned value is assigned to the `target` window — probably not what you want.

To prevent the assignment of a returned value to the `href` attribute, prefix the function call with the `void` operator:

```
<a href="javascript:void loadPages()">
```

If you don't want the `href` attribute to do anything (that is, you want to let the `onclick` event handler do all the work), assign a blank function after the operator:

```
<a href="javascript:void (0)">
```

Experienced programmers of many other languages recognize this operator as a way of indicating that no values are returned from a function or procedure. The operator has that precise functionality here, but in a nontraditional location.

Variable scope: Globals and locals

A variable can have two scopes in JavaScript. As you might expect, any variable initialized within the main flow of a script (not inside a function) is a *global variable*, in that any statement in the same document's script can access it by name. You can, however, also initialize variables inside a function (in a `var` statement) so the variable name applies only to statements inside that function; the scope is local to the function, so they are *local variables*. By limiting the scope of the variable to a single function, you can reuse the same variable name in multiple functions, thereby enabling the variables to carry

very different information in each function. Keep in mind that when you declare a variable inside of a function without the `var` keyword, you are creating a global variable and not a local variable. Listing 23-2 demonstrates the various possibilities.

LISTING 23-2

Variable Scope Workbench Page

HTML: jsb-23-02.html

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Variable scope</title>
      <script type="text/javascript" src="../jsb-global.js"></script>
      <script type="text/javascript" src="jsb-23-02.js"></script>
   </head>
   <body>
      <h1>Variable scope</h1>
      <h3>Charlie Brown has a global dog (Snoopy), a local dog (Gromit),
          a global toy (Gumby), and a local toy (Pokey).</h3>
      <div id="placeHolder"></div>
   </body>
</html>
```

JavaScript: jsb-23-02.js

```
// initialize when the page has loaded
addEvent(window, "load", testValues);

var newElem;
var newText;

var toyGlobal = "Gumby";
var aBoy = "Charlie Brown";
var hisDog = "Snoopy";

function showLocal()
{
   var toyLocal = "Pokey";
   return toyLocal;
}

function showGlobal()
{
   newElem = document.createElement("div");
   newElem.className = "objProperty";
   newText = document.createTextNode("Global version of hisDog is intact: "
             + hisDog);
      // just picked up a global variable value instead
      // of the local variable in the calling function
```

continued

LISTING 23-2 *(continued)*

```
    newElem.appendChild(newText);
    placeHolderElement.appendChild(newElem);
}

function testValues()
{
    // Dangerous ground here -- declaring a global variable in a function
    placeHolderElement = document.getElementById("placeHolder");

    // Now declare the local variable
    var hisDog = "Gromit";  // initializes local version of "hisDog"

        if (placeHolderElement)
        {
            newElem = document.createElement("div");
            newElem.className = "objProperty";
            newText = document.createTextNode("aBoy is: " + aBoy);
                // just picked up a global variable value
            newElem.appendChild(newText);
            placeHolderElement.appendChild(newElem);

            newElem = document.createElement("div");
            newElem.className = "objProperty";
            newText = document.createTextNode("His toyGlobal is " + toyGlobal);
            newElem.appendChild(newText);
                // just picked up another global variable value
            placeHolderElement.appendChild(newElem);

            // Do not bother with toyLocal here because it will throw undefined

            newElem = document.createElement("div");
            newElem.className = "objProperty";
            newText = document.createTextNode(
                      "toyLocal value returned from the showLocal function is: "
                      + showLocal() );
            newElem.appendChild(newText);
            placeHolderElement.appendChild(newElem);

            newElem = document.createElement("div");
            newElem.className = "objProperty";
            newText = document.createTextNode("Local version of hisDog is: " + hisDog);
            newElem.appendChild(newText);
                // just picked up another global variable value
            placeHolderElement.appendChild(newElem);

            // now call another function that does not set the variable hisDog
            // and display the value. we'll see that the global value is intact
            showGlobal();

        }
}
```

In this page, you define a number of variables — some global, others local — that are spread out in the JavaScript file. When you load this page, it runs the testValues() function, which accounts for the current values of all the variable names. The script then follows up with at least one value extraction that was masked in the function. The results of the JavaScript look like this:

```
aBoy is: Charlie Brown
His toyGlobal is Gumby
toyLocal value returned from the showLocal function is: Pokey
Local version of hisDog is: Gromit
Global version of hisDog is intact: Snoopy
```

Examine the variable initialization throughout the JavaScript file. In the beginning of the script there are several lines of code that are not contained within a function; this is the immediate execution section. In that section you define the first variable (toyGlobal) as a global style outside of any function definition. The var keyword for the global variable is optional but often helpful for enabling you to see at a glance where you initialize your variables. You also define two more global variables: aBoy and hisDog.

You then create two short functions. The showLocal() function defines a variable (toyLocal) that only statements in it can use. The showGlobal() function defines a function that simply displays the value of a variable (more on which variable in just a bit).

There is also a testValues() function that is the onload event handler. The first thing you'll notice is a variable (placeHolderElement) declared without the var keyword. Since placeHolderElement is declared within a function, it is a global variable. The testValues() function calls the showGlobal() function that uses the placeHolderElement variable. Why wasn't this variable declaration made in the immediate execution section of the script? The placeHolderElement variable is a reference to one of the elements on the HTML page that uses these functions. The placeHolderElement variable cannot successfully reference the element until the page is fully loaded. The only way to guarantee that the page is loaded before you declare the variable is to place the variable declaration within the onload event handler. Since the testValues() function is the onload event handler, the only way to make this variable available to other functions is to declare it here globally.

Also inside the testValues() function (for purposes of demonstration), you reuse the hisDog variable name. By initializing hisDog with the var statement inside the function, you tell JavaScript to create a separate local variable whose scope is only within the function. This initialization does not disturb the global variable of the same name. It can, however, make things confusing for you as the script author.

Statements in the testValues() function attempt to collect the values of variables scattered around the JavaScript file. Even from within this function, JavaScript has no problem extracting global variables directly, including the one defined in the immediate execution section. But JavaScript cannot get the local variable defined in the showLocal() function; that toyLocal variable is private to its own function. If you try to reference the showLocal() function's variable value from within the showGlobal() function, you will get an error message saying that the variable name is not defined. In the eyes of everyone else outside of the showLocal() function, that's true. If you really need that value, you can have the function return the value to a calling statement, as you do in the testValues() function.

At the beginning of the testValues() function, the script reads the aBoy global value without a hitch. But because you initialized a separate version of hisDog inside that function, only the localized version is available to the testValues() function. If you reassign a global variable name inside a function, you cannot access the global version from inside that function.

As proof that the global variable — whose name was reused inside the `testValues()` function — remains untouched, the `testValues()` function calls the `showGlobal()` function. The `showGlobal()` function displays the value of the `hisDog` variable. Since we're no longer inside the `testValues()` function, we see the value of the global variable `hisDog`. Charlie Brown and his dog, Snoopy, are united.

A benefit of this variable-scoping scheme is that you can reuse throwaway variable names in any function you like. For example, you can use the `i` loop counting variable in every function that employs loops. (In fact, you can reuse it in multiple `for` loops of the same function because the `for` loop reinitializes the value at the start of the loop.) If you pass parameters to a function, you can assign the same names to those parameter variables to aid in consistency. For example, a common practice is to pass an entire form object reference as a parameter to a function (using a `this.form` parameter in the event handler). For every function that catches one of these objects, you can use the variable name `form` in the parameter:

```
function doSomething(form)
{
   // statements
}
// ...
<input type="button" value="Do Something" onclick="doSomething(this.form)">
```

If five buttons on your page pass their form objects as parameters to five different functions, each function can assign `form` (or whatever you want to use) to that parameter value.

We recommend reusing variable names only for these throwaway variables. In this case, the variables are all local to functions, so the possibility of a mix-up with global variables does not exist. But the thought of reusing a global variable name as, say, a special case inside a function sends shivers up our spines. Such a tactic is doomed to cause confusion and error.

Caution

Reusing a global variable name locally is one of the most subtle and therefore difficult bugs to find in JavaScript code. The local variable ends up temporarily hiding the global variable without making any effort to let you know. Just do yourself a favor and make sure you don't reuse a global variable name as a local variable in a function. In the same vein, declaring a global variable within a function may also create bugs that are difficult to find in JavaScript code. ■

Some programmers devise naming conventions to avoid reusing global variables as local variables. A popular scheme puts a lowercase `g` in front of any global variable name. In the example from Listing 23-2, you could have named the global variables:

```
gToyGlobal
gABoy
gHisDog
gPlaceHolderElement
```

Then, if you define local variables, don't use the leading `g`. A similar scheme involves using an underscore character (_) global instead of a `g` in front of global variable names. In the example from Listing 23-2, you could have named the `local` variables:

```
_ToyLocal
_HisDog
```

Any scheme you employ to prevent the reuse of variable names in different scopes is fine, as long as it does the job. The same can be said of any scheme that clearly identifies global variables.

In a multiframe or multiwindow environment, your scripts can also access global variables from any other document currently loaded into the browser. For details about this level of access, see Chapter 27, "Window and Frame Objects."

Variable scoping rules apply equally to nested functions. Any variables defined in an outer function (including parameter variables) are exposed to all functions nested inside. But if you define a new local variable inside a nested function, that variable is not available to the outer function. Instead, you can return a value from the nested function to the statement in the outer function that invokes the nested function.

Parameter variables

When a function receives data in the form of parameters, remember that the values may be copies of the data (in the case of run-of-the-mill data values) or references to real objects (such as a form object). In the latter case, you can change the object's modifiable properties in the function when the function receives the object as a parameter, as shown in the following example:

```
function validateCountry(form)
{
   if (form.country.value == "")
   {
      form.country.value = "USA";
   }
}
```

Therefore, whenever you pass an object reference as a function parameter, be aware that the changes you make to an object you've passed into a function affect the real object.

As a matter of style, if our function needs to extract properties, or results of methods, from passed data (such as object properties or string substrings), we like to do that at the start of the function. We initialize as many variables as needed for each piece of data used later in the function. This task enables us to assign meaningful names to the data chunks, rather than rely on potentially long references within the working part of the function (such as using a variable like `inputStr` instead of `form.entry.value`). Today's machines are so incredibly fast that we don't really need to do this for machine speed; we're doing it for human speed — the code is now more readable and maintainable. Here's a quick example:

```
function updateContactInfo(form)
{
   var firstName = form.firstname.value;
   // or the preferred way:
   var lastName = document.getElementById("lastname").value;
   var address1 = document.getElementById("addr1").value;
   var address2 = document.getElementById("addr2").value;
   var phone = document.getElementById("phone").value;
   var email = document.getElementById("email").value;

   // Process contact info using local variables
}
```

Notice in this example how the form field information is first stored in local variables, which are then used to carry out the hypothetical updating of contact information. Throughout the remainder of the function you can use the local variables instead of the longer and less wieldy form fields.

Recursion in functions

In what may come as a strange surprise, it is possible for functions to call themselves — a process known as *recursion*. The classic example of programmed recursion is the calculation of the factorial (the factorial for a value of 4 is `4 * 3 * 2 * 1`), shown in Listing 23-3.

In the third line of this function, the statement calls itself, passing along a parameter of the next lower value of `n`. As this function executes, diving ever deeper into itself, JavaScript watches intermediate values and performs the final evaluations of the nested expressions. If designed properly, a recursive function eventually stops calling itself, and the program flow eventually returns back to the original function call.

Recursive functions are dangerous in the sense that they can easily fall into an infinite state (popularly called an *infinite loop*). For this reason, it is very important that you test them carefully. In particular, make sure that the recursion is finite: that a limit exists for the number of times it can recurse. In the case of Listing 23-3, that limit is the initial value of `n`. Failure to watch out for this limit may cause the recursion to overpower the limits of the browser's memory and even lead to a crash.

LISTING 23-3

A JavaScript Function Utilizing Recursion

```
function factorial(n)
{
   if (n > 0)
   {
      return n * (factorial(n-1));
   } else
   {
      return 1;
   }
}
```

Turning functions into libraries

As you start writing functions for your scripts, be on the lookout for ways to make functions generalizable (written so that you can reuse the function in other instances, regardless of the object structure of the page). The likeliest candidates for this kind of treatment are functions that perform specific kinds of validation checks (see examples in Chapter 46, "Data-Entry Validation," on the CD-ROM), data conversions, or iterative math problems.

To make a function generalizable, don't let it make any references to specific objects by name. Object names generally change from document to document. Instead, write the function so that it accepts a named object as a parameter. For example, if you write a function that accepts a `text` object as its parameter, the function can extract the object's data or invoke its methods without knowing anything about its enclosing form or name. Look again, for example, at the `factorial()` function, but now as part of an entire document in Listing 23-4.

LISTING 23-4

Calling a Generalizable Function

HTML: jsb-23-04.html

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Variable Scope Trials</title>
      <script type="text/javascript" src="../jsb-global.js"></script>
      <script type="text/javascript" src="jsb-23-04.js"></script>
   </head>
   <body>
      <h1>Variable Scope Trials</h1>
      <form id="theForm" action="calc-factorial.php" method="get">
         <p>
            <label for="visitorInput">Enter an input value:</label>
            <input type="text" id="input" name="input" value="0">

            <input type="submit" value="Calc Factorial">

            <label for="output">Results:</label>
            <input type="text" id="output" value="">
         </p>
      </form>
   </body>
</html>
```

JavaScript: jsb-23-04.js

```
// initialize when the page has loaded
addEvent(window, "load", initialize);

function initialize()
{
   // intercept form submission
   addEvent(document.getElementById("theForm"), "submit", calcFactorial);
}

function calcFactorial(evt)
{
   // consolidate event handling
   if (!evt) evt = window.event;

   // plug transformed input to output field
   var oInput = document.getElementById("input");
   var oOutput = document.getElementById("output");
```

continued

LISTING 23-4 *(continued)*

```
        // if they both exist
        if (oInput && oOutput)
        {
           oOutput.value = factorial(oInput.value);
        }

     // cancel form submission
        // W3C DOM method (hide from IE)
        if (evt.preventDefault) evt.preventDefault();
        // IE method
        return false;
}

function factorial(n)
{
        if (n > 0)
        {
           return n * (factorial(n - 1));
        }
        else
        {
           return 1;
        }
}
```

This function is designed to be generalizable, accepting only the input value (`n`) as a parameter. In the form, the `onsubmit` event handler sends only the input value from one of the form's fields to the `factorial()` function. The returned value is assigned to the output field of the form. The `factorial()` function is totally ignorant about forms, fields, or buttons in this document. If you need this function in another script, you can copy and paste it into that script knowing that it has been pre-tested. Any generalizable function may be part of your personal library of scripts — from which you can borrow — and save time in future scripting tasks.

You cannot always generalize a function. Somewhere along the line in your scripts, you will have to have references to JavaScript or custom objects. But if you find that you're frequently writing functions that perform the same kind of actions, see how you can generalize the code and put the results in your library of ready-made functions. You should also consider placing these reusable library functions in an external `.js` library file, as you've seen us do in this chapter and in others. You can also see Chapter 4 for details on this convenient way to share utility functions among many documents.

Making sense of closures

A topic that has confused many an aspiring scripter is closures, which may enter the picture when you declare a function within another function. At the core of JavaScript, *closures* refer to the fact that you can keep a local variable defined in a function alive even after the function has executed — which normally signals the end of life for a local variable. That's right, it's possible for a function to return

and its local variables to go on living like some kind of strange zombie data. How is this possible? Take a look at an example:

```
function countMe()
{
  var count = 1;
  var showCount = function() { alert(count); }
  count++;
  return showCount;
}
```

The unusual thing to note about this code is how the inner function assigned to the `showCount` variable is returned by the function. When you call the `countMe()` function, you receive a reference to the inner function that displays the `count` variable value. That wouldn't be a problem, except for the fact that the inner function acts on a variable (`count`) that is local to the `countMe()` function.

To see the closure come to life, take a look at this code that calls the `countMe()` function:

```
var countamatic = countMe();
countamatic();
```

The first line calls the `countMe()` function, which results in the local `count` variable being created, and a reference to the inner function being passed out and stored in the `countamatic` variable. The local variable `count` is also incremented within the `countMe()` function. Without knowing about closures, you would clearly be in some serious gray area at this point because the `countamatic()` function defined in the first line is now set to display the value of a variable that is clearly out of scope. But JavaScript works a miracle by keeping the `count` variable alive in a closure and still allowing the `countamatic()` function to access it. Thus, the second statement displays an alert with the number 2 in it.

If closures still seem a bit mysterious, just remember that a potential closure is created any time you specify a function within another function. You really only take advantage of a closure when you pass an inner function reference outside the scope of the function in which it is defined.

Okay, so closures reveal a sneaky way to manipulate scope in JavaScript, but what good are they? Closures are gaining a great deal of usage in Ajax applications because Ajax programming often uses them to work around inherent limitations in how you normally use the `this` keyword.

Cross-Reference

For more on Ajax, see Chapter 39, "Ajax, E4X, and XML." ■

Although we could just pawn everything closure-related off on Ajax, we can demonstrate a simple but practical application of how closures can help you carry out the seemingly impossible. Consider the scenario where you want to set a timer that calls a function after an interval of time has elapsed. You may be thinking: No problem — just create a function and pass its reference to the `setTimeout()` function. End of story. What we didn't mention is that you need to pass a couple of parameters to the function. See the problem?

Unless you design with anonymous functions, there is no mechanism for using parameters when you pass a function reference to another function, as in specifying a timer event handler when

calling the `setTimeout()` function. Now take a look at this code that uses closures to circumvent the problem:

```
function wakeupCaller(name, roomnum)
{
   return
   (
      function()
      {
         alert("Call " + name + " in room #" + roomnum + ".");
      }
   );
}
```

By placing a zero-param function within a parameterized function, you now have the ingredients for a timer handler that can accept parameters. The next step is to create an actual function reference that uses the closure:

```
var wakeWilson = wakeupCaller("Mr. Wilson", 515);
```

At this point, you've passed parameters to a function and received a zero-param function reference in return, which can then be passed along to the `setTimeout()` function:

```
setTimeout(wakeWilson, 600000);
```

Thanks to closures, Mr. Wilson will now get his wake up call!

Caution

Before you dive into closures and begin exploiting them in all of your code, let us caution you that they can result in some tricky bugs when used incorrectly. Extensive use of closures involving references to DOM objects can also cause memory leaks (gradual increase of memory used by the browser) if the objects are not disposed of (that is, set to `null`) when they are no longer needed. Definitely spend the time to explore closures in more detail before you get too wild with them. ■

Creating Your Own Objects with Object-Oriented JavaScript

In all the previous chapters of this book, you've seen how conveniently the browser document object models organize all the information about the browser window and its document. More specifically, you learned how to use standard objects as a means of accessing different aspects of the browser window and document. What may not be obvious from the scripting you've done so far is that JavaScript enables you to create your own objects in memory — objects with properties and methods that you define. These objects are not user-interface elements on the page, but rather the kinds of objects that may contain data and script functions (behaving as methods) whose results the user can see displayed in the browser window.

You actually had a preview of this power in the discussion about arrays (Chapter 18, "The Array Object"). An array, you recall, is an ordered collection of data. An object typically contains different kinds of data. It doesn't have to be an ordered collection of data — although your scripts can

use objects in constructions that strongly resemble arrays. Moreover, you can attach any number of custom functions as methods for that object. You are in total control of the object's structure, data, and behavior.

The practice of employing custom objects in your JavaScript code is known as *object-oriented programming*, or *OO programming* for short. OO has been around a long time and has been used to great success in other programming languages such as C++ and Java. However, the scripted nature of JavaScript has caused OO to catch on a bit more slowly in the JavaScript world. Even so, support for custom objects is a standard part of modern JavaScript-enabled browsers and is something you should consider taking advantage of whenever prudent.

Note

To split hairs, technically, we have to clarify that JavaScript isn't truly an object-oriented language in a strict sense. Instead, JavaScript is considered an *object-based* language. The difference between object-oriented and object-based is significant and has to do with how objects can be extended. Even so, conceptually, JavaScript's support of objects is enough akin to true OO languages that it's not unreasonable to discuss JavaScript in OO terms. You learn about some of the specific object features that allow JavaScript to approach OO languages later in the chapter. What's of further interest is that ECMAScript now supports the notions of classes, interfaces, etc., in terms that are closer to the traditional OO manner, which means that the next release(s) of JavaScript will start to support these notions as well. ■

There is no magic to knowing when to use a custom object instead of an array in your application. The more you work with and understand the way custom objects work, the more likely you will think about your data-carrying scripts in these terms — especially if an object can benefit from having one or more methods associated with it. This avenue is certainly not one for beginners, but we recommend that you give custom objects more than a casual perusal after you have gained some JavaScripting experience.

The nuts and bolts of objects

An *object* in JavaScript is really just a collection of properties. *Properties* can take on the form of data types, functions (methods), or even other objects. In fact, you might find it easier to think of a custom object as an array of values, each of which is mapped to a property (a data type, method, or object). Wait — a method can be a property of an object? We'll see how in just a bit; for now, take it on faith. A function contained in an object is known as a *method*. Methods are no different than other functions, except that they are intended to be used in the context of an object, and therefore are assumed to have access to properties of that object. This connection between properties and methods is one of the core concepts prevalent in OO.

Objects are created using a special function known as a *constructor*, which determines the name of the object — the constructor is given the same name as the object. Here's an example of a constructor function:

```
function Alien()
{
}
```

Although this function doesn't contain any code, it does nonetheless lay the groundwork for creating an `Alien` object. You can think of a constructor as a blueprint that is then used to create actual objects. A word about naming constructor functions: Conventionally, the constructor name is in

TitleCase, and each instance variable referencing the object is in camelCase. Here's an example of how you create an object using a constructor:

```
var myAlien = new Alien();
```

The `new` keyword is used, in conjunction with a constructor, to create JavaScript objects.

Creating properties for custom objects

Earlier we mentioned that properties are key to objects, so you might be wondering how you go about creating a property for a custom object. Custom object properties are created in the constructor with some help from the `this` keyword, as the following code reveals:

```
function Alien()
{
   this.name = "Clyde";
   this.aggressive = true;
}
```

The `this` keyword is used to reference the current object, which in this case is the object that is being created by the constructor. So, you use the `this` keyword to identify new properties for the object. The only problem with this example is that it results in all aliens that are created with the `Alien()` constructor having the same name and aggression. The fix is to pass in the property values to the constructor so that each alien can be customized upon creation:

```
function Alien(name, aggressive)
{
   this.name = name;
   this.aggressive = aggressive;
}
```

Now you can create different aliens that each have their own unique property values:

```
var alien1 = new Alien("Ernest", false);
var alien2 = new Alien("Wilhelm", true);
```

To get to the properties of the object (for reading or writing after the object has been created), use the same type of dot syntax you use with DOM objects. To change the `name` property of one of the objects, the statement would be:

```
alien1.name = "Julius";
```

Creating methods for custom objects

Properties are really only half of the JavaScript OO equation. The other half is methods, which are functions that you tie to objects so that they can access object properties. Following is an example of a method that you might use with the `alien` class:

```
function attack()
{
   if (this.aggressive)
   {
      // Do some attacking and return true to indicate that the attack
      // commenced
```

```
      return true;
   }
   else
   {
      // Don't attack and return false to indicate that the attack didn't
      // happen
      return false;
   }
}
```

Notice that the `attack()` method references the `this.aggressive` property to decide if the attack should take place. The only thing missing at this point is the connection between the `attack()` method and the `alien` object. Without this connection the `this` keyword would have no meaning because there would be no associated object. Here's how the connection is made:

```
function Alien(name, aggressive)
{
   // properties
   this.name = name;
   this.aggressive = aggressive;
   // methods
   this.attack = attack;
}
```

This code clearly illustrates how methods are really just properties: they are declared the same as properties, so they masquerade as properties. A new property named `attack` is created and assigned a reference to the `attack()` method. It's very important to note that the `attack()` method is specified by reference (without the parentheses). Here, then, is the creation of an `alien` object and the invocation of its `attack()` method:

```
var alien1 = new Alien("Ernest", false);
alien1.attack();
```

The way we have defined the attack method above for the alien object places it in the global namespace, which is not a good thing in terms of OO best practices (as you'll see later in this chapter when we discuss encapsulation). A better approach is to embed it in the constructor:

```
function Alien(name, aggressive)
{
   // properties
   this.name = name;
   this.aggressive = aggressive;
   // methods
   this.attack = function(name, aggressive)
   {
      if (this.aggressive)
      {
         // Do some attacking and return true to indicate that
         // the attack commenced
         return true;
      }
      else
      {
```

```
                // Don't attack and return false to indicate that
               // the attack didn't happen
                return false;
            }
        };
    }
```

You're now armed with enough of the basics of JavaScript objects to move on to a more complete example, which the next section provides. This example will be progressively improved over the rest of the sections in this chapter as you build your custom object skills in a step-wise fashion.

An OO example – planetary objects

Building on your familiarity with the planetary data array created in Chapter 18, "The Array Object," this chapter shows you how convenient it is to use the data when it is constructed in the form of an OO design that utilizes custom objects. The application goal for the extended example in this section is to present a drop-down list of the nine planets of the solar system and display the properties of the selected planet.

In this chapter, instead of building arrays to hold the data, you build objects — one object for each planet. The design of your planetary object has five properties and one method. The properties of each planet are name, diameter, distance from the sun, year length, and day length. To assign more intelligence to these objects, in Listing 23-5 you give each of them the capability to display their data below the drop-down list. You can conveniently define one function that knows how to behave with any of these planet objects, rather than having to define nine separate functions. When used within the context of an object, a function is actually referred to as a *method*.

LISTING 23-5

OO Planetary Data Presentation

HTML: jsb-23-05.html

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>OO Planetary Data Presentation</title>
      <script type="text/javascript" src="../jsb-global.js"></script>
      <script type="text/javascript" src="jsb-23-05.js"></script>
   </head>
   <body>
      <!-- Display the Planet choices to the visitor -->
      <h1>The Daily Planet</h1>
      <form>
         Select a planet to view its planetary data:
         <select name="planetsList" onchange="doDisplay(this)">
            <option>Mercury</option>
            <option>Venus</option>
            <option selected="selected">Earth</option>
            <option>Mars</option>
```

```
            <option>Jupiter</option>
            <option>Saturn</option>
            <option>Uranus</option>
            <option>Neptune</option>
            <option>Pluto</option>
         </select>
      </form>
   </body>
</html>
```

JavaScript: jsb-23-05.js

```
// definition of Planet object constructor
function Planet(name, diameter, distance, year, day)
{
   // properties
   this.name = name;
   this.diameter = diameter;
   this.distance = distance;
   this.year = year;
   this.day = day;
   // methods
   this.showPlanet = function()
   {
      var result = "<html><body><center><table border='2' cellpadding ='2'>";
      result += "<caption align='top'>Planetary data for <b>"
             + this.name + "</b></caption>";
      result += "<tr><td align='right'>Diameter</td><td>"
             + this.diameter + "</td></tr>";
      result += "<tr><td align='right'>Distance from Sun</td><td>"
             + this.distance + "</td></tr>";
      result += "<tr><td align='right'>One Orbit Around Sun</td><td>"
             + this.year + "</td></tr>";
      result += "<tr><td align='right'>One Revolution (Earth Time)</td><td>"
             + this.day + "</td></tr>";
      result += "</table></center></body></html>";
      // display results
      document.write(result);
      document.close();
   };
}

// create new Planet objects, and store in a series of reference variables
// 'new' will create a new Planet instance and insert parameter data into object
var mercury = new Planet("Mercury","3100 miles", "36 million miles",
                         "88 days", "59 days");
var venus = new Planet("Venus", "7700 miles", "67 million miles",
                       "225 days", "244 days");
var earth = new Planet("Earth", "7920 miles", "93 million miles",
                       "365.25 days","24 hours");
var mars = new Planet("Mars", "4200 miles", "141 million miles",
                      "687 days", "24 hours, 24 minutes");
```

continued

LISTING 23-5 *(continued)*

```
var jupiter = new Planet("Jupiter","88,640 miles","483 million miles",
                         "11.9 years", "9 hours, 50 minutes");
var saturn = new Planet("Saturn", "74,500 miles","886 million miles",
                        "29.5 years", "10 hours, 39 minutes");
var uranus = new Planet("Uranus", "32,000 miles","1.782 billion miles",
                        "84 years", "23 hours");
var neptune = new Planet("Neptune","31,000 miles","2.793 billion miles",
                         "165 years", "15 hours, 48 minutes");
var pluto = new Planet("Pluto", "1500 miles", "3.67 billion miles",
                       "248 years", "6 days, 7 hours");

// invoke Planet object method corresponding to the
// index of the visitor's choice
function doDisplay(popup)
{
   i = popup.selectedIndex;
   eval(popup.options[i].text.toLowerCase() + ".showPlanet()");
}
```

The first task is to define the object constructor function, which performs several important jobs. For one, everything in this function establishes the structure of your custom object: the properties available for data storage and retrieval, and any methods that the object can invoke. The name of the function is the name you use later to create new instances of the object. Therefore, choosing a name that truly reflects the nature of the object is important. And, because you probably want to stuff some data into the function's properties to get one or more instances of the object loaded and ready for the page's user, the function definition includes parameters for each of the properties defined in this object definition.

Inside the object, you use the `this` keyword to assign data that comes in as parameters to labeled properties. For this example, we use the same names for both the incoming parameter variables and the properties. That's primarily for convenience (and is very common practice), but you can assign any parameter and property names you want and connect them any way you like. In the `Planet()` constructor function, five property slots are reserved for every instance of the object, whether or not any data actually is placed in every property (any unassigned slot has a value of `null`).

The last entry in the `Planet()` constructor function is the definition of the `showPlanet()` method. As you do in every JavaScript method you encounter, you must invoke a method by using a reference to the object, a period, and the method name, followed by a set of parentheses. You see that syntax in a minute.

The next long block of statements creates the individual objects according to the definition established in the `Planet()` constructor. Just like in an array, an assignment statement and the keyword `new` create an object. We assign names that are the real names of planets (the `mercury` object reference name refers to the Mercury planet object).

The act of creating a new object sets aside space in memory (associated with the current document) for this object and its properties. An object created in memory is known as an *instance*. In this script, you create nine object instances, each with a different set of properties. Note that no parameter is sent

(or expected by the function) that corresponds to the `showPlanet()` method. Omitting that parameter here is fine because the specification of that method in the object definition means that the script automatically attaches the method to every version (instance) of the planet object that it creates. This is important, because it links the function (method) to the object, thereby providing it access to the object's properties.

The last function definition, `doDisplay()`, is an event handler, invoked whenever the user makes a choice from the list of planets. The first statement extracts the index value of the selected item. Using that index value, the script extracts the text. But things get a little tricky because you need to use that text string as a variable name — the name of the planet — so that you can append to it the call to the `showPlanet()` method. To make the disparate data types come together, use the `eval()` function (see Chapter 24 for a complete discussion). Inside the parentheses, extract the string for the planet name, change it to lowercase, and concatenate a string that completes the reference to the object's `showPlanet()` method. The `eval()` function evaluates that string, which turns it into a valid method call. Therefore, if the user selects jupiter from the drop-down list, the method call becomes `jupiter.showPlanet()`.

Now it's time to look back to the `showPlanet()` method definition. When that method runs in response to a user selection of the planet Jupiter, the method's only scope is of the `jupiter` object. Therefore, all references to `this.propertyName` in `showPlanet()` refer just to `jupiter`. The only possibility for `this.name` in the `jupiter` object is the value assigned to the `name` property for `jupiter`. The same goes for the rest of the properties extracted in the method.

Further encapsulation

One of the benefits of using objects in scripting is that all the "wiring" inside the object — properties and methods — are defined within the local variable scope of the object. It's a concept called *encapsulation*. Thus, when you use the `this` keyword to create object properties and assign values to them inside a constructor function, the property names, and any code associated with them, are private to the object and never conflict with global variable or object names.

In large scripting projects, especially when multiple programmers are involved, it's fairly easy for global names to clash. For example, in Internet Explorer, all DOM object IDs become part of the global variable space because that browser allows scripts to reference an element object simply by its ID (unlike the W3C DOM, which uses the `document.getElementById()` method). As a project grows in size and complexity, it is increasingly important to avoid piling up more and more objects — including function definitions — in the global variable name space.

In Listing 23-5, for example, the `showPlanet()` method is defined in the local name space. It's a local method to the object because the method definition is defined within the `planet()` constructor function. If we were to define the `showPlanet()` method outside of the `Planet()` constructor function (as shown in an earlier example in this chapter), the `showPlanet()` method would be taking up a global object name that might be used elsewhere.

Creating an array of objects

In Listing 23-5, each of the planet objects is assigned to a global variable whose name is that of the planet. If the idea of custom objects is new to you, this idea probably doesn't sound so bad, because it's easy to visualize each variable representing an object. But, as shown in the `doDisplay()` function, accessing an object by name requires use of the `eval()` function to convert a string representation to a valid object reference. Although it's not too important in this simple example, the `eval()` function is not particularly efficient in JavaScript. If you find yourself using an `eval()` function, look

for ways to improve efficiency, such that you can reference an object directly. The way to accomplish that streamlining for this application is to place the objects in an array.

To assign the custom objects in Listing 23-5 to an array, first create an empty array and then assign the result of each object constructor call as an entry in the array. The modified code section would result in Listing 23-6. This partial listing of the code from the external JavaScript file shows just the change in the code. The HTML for Listings 23-5 and 23-6 are exactly the same, so only the code from the partial external JavaScript file is included here. Both files are complete on the CD-ROM.

LISTING 23-6

Array of Planet Objects

JavaScript: partial of jsb-23-06.js

```
// create new Planet objects, and store in an array of reference variables
// 'new' will create a new Planet instance and insert parameter data into object
var planets = new Array
(
   new Planet("Mercury","3100 miles", "36 million miles",
              "88 days", "59 days"),
   new Planet("Venus", "7700 miles", "67 million miles",
              "225 days", "244 days"),
   new Planet("Earth", "7920 miles", "93 million miles",
              "365.25 days","24 hours"),
   new Planet("Mars", "4200 miles", "141 million miles",
              "687 days", "24 hours, 24 minutes"),
   new Planet("Jupiter","88,640 miles","483 million miles",
              "11.9 years", "9 hours, 50 minutes"),
   new Planet("Saturn", "74,500 miles","886 million miles",
              "29.5 years", "10 hours, 39 minutes"),
   new Planet("Uranus", "32,000 miles","1.782 billion miles",
              "84 years", "23 hours"),
   new Planet("Neptune","31,000 miles","2.793 billion miles",
              "165 years", "15 hours, 48 minutes"),
   new Planet("Pluto", "1500 miles", "3.67 billion miles",
              "248 years", "6 days, 7 hours")
);
}
```

The supreme advantage to this approach comes in a modified `doDisplay()` function, which can use the index value from the `select` element's `selectedIndex` directly:

```
function doDisplay(popup)
{
   i = popup.selectedIndex;
   planets[i].showPlanet();
}
```

The presence of so many similar objects cries out for their storage as an array. Here, we are using the numeric value of the drop-down list's `selectedIndex` to locate the proper array element.

Taking advantage of embedded objects

One powerful technique for using custom objects is embedding an object within another object. When you place one object inside another object, the object being placed is known as an *embedded object* or *object property*. Let's extend the planet example to help you understand the implications of using a custom object property.

Say that you want to beef up the planet page with a photo of each planet. Each photo has a URL for the photo file; each photo also contains other information, such as the copyright notice and a reference number, which displays on the page for the user. One way to handle this additional information is to create a separate object definition for a photo database. Such a definition may look like this:

```
function Photo(name, URL, copyright, refNum)
{
   this.name = name;
   this.URL = URL;
   this.copyright = copyright;
   this.refNum = refNum;
}
```

You then need to create individual `photo` objects for each picture. One such definition may look like this:

```
mercuryPhoto = new Photo("Planet Mercury", "/images/merc44.gif",
                         "(c)1990 NASA", 28372);
```

Attaching a `photo` object to a `planet` object requires modifying the `planet` constructor function to accommodate one more property — an object property. The new `planet` constructor looks like this:

```
function Planet(name, diameter, distance, year, day, photo)
{
   this.name = name;
   this.diameter = diameter;
   this.distance = distance;
   this.year = year;
   this.day = day;
   this.showPlanet = showPlanet;
   this.photo = photo; // add photo property
}
```

When the `photo` objects are created, you can then create each `planet` object by passing one more parameter — a `photo` object you want associated with that object:

```
// create new planet objects, and store in a series of variables
Mercury = new Planet("Mercury", "3100 miles", "36 million miles",
                     "88 days", "59 days", mercuryPhoto);
```

To access a property of a `photo` object, your scripts then have to assemble a reference that works its way through the connection with the `planet` object:

```
copyrightData = Mercury.photo.copyright;
```

The potential of embedded objects of this type is enormous. For example, you can embed all the copy elements and image URLs for an online catalog in a single document. As the user selects items to

view (or cycles through them in sequence), a new JavaScript-written page displays the information in an instant. This requires only the image to be downloaded — unless the image was pre-cached, as described in the image object discussion in Chapter 29, “The Document and Body Objects.” In this case, everything works instantaneously — no waiting for page after page of catalog.

If, by now, you think you see a resemblance between this object-within-an-object construction and a relational database, give yourself a gold star. Nothing prevents multiple objects from having the same sub-object as their properties — like multiple business contacts having the same company object property.

The current way to create objects

For current browsers, you can also benefit from a shortcut literal syntax for creating a new object. You can set pairs of property names and their values inside a set of curly braces, and you can assign the whole construction to a variable that becomes the object name. The following script shows how to organize this kind of object constructor:

```
var Earth = {diameter:"7920 miles", distance:"93 million miles", year:"365.25",
             day:"24 hours", showPlanet:showPlanet};
```

Colons link name-value pairs, and commas separate multiple name-value pairs. The value portion of a name-value pair can even be an array (using the `[...]` constructor shortcut) or a nested object (using another pair of curly braces). In fact, you can nest arrays and objects to your heart’s content to create exceedingly complex objects. All in all, this is a very compact way to embed data in a page for script manipulation. If your CGI, XML, and database skills are up to the task, consider using a server program to convert XML data into this compact JavaScript version, with each XML record being its own JavaScript object. For multiple records, assign the curly-braced object definitions to an array entry. Then your scripts on the client can iterate through the data and generate HTML to display the data in a variety of forms, and sorted according to different criteria (thanks to the JavaScript array-sorting powers).

Defining object property getters and setters

The idea of creating getters and setters, generally known in JavaScript as *accessors*, for an object is introduced in Chapter 25, “Document Object Model Essentials,” where they are described in the context of W3C DOM objects. Here, we look at an example of using getters and setters in the context of custom objects.

The purpose of a setter is to assign a new value to a property in an object and at the same time check it for validity if necessary. A setter defines how a new value assigned to the property should apply the value to the object. We would define a getter for each property we wanted to read in the object. In our object, we would typically define a getter and setter for each property of the object. If you have a requirement for a read-only property, then you would define only a getter method.

As you can see in Listing 23-7, both getter and setter methods in the `TitleCaseName` object property are written in the form of methods, using the name of the corresponding property. Reading or writing an object’s property value can include sophisticated processing for either operation. This is called *object literal notation*. Firefox 2.0+, Safari 3.0+, Chrome+, and Opera 9.5+ support this. As of IE8, there is no support for getters and setters for custom objects. This partial listing of the code in the external JavaScript file shows just the change in the code for Listing 23-6. The HTML for

Listings 23-5 and 23-7 are exactly the same, so only the code from the partial external JavaScript file is included here. Both files are complete on the CD-ROM.

LISTING 23-7

Planet Object Constructor Using Getters and Setters

JavaScript: partial of jsb-23-07.js

```
// object property with accessors using object literal notation
// its property 'name', is a String value that is TitleCased when set
var TitleCaseName =
{
   _name: null,

   get name()
   {
     return this._name;
   },

   set name(value)
   {
     this._name = value.substring(0,1).toUpperCase() + value.substring(1);
   }
};

// definition of Planet object constructor
function Planet(name, diameter, distance, year, day)
{
   // properties
   TitleCaseName.name = name;
   this.name = TitleCaseName.name;
   this.diameter = diameter;
   this.distance = distance;
   this.year = year;
   this.day = day;
   // methods
   this.showPlanet = function()
   {
      // . . .
   };
}
```

In our continuing planet scenario, we want to make sure that when a `Planet` object is created, its `name` property is in TitleCase, regardless of how the developer programs the code. This is the perfect opportunity for a setter: we want to take the current value of the `name` property, convert its case, and then apply the new value to the `name` property.

The first thing in the listing is the definition of the getter and setter methods that will be used in the `TitleCaseName` object. You'll notice that the `function` keyword is not used. Once we've defined

our local properties, we define the getters and/or setters. To define each getter, we only need the `get` keyword, followed by the property name we want to read. Typically, we would return the value of the corresponding property. In our case, we're only defining a getter that simply reads the current value of the object's `_name` property and returns it. We use the underscore prefix in `_name` to indicate that this property is strictly internal and should not be referenced outside its containing object.

Similarly, we use the `set` keyword to define each setter. In this case, our setter converts the `value` parameter to TitleCase, and then assigns it to the `_name` property.

The final piece is in the `Planet` object's constructor function. Notice the first two properties in this function. In our case, the value supplied at object instantiation for the `name` property of a planet will "call" the `TitleCaseName` getter and setter function. In the first of these two lines, we assign the name to `TitleCaseName`. This invokes the setter, converting the value to `TitleCase`. In the second line, the getter is invoked, assigning the `TitleCased` name to `Planet` name.

Object-Oriented Concepts

As mentioned previously, JavaScript is object-based rather than object-oriented. This means that instead of adhering to the class, subclass, and inheritance schemes of object-oriented languages such as Java, JavaScript uses what is called *prototype inheritance*. This scheme works not only for native and DOM objects but also for custom objects.

Adding a prototype

A custom object is frequently defined by a constructor function, which typically parcels out initial values to properties of the object, as in the following example:

```
function Car(plate, model, color)
{
   this.plate = plate;
   this.model = model;
   this.color = color;
}
var car1 = new Car("AB 123", "Ford", "blue");
```

Current browsers offer a handy shortcut to stuff default values into properties if none are provided (the supplied value is `null`, `0`, or an empty string). The `OR` operator (`||`) can let the property assignment statement apply the passed value, if present, or a default value you hard-wire into the constructor. Therefore, you can modify the preceding function to offer default values for the properties:

```
function Car(plate, model, color)
{
   this.plate = plate || "missing";
   this.model = model || "unknown";
   this.color = color || "unknown";
}
var car1 = new Car("AB 123", "Ford", "");
```

After the preceding statements run, the `car1` object has the following properties:

```
car1.plate    // value = "AB 123"
car1.model    // value = "Ford"
car1.color    // value = "unknown"
```

You can then add a new property to the constructor's `prototype` property, as in

```
Car.prototype.companyOwned = true;
```

Any `Car` object you already created, or are about to create, automatically inherits the new `companyOwned` property and its value. You can still override the value of the `companyOwned` property for any individual `Car` object. But if you don't override the property for instances of the `Car` object, the `Car` objects whose `companyOwned` property is not overridden automatically inherit any change to the `prototype.companyOwned` value. This has to do with the way JavaScript looks for `prototype` property values. In a similar fashion, we use prototypes in Listing 23-8 in our ongoing improvements to our `Planet` object. This partial listing of the code in the external JavaScript file shows just the change in the code for Listing 23-7. The HTML for Listings 23-5 and 23-8 are exactly the same, so only the code from the partial external JavaScript file is included here. Both files are complete on the CD-ROM.

LISTING 23-8

Planet Object Constructor Using Prototype

JavaScript: partial of jsb-23-08.js

```
// definition of Planet object constructor
function Planet(name, diameter, distance, year, day)
{
   // properties
   TitleCaseName.name = name;
   this.name = TitleCaseName.name;
   this.diameter = diameter;
   this.distance = distance;
   this.year = year;
   this.day = day;
}

Planet.prototype =
{
   constructor : Planet,
   // methods
   showPlanet : function()
   {
      // . . .
   }
};
```

Prototype inheritance

Each time your script attempts to read or write a property of an object, JavaScript follows a specific sequence in search of a match for the property name. The sequence is as follows:

1. If the property has a local value assigned to the current object, this is the value to use.
2. If there is no local value, check the value of the property's prototype of the object's constructor.

3. Continue up the prototype chain until either a match of the property is found (with a value assigned to it) or the search reaches the native `Object` object.

Therefore, if you change the value of a constructor's `prototype` property and you do not override the property value in an instance of that constructor, JavaScript returns the current value of the constructor's `prototype` property.

Nested objects and prototype inheritance

When you begin nesting objects, especially when one object invokes the constructor of another, there is an added wrinkle to the prototype inheritance chain. Let's continue with the `Car` object defined earlier. In this scenario, consider the `Car` object to be akin to a root object that has properties shared among two other types of objects. One of the object types is a company fleet vehicle, which needs the properties of the root `Car` object (`plate`, `model`, `color`) but also adds some properties of its own. The other object that shares the `Car` object is an object representing a car parked in the company garage — an object that has additional properties regarding the parking of the vehicle. This explains why the `Car` object is defined on its own.

Now look at the constructor function for the parking record, along with the constructor for the basic `Car` object:

```
function Car(plate, model, color)
{
   this.plate = plate || "missing";
   this.model = model || "unknown";
   this.color = color || "unknown";
}
function CarInLot(plate, model, color, timeIn, spaceNum)
{
   this.timeIn = timeIn;
   this.spaceNum = spaceNum;
   this.carInfo = car;
   this.carInfo(plate, model, color);
}
```

The `CarInLot` constructor not only assigns values to its unique properties (`timeIn` and `spaceNum`) but it also includes a reference to the `Car` constructor, arbitrarily assigned to a property called `carInfo`. This property assignment is merely a conduit that allows property values intended for the `Car` constructor to be passed within the `CarInLot` constructor function. To create a `CarInLot` object, use a statement like the following:

```
var car1 = new CarInLot("AA 123", "Ford", "blue", "10:02AM", "31");
```

After this statement, the `car1` object has the following properties and values:

```
car1.timeIn      // value = "10:02AM"
car1.spaceNum    // value = "31"
car1.carInfo     // value = reference to car object constructor function
car1.plate       // value = "AA 123"
car1.model       // value = "Ford"
car1.color       // value = "blue"
```

Let's say that five `CarInLot` objects are created in the script (`car1` through `car5`). The prototype wrinkle comes into play if, for example, you assign a new property to the `Car` constructor prototype:

```
Car.prototype.companyOwned = true;
```

Even though the `CarInLot` objects use the `Car` constructor, the instances of `CarInLot` objects do not have a prototype chain back to the `Car` object. As the preceding code stands, even though you've added a `companyOwned` property to the `Car` constructor, no `CarInLot` object inherits that property (even if you were to create a new `CarInLot` object after defining the new `prototype` property for `Car`). To get the `CarInLot` instances to inherit the `prototype.companyOwned` property, you must explicitly connect the prototype of the `CarInLot` constructor to the `Car` constructor prior to creating instances of `CarInLot` objects:

```
CarInLot.prototype = new Car();
```

The complete sequence, then, is as follows:

```
function Car(plate, model, color)
{
   this.plate = plate || "missing";
   this.model = model || "unknown";
   this.color = color || "unknown";
}
function CarsInLot(plate, model, color, timeIn, spaceNum)
{
   this.timeIn = timeIn;
   this.spaceNum = spaceNum;
   this.carInfo = car;
   this.carInfo(plate, model, color);
}
CarsInLot.prototype = new Car();
var car1 = new CarsInLot("123ABC", "Ford","blue","10:02AM", "32");
Car.prototype.companyOwned = true;
```

After this stretch of code runs, the `car1` object has the following properties and values:

```
car1.timeIn            // value = "10:02AM"
car1.spaceNum          // value = "31"
car1.carInfo           // value = reference to car object constructor function
car1.plate             // value = "AA 123"
car1.model             // value = "Ford"
car1.color             // value = "blue"
car1.companyOwned      // value = true
```

NN4+/Moz provides one extra, and proprietary, bit of syntax in this prototype world. The `__proto__` property (that's with double underscores before and after the word *proto*) returns a reference to the object that is next up the prototype chain. For example, if you inspect the properties of `car1.__proto__` after the preceding code runs, you see that the properties of the object next up the prototype chain are as follows:

```
car1.__proto__.plate           // value = "AA 123"
car1.__proto__.model           // value = "Ford"
```

```
car1.__proto__.color          // value = "blue"
car1.__proto__.companyOwned   // value = true
```

This property can be helpful in debugging custom objects and prototype inheritance chain challenges, but the property is not part of the ECMA standard. Therefore, we discourage you from using the property in your production scripts, since it isn't available in non-Mozilla-based browsers.

Object Object

Properties	Methods
constructor	hasOwnProperty()
prototype	isPrototypeOf()
	propertyIsEnumerable()
	toSource()
	toString()
	unwatch()
	valueOf()
	watch()

Syntax

Creating an `Object` object:

```
function constructorName([arg1,...[,argN]])
{
    statement(s)
}
var objName = new constructorName(["argName1",...[,"argNameN"]);
var objName = new Object();
var objName = {propName1:propVal1[, propName2:propVal2[,...N]]}
```

Accessing an `Object` object properties and methods:

```
objectReference.property | method([parameters])
```

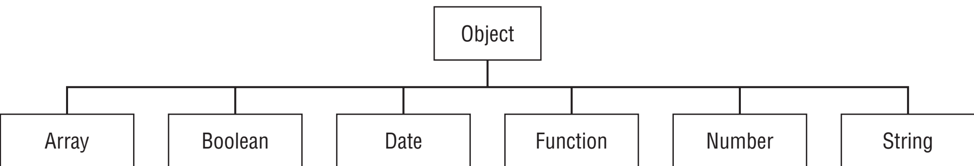
Compatibility: WinIE4+, MacIE4+, NN3+, Moz+, Safari+, Chrome+

About this object

Although it might sound redundant, the `Object` object is a vital native object in the JavaScript environment. It is the root object on which all other native objects — such as `Date`, `Array`, `String`, and the like — are based (see Figure 23-1). This object also provides the foundation for creating custom objects, as described earlier in this chapter.

FIGURE 23-1

The `Object` object hierarchy.



By and large, your scripts do not access the properties of the native `Object` object. The same is true for many of its methods, such as `toString()` and `valueOf()`, which internally allow debugging alert dialog boxes (and The Evaluator) to display something when referring to an object or its constructor. In the case of methods like `toString()` and `valueOf()`, you will typically override these in the child. Similarly, the `toSource()` method returns a string representation of the source code for an object, which is not something you would typically need to use in a script.

A little more practical are the `watch()` and `unwatch()` methods of the `Object` object, which provide a mechanism for taking action when a property value changes. You can call the `watch()` method and assign a function that is called when a certain property changes. This enables you to handle an internal property change as an event. When you're finished watching a property, the `unwatch()` method is used to stop the watchpoint.

Listing 23-9 contains an example of how the `watch()` and `unwatch()` methods can be used to track the value of a property.

LISTING 23-9

Watching an Object Property

HTML: jsb-23-09.html

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Object Watching</title>
      <script type="text/javascript" src="../jsb-global.js"></script>
      <script type="text/javascript" src="jsb-23-09.js"></script>
   </head>
   <body>
      <h1>Watching Over You</h1>
      <form id="theForm" action="validate.php" method="get">
         <div>
         <label for="theText">Enter text here: </label>
         <input id="theText" type="text" name="entry" size="50"
                value="Default Value">
         <p><input id="fourScoreButton" type="button" value="Set to Phrase I"
                   onclick="setIt('Four score and seven years ago...')"></p>
         <p><input id="whenInButton" type="button" value="Set to Phrase 2"
                   onclick="setIt('When in the course of human events...')"></p>
```

continued

LISTING 23-9 *(continued)*

```
          <p><input id="resetButton" type="reset"
                    onclick="setIt('Default value')"></p>
          <p><input id="watchButton" type="button" value="Watch It"
                    onclick="watchIt(true)">
             <input id="unWatchButton" type="button" value="Don't Watch It"
                    onclick="watchIt(false)"></p>
          </div>
       </form>
    </body>
</html>
```

JavaScript: jsb-23-09.js

```
function setIt(msg)
{
   document.forms[0].entry.value = msg;
}

function watchIt(on)
{
   var obj = document.forms[0].entry;
      if (on)
      {
         obj.watch("value",report);
      } else
      {
         obj.unwatch("value");
      }
}

function report(id, oldval, newval)
{
   alert("The field's " + id + " property on its way from \n'" +
         oldval + "'\n to \n'" + newval + "'.");
   return newval;
}
```

You can use a trio of methods (`hasOwnProperty()`, `isPrototypeOf()`, and `propertyIsEnumerable()`) to perform some inspection of the prototype environment of an object instance. They are of interest primarily to advanced scripters who are building extensive, simulated, object-oriented applications.

Properties

constructor

Value: Object type Read-Only

Compatibility: WinIE4+, MacIE4+, NN4+, Moz+, Safari+, Chrome+

This is the property that is inherited by any custom, `Array`, `String`, or other objects, when one of those objects is instantiated. It is a reference to the constructor function that was used to create the object. Be aware, though, that if a prototype were used, this property might not contain a valid value. Using the earlier car example, if you were to create a car with the `Car` constructor, then the constructor property would refer to `Car`:

```
var car1 = new Car("AA 123", "Ford", "blue");
   if (car1.constructor == Car)    // would evaluate to true
   {
      // statements dealing with the car object
   }
```

Methods

hasOwnProperty("*propName*")

Returns: Boolean.

Compatibility: WinIE5.5+, MacIE-, NN6+, Moz+, Safari-, Chrome+

The `hasOwnProperty()` method returns `true` if the current object instance has the property defined in its constructor or in a related constructor function. But if this property is defined externally, as through assignment to the object's `prototype` property, the method returns `false`.

Using the example of the `Car` and `CarInLot` objects from earlier in this chapter, the following expressions evaluate to `true`:

```
car1.hasOwnProperty("spaceNum");
car1.hasOwnProperty("model");
```

Even though the `model` property is defined in a constructor that is invoked by another constructor, the property belongs to the `car1` object. The following statement, however, evaluates to `false`:

```
car1.hasOwnProperty("companyOwned");
```

This property is defined by way of the prototype of one of the constructor functions and is not a built-in property for the object instance.

isPrototypeOf(*objRef*)

Returns: Boolean

Compatibility: WinIE5.5+, MacIE-, NN6+, Moz+, Safari-, Chrome+

The `isPrototypeOf()` method is intended to reveal whether or not the current object has a prototype relation with an object passed as a parameter. In practice, the IE and NN/Moz versions of this method operate differently and return different results.

propertyIsEnumerable("*propName*")

Returns: Boolean

Compatibility: WinIE5.5+, MacIE-, NN6+, Moz+, Safari-, Chrome+

In the terminology of the ECMA-262 language specification, a value is enumerable if constructions such as the `for...in` property inspection loop (see Chapter 21, "Control Structures and Exception

Handling”) can inspect it. Enumerable properties include values such as arrays, strings, and virtually every kind of object. According to the ECMA specification, this method is not supposed to work its way up the prototype chain.

toSource()

Returns: String

Compatibility: WinIE5.5+, MacIE-, NN6+, Moz+, Safari-, Chrome+

The `toSource()` method obtains a string representation of the source code for an object. Unlike the `toString()` method, which returns a string equivalent of the value of an object, the `toSource()` method returns the underlying code for the object as a string. Seeing as how this is a very specialized, low-level feature, the method is typically only used internally by JavaScript, and potentially in some debugging scenarios.

toString()

Returns: String

Compatibility: WinIE5.5+, MacIE-, NN6+, Moz+, Safari-, Chrome+

The `toString()` method is used to obtain a string representation of the value of an object. It is fairly common to take advantage of this method in situations where an object needs to be examined as raw text. When creating custom objects, you have the ability to create your own `toString()` method that reveals whatever information about the object you desire to be seen.

unwatch("*propName*")

Returns: Boolean

Compatibility: WinIE-, MacIE-, NN6+, Moz+, Safari-, Chrome+

As the counterpart to the `watch()` method, the `unwatch()` method terminates a watchpoint that has been set for a particular property.

valueOf()

Returns: Object

Compatibility: WinIE5.5+, MacIE-, NN6+, Moz+, Safari-, Chrome+

The `valueOf()` method is used to resolve an object to a primitive data type. This may not always be possible, in which case the method simply returns the object itself. Otherwise, the method returns a primitive value that represents the object.

watch("*propName*")

Returns: Boolean

Compatibility: WinIE-, MacIE-, NN6+, Moz+, Safari-, Chrome+

The `watch()` method is the key to a handy little JavaScript debugging feature known as watchpoints. A watchpoint enables you to specify a function that is called whenever the value of a property is set. You can then carefully track the state of properties and take action when an important property value changes.

You set a watchpoint by calling the `watch()` method and passing the name of the property that you want to watch, like this:

```
obj.watch("count",
   function(prop, oldval, newval)
   {
      document.writeln(prop + " changed from " + oldval + " to " + newval);
      return newval;
   }
);
```

In this example, a function is created to output a property change notification and then return the new property value. Every watchpoint handler must follow this same convention of using three arguments that specify the property, old value, and new value, respectively.

CHAPTER 24

Global Functions and Statements

IN THIS CHAPTER

Converting strings into object references

Creating URL-friendly strings

Adding comments to scripts

In addition to all the objects and other language constructs described in the preceding chapters of this reference part of the book, several language items need to be treated on a global scale. These items do not apply to any particular objects (or, in fact, to any object), and you can use them anywhere in a script. If you read earlier chapters, you were introduced to many of these functions and statements. This chapter serves as a convenient place to highlight these all-important items that are otherwise easily forgotten. At the end of the chapter, note the brief introduction to several objects that are built into the Windows-only versions of Internet Explorer.

This chapter begins with coverage of the following global functions and statements that are part of the core JavaScript language:

Functions	Statements
`decodeURI()`	`// and /*...*/` (*comment*)
`decodeURIComponent()`	`const`
`encodeURI()`	`var`
`encodeURIComponent()`	
`escape()`	
`eval()`	
`isFinite()`	
`isNaN()`	
`Number()`	
`parseFloat()`	
`parseInt()`	

Functions	Statements
toString()	
unescape()	
unwatch()	
watch()	

Global functions are not tied to the document object model. Instead, they typically enable you to convert data from one type to another. The list of global statements is short, but a couple of them appear extensively in your scripting.

Functions

decodeURI("*encodedURI*")
decodeURIComponent("*encodedURIComponent*")
encodeURI("*URIString*")
encodeURIComponent("*URIComponentString*")

Returns: String

Compatibility: WinIE5.5+, MacIE–, NN6+, Moz+, Safari+, Opera+, Chrome+

The ECMA-262 standard, as implemented in IE5.5+, NN6+, and Mozilla-based browsers, provides utility functions that perform a more rigorous conversion of strings to valid URI strings, and vice versa, than was achieved earlier via the `escape()` and `unescape()` functions (described later in this chapter). In fact, the modern `encodeURI()`, `encodeURIComponent()`, `decodeURI()`, and `decodeURIComponent()` functions serve as replacements to the now deprecated `escape()` and `unescape()` functions.

The purpose of the encoding functions is to convert any string to a version that you can use as a Uniform Resource Identifier, such as a web page address or an invocation of a server CGI script. Whereas Latin alphanumeric characters pass through the encoding process untouched, you must use the encoding functions to convert some symbols and other Unicode characters into a form (hexadecimal representations of the character numbers) that the Internet can pass from place to place. The space character, for example, must be encoded to its hex version: `%20`.

Perhaps the biggest difference between the `encodeURI()` and `escape()` functions (and their `decodeURI()` and `unescape()` counterparts) is that the more modern versions do not encode a wide range of symbols that are perfectly acceptable as URI characters according to the syntax recommended in RFC2396 (`http://www.ietf.org/rfc/rfc2396.txt`). Thus, the following characters are not encoded via the `encodeURI()` function:

```
; / ? : @ & = + $ , - _ . ! ~ * ' ( ) #
```

Use the `encodeURI()` and `decodeURI()` functions only on complete URIs. Applicable URIs can be relative or absolute, but these two functions are wired especially so that symbols that are part of the protocol (`://`), search string (`?` and `=`, for instance), and directory-level delimiters (`/`) are not encoded. The `decodeURI()` function should work with URIs that arrive from servers as page

locations, but be aware that some server CGIs encode spaces into plus symbols (+) that are not decoded back into spaces by the JavaScript function. If the URIs your script needs to decode contain plus symbols in place of spaces, you need to run your decoded URI through a string replacement method to finish the job (regular expressions come in handy here). If you are decoding URI strings that your scripts encoded, use the decode functions only on URIs that were encoded via the corresponding encode function. Do not attempt to decode a URI that was created via the old `escape()` function, because the conversion processes work according to different rules.

The difference between a URI and a URI component is that a *component* is a single piece of a URI, generally not containing delimiter characters. For example, if you use the `encodeURIComponent()` function on a complete URI, almost all of the symbols (other than things such as periods) are encoded into hexadecimal versions — including directory delimiters. Therefore, you should use the component-level conversion functions only on quite granular pieces of a URI. For example, if you assemble a search string that has a name-value pair, you can use the `encodeURIComponent()` function separately on the name and on the value. But if you use that function on the pair that is already in the form `name=value`, the function encodes the equal symbol to a hexadecimal equivalent.

Since the `escape()` and `unescape()` functions were sometimes used on strings that weren't necessarily URL (URI) strings, you will generally use the `encodeURIComponent()` and `decodeURIComponent()` functions when modernizing code that utilizes `escape()` and `unescape()`.

Example

Use The Evaluator (see Chapter 4, "JavaScript Essentials") to experiment with the differences between encoding a full URI and a component, and encoding and escaping a URI string. For example, compare the results of the following three statements:

```
escape("http://www.giantco.com/index.htm?code=42")
encodeURI("http://www.giantco.com/index.htm?code=42")
encodeURIComponent("http://www.giantco.com/index.htm?code=42")
```

Because the sample URI string is valid as is, the `encodeURI()` version makes no changes. Experiment further by making the search string value into a string with a space, and see how each function treats that character.

escape("*URIString*" [,1])
unescape("*escapedURIString*")

Returns: String

Compatibility: WinIE3+, MacIE3+, NN2+, Moz+, Safari+, Opera+, Chrome+

If you watch the content of the Location field in your browser, you may occasionally see URLs that include a lot of % symbols, plus some numbers. The format you see is *URL encoding* (more accurately called *URI encoding* — Uniform Resource Identifier rather than Uniform Resource Locator). This format allows even multiple word strings and nonalphanumeric characters to be sent as one contiguous string of a very low, common denominator character set. This encoding turns a character, such as a space, into its hexadecimal equivalent value, preceded by a percent symbol. For example, the space character (ASCII value 32) is hexadecimal 20, so the encoded version of a space is `%20`.

All characters, including tabs and carriage returns, can be encoded in this way and sent as a simple string that can be decoded on the receiving end for reconstruction. You can also use this encoding

to preprocess multiple lines of text that must be stored as a character string in databases. To convert a plain-language string to its encoded version, use the `escape()` method. This function returns a string consisting of the encoded version. For example:

```
var theCode = escape("Hello there");
    // result: Hello%20there
```

Most, but not all, non-alphanumeric characters are converted to escaped versions with the `escape()` function. One exception is the plus sign, which URLs use to separate components of search strings. If you must encode the plus symbol, too, then add the optional second parameter to the function to make the plus symbol convert to its hexadecimal equivalent (2B):

```
var a = escape("Adding 2+2");
    // result: Adding%202+2
var a = escape("Adding 2+2",1);
    // result: Adding%202%2B2
```

Note that browser support is not consistent in the treatment of the optional second parameter; the same result could be returned whether or not the second argument is included.

The `unescape()` function was used to convert an escaped string back into plain language. We say "was" because the function is now deprecated thanks to `decodeURI()` and `decodeURIComponent()` and shouldn't be used.

The `escape()` function operates in a way that is approximately midway between the newer functions `encodeURI()` and `encodeComponentURI()`. However, the function is now deprecated in lieu of the newer functions and shouldn't be used.

eval("*string*")

Returns: Object reference

Compatibility: WinIE3+, MacIE3+, NN2+, Moz+, Safari+, Opera+, Chrome+

Expression evaluation, as you probably are well aware by now, is an important concept to grasp for scripting with JavaScript (and for programming in general). An expression evaluates to some value. But occasionally, you need to force an additional evaluation on an expression to receive the desired results. The `eval()` function acts on a string value to force an evaluation of that string expression. Perhaps the most common application of the `eval()` function is to convert a string version of an object reference to a genuine object reference.

Example

The `eval()` function can evaluate any JavaScript statement or expression stored as a string. This includes string equivalents of arithmetic expressions, object value assignments, and object method invocation. We do not recommend that you rely on the `eval()` function, however, because this function is inherently inefficient (from the standpoint of performance). Fortunately, you may not need the `eval()` function to get from a string version of an object's name to a valid object reference. For example, if your script loops through a series of objects whose names include serial numbers, you can use the object names as array indexes rather than use `eval()` to assemble the object references. The inefficient way to set the value of a series of fields named `data0`, `data1`, and so on, is as follows:

```
function fillFields()
{
   var theObj;
```

```
        for (var i = 0; i < 10; i++)
        {
            theObj = eval("document.forms[0].data" + i);
            theObj.value = i;
        }
    }
```

A more efficient way is to perform the concatenation within the index brackets for the object reference:

```
function fillFields()
{
    for (var i = 0; i < 10; i++)
    {
        document.getElementById("myForm").elements["data" + i].value = i;
    }
}
```

Tip

Whenever you are about to use an `eval()` function, look for ways to use string index values of arrays of objects instead. The W3C DOM makes it even easier with the help of the `document.getElementById()` method, which takes a string as a parameter and returns a reference to the named object. ■

`isFinite(`*`number`*`)`

Returns: Boolean

Compatibility: WinIE4+, MacIE4+, NN4+, Moz+, Safari+, Opera+, Chrome+

It is rare that you will ever need the `isFinite()` function, but its purpose is to advise whether a number is beyond the absolute minimum or maximum values that JavaScript can handle. If a number is outside of that range, the function returns `false`. The parameter to the function must be a number data type.

`isNaN(`*`expression`*`)`

Returns: Boolean

Compatibility: WinIE3+, MacIE3+, NN2+, Moz+, Safari+, Opera+, Chrome+

For those instances in which a calculation relies on data coming from a text field or other string-oriented source, you frequently need to check whether the value is a number. If the value is not a number, the calculation may result in a script error.

Example

Use the `isNaN()` function to test whether a value is a number, prior to passing the value onto a calculation. The most common use of this function is to test the result of a `parseInt()` or `parseFloat()` function. If the strings submitted for conversion to those functions cannot be converted to a number, the resulting value is `NaN` (a special symbol indicating "not a number"). The `isNaN()` function returns `true` if the value is not a number.

A convenient way to use this function is to intercept improper data before it can do damage, as follows:

```
function calc(form)
{
   var inputValue = parseInt(form.entry.value);
      if (isNaN(inputValue))
      {
         alert("You must enter a number to continue.");
      }
      else
      {
         // statements for calculation
      }
}
```

Probably the biggest mistake scripters make with this function is failing to observe the case of all the letters in the function name. The trailing uppercase "N" is easy to miss.

Number("*string*")
parseFloat("*string*")
parseInt("*string*" [,*radix*])

Returns: Number

Compatibility: WinIE3+, MacIE3+, NN2+, Moz+, Safari+, Opera+, Chrome+

All three of these functions convert a string value into a numeric value. The `parseInt()` and `parseFloat()` functions are compatible across all versions of all browsers, including very old browsers; the `Number()` function was introduced in version 4 browsers.

Use the `Number()` function when your script is not concerned with the precision of the value, and prefers to let the source string govern whether the returned value is a floating-point number or an integer. The function takes a single parameter — a string to convert to a number value.

The `parseFloat()` function also lets the string source value determine whether the returned value is a floating-point number or an integer. If the source string includes any non-zero value to the right of the decimal, the result is a floating-point number. But if the string value were, say, `"3.00"`, the returned value would be an integer value.

An extra, optional parameter for `parseInt()` enables you to define the number base for use in the conversion. If you don't specify a radix parameter, JavaScript tries to look out for you; but in doing so, JavaScript may cause some difficulty for you. The primary problem arises when the string parameter for `parseInt()` starts with a zero, which a text box entry or database field might do. In JavaScript, numbers starting with zero are treated as octal (base-8) numbers. Therefore, `parseInt("010")` yields the decimal value 8.

When you apply the `parseInt()` function, always specify the radix of 10 if you are working in base-10 numbers. You can, however, specify any radix value from 2 through 36. For example, to convert a binary number string to its decimal equivalent, assign a radix of 2 as follows:

```
var n = parseInt("011",2);
   // result: 3
```

Similarly, you can convert a hexadecimal string to its decimal equivalent by specifying a radix of 16:

```
var n = parseInt("4F",16);
    // result: 79
```

Example

Both parseInt() and parseFloat() exhibit a very useful behavior: If the string passed as a parameter starts with at least one number followed by, say, letters, the functions do their jobs on the numeric part of the string and ignore the rest. This is why you can use parseFloat() on the navigator.appVersion string to extract just the reported version number without having to parse the rest of the string. For example, Firefox 1.0 for Windows reports a navigator.appVersion value as

```
5.0 (Windows; en-US)
```

But you can get just the numeric part of the string via parseFloat():

```
var ver = parseFloat(navigator.appVersion);
```

Note

The number stored in the navigator.appVersion property refers to the version number of the underlying browser engine, which helps explain why the reported version in this case is 5.0, even though the Firefox browser application is considered version 1.0. ■

Because the result is a number, you can perform numeric comparisons to see, for instance, whether the version is greater than or equal to 4.

toString([*radix*])

Returns: String

Compatibility: WinIE4+, MacIE4+, NN4+, Moz+, Safari+, Opera+, Chrome+

Every JavaScript core language object and every DOM document object has a toString() method associated with it. This method is designed to render the contents of the object as a string of text that is as meaningful as possible. Table 24-1 shows the result of applying the toString() method on each of the convertible core language object types.

TABLE 24-1

toString() Method Results for Object Types

Object Type	Result
String	The same string
Number	String equivalent (but numeric literals cannot be converted)
Boolean	true or false
Array	Comma-delimited list of array contents (with no spaces after commas)
Function	Decompiled string version of the function definition

toString()

Many DOM objects can be converted to a string. For example, a `location` object returns its URL. But when an object has nothing suitable to return for its content as a string, it usually returns a string in the following format:

```
[object objectType]
```

Example

The `toString()` method is available on all versions of all browsers. By setting the optional `radix` parameter between 2 and 16, you can convert numbers to string equivalents in different number bases. Listing 24-1 calculates and draws a conversion table for decimal, hexadecimal, octal, and binary numbers between 0 and 20. In this case, the source of each value is the value of the index counter variable, each time the `for` loop's statements execute.

Note

The function to assign event handlers throughout the code in this chapter and much of the book is `addEvent()`, a cross-browser event handler explained in detail in Chapter 32, "Event Objects."

The `addEvent()` function is part of the script file `jsb-global.js` located on the accompanying CD-ROM in the `Content/` folder, where it is accessible to all chapters' scripts. ■

LISTING 24-1

Using toString() with Radix Values

HTML: jsp-24-01.html

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Using toString() to convert to other number bases:</title>
      <link rel="stylesheet" type="text/css" href="jsp-24-01.css">
      <script type="text/javascript" src="../jsb-global.js"></script>
      <script type="text/javascript" src="jsb-24-01.js"></script>
   </head>
   <body>
      <h1>Using toString() to convert to other number bases:</h1>
      <table id="numberConversions">
         <tr>
            <th>Decimal</th>
            <th>Hexadecimal</th>
            <th>Octal</th>
            <th>Binary</th>
         </tr>
      </table>
   </body>
</html>
```

CSS: jsp-24-01.css

```
th, td
{
   border: 5px groove black;
```

```
   text-align: center;
   padding-left: 2px;
   padding-right: 2px;
}

th
{
   font-weight: bold;
}
```

JavaScript: jsb-24-01.js

```
// initialize when the page has loaded
addEvent(window, "load", initialize);

function initialize()
{
   var theTable = document.getElementById("numberConversions");

      if (theTable)
      {
         var newRowElem;
         var newCellElem;
         var newText;
         // Insert the rows after the existing row
         var newRowPosition = 1;

         // put the converted number data into the table
         for (var i = 0; i <= 20; i++)
         {

            // create a new row
            newRowElem = theTable.insertRow(newRowPosition);

            // create the 1st new cell in the new row and its text
            newCellElem = newRowElem.insertCell(0);
            newText = document.createTextNode(i.toString(10));
            newCellElem.appendChild(newText);

            // create the next new cell in the new row and its text
            newCellElem = newRowElem.insertCell(1);
            newText = document.createTextNode(i.toString(8));
            newCellElem.appendChild(newText);

            // create the next new cell in the new row and its text
            newCellElem = newRowElem.insertCell(1);
            newText = document.createTextNode(i.toString(16));
            newCellElem.appendChild(newText);

            // create the next new cell in the new row and its text
            newCellElem = newRowElem.insertCell(2);
            newText = document.createTextNode(i.toString(2));
```

continued

LISTING 24-1 *(continued)*

```
              newCellElem.appendChild(newText);

              newRowPosition += 1;

           }
        }
}
```

The `toString()` method of user-defined objects does not convert the object into a meaningful string, but you can create your own method to do just that. For example, if you want to make your custom object's `toString()` method behave like an array's method, define the action of the method and assign that function to a property of the object (as shown in Listing 24-2).

LISTING 24-2

Creating a Custom toString() Method

HTML: jsp-24-02.html

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>A custom object defined toString() result:</title>
      <script type="text/javascript" src="../jsb-global.js"></script>
      <script type="text/javascript" src="jsb-24-02.js"></script>
   </head>
   <body>
      <h1>A custom object defined toString() result:</h1>
      <!-- Hard code what's in the object so we can validate
           the JavaScript visually on the page. -->
      <h3>The custom object is book and its properties are</h3>
      <h4>title:"The Aeneid", author:"Virgil", pageCount:543</h4>
      <h3>Now let's look at the results of the custom object's toString() in
          the JavaScript:</h3>
      <div id="placeHolder"></div>
   </body>
</html>
```

JavaScript: jsb-24-02.js

```
// initialize when the page has loaded
addEvent(window, "load", initialize);

var newElem;
var newText;
```

```
// define the function that will be the method for the custom object
function customToString()
{
   var dataArray = new Array();
   var count = 0;
   for (var i in this)
   {
      dataArray[count++] = this[i];
         if (count > 2)
         {
            break;
         }
   }
   return dataArray.join(",");
}

// define a custom object
var book = { title:"The Aeneid", author:"Virgil", pageCount:543 };

// declare a method for the custom object
book.toString = customToString;

// the onload event we identified earlier
function initialize()
{

   var placeHolderElement = document.getElementById("placeHolder");

      if (placeHolderElement)
      {
         //book.toString();

         newElem = document.createElement("h4");
         newText = document.createTextNode(book.toString());
         // insert the text into the new h4
         newElem.appendChild(newText);
         // insert the completed h4 into placeholder
         placeHolderElement.appendChild(newElem);

      }

}
```

When you run Listing 24-2, you can see how the `custom` object's `toString()` handler extracts the values of all elements of the object. You can customize how the data should be labeled and/or formatted.

Keep in mind that you can provide a custom `toString()` method to any object that you create, not just arrays. This is a handy way to provide a glimpse at the contents of an object for debugging purposes. For example, you could craft a `toString()` method that carefully formats all of the properties of an object into an easily readable string of text. Then, use an alert box or browser debugging console to view the contents of the object if a problem arises.

```
unwatch(property)
watch(property, handler)
```

Returns: Nothing

Compatibility: WinIE-, MacIE-, NN4+, Moz-, Safari-, Opera-, Chrome-

To supply the right kind of information to external debuggers, JavaScript in NN4+ implements two global functions that belong to every object — including user-defined objects. The `watch()` function keeps an eye on a desired object and property. If that property is set by assignment, the function invokes another user-defined function that receives information about the property name, its old value, and its new value. The `unwatch()` function turns off the watch functionality for a particular property.

Statements

//
/*...*/

Compatibility: WinIE3+, MacIE3+, NN2+, Moz+, Safari+, Opera+, Chrome+

Comments are statements that the JavaScript interpreter (or server-side compiler) ignores. However, these statements enable authors to leave notes about how things work in their scripts. Although lavish comments are useful to authors during a script's creation and maintenance, the full content of a client-side comment is downloaded with the document. Every byte of non-operational content of the page takes a bit more time to download. Still, we recommend lots of comments — particularly as you create a script.

JavaScript offers two styles of comments. One style consists of two forward slashes (no spaces between them), and is useful for creating a comment on a single line. JavaScript ignores any characters to the right of those slashes on the same line, even if they appear in the middle of a line. You can stack as many lines of these single-line comments as is necessary to convey your thoughts. We typically place a space between the second slash and the beginning of the comment. The following are examples of valid, one-line comment formats:

```
// this is a comment line usually about what's to come
var a = "Fred";  // a comment about this line
// You may want to capitalize the first word of a comment
// sentence if it runs across multiple lines.
//
// And you can leave a completely blank line, like the one above.
```

For longer comments, it is usually more convenient to enclose the section in the other style of comment, which is fully capable of spanning multiple lines. The following comment opens with a forward slash and asterisk (`/*`) and ends with an asterisk and forward slash (`*/`). JavaScript ignores all statements in between — including multiple lines. If you want to temporarily comment out a large segment of your script for debugging purposes, it is easiest to enclose the segment with these comment symbols. To make comment blocks easier to find, we generally place the symbols on their own lines as follows:

```
/*
some
  commented-out
    statements
*/
```

If you are developing rather complex scripts, you might find using comments a convenient way to help you organize segments of your scripts and make each segment easier to find. For example, you can define a comment block above each function and describe what the function is about, as in the following example.

```
/*-----------------------------------------------
   calculate()
   Performs a mortgage calculation based on
   parameters blah, blah, blah.  Called by blah
   blah blah.
-----------------------------------------------*/
function calculate(form)
{
   // statements
}
```

const

Compatibility: WinIE-, MacIE-, NN6+, Moz+, Safari+, Opera+, Chrome+

The `const` keyword initializes a constant. Unlike a variable, whose data is subject to change while a page loads, a constant's value cannot be modified once it is assigned. It is common practice in many programming languages to define constant identifiers with all uppercase letters, usually with underscore characters to delimit multiple words. This style makes it easier to quickly find constants in your code and reminds you that their values are fixed.

Example

Listing 24-3 shows how you can use a constant in any browser other than IE. The page conveys temperature data for several cities. (Presumably, this data is updated on the server and fashioned into an array of data when the user requests the page.) For temperatures below freezing, the temperature is shown in a distinctive text style. Because the freezing temperature is a constant reference point, it is assigned as a constant.

LISTING 24-3

Using the const Keyword

HTML: jsp-24-03.html

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>The const keyword</title>
```

continued

LISTING 24-3 *(continued)*

```
      <link rel="stylesheet" type="text/css" href="jsp-24-03.css">
      <script type="text/javascript" src="../jsb-global.js"></script>
      <script type="text/javascript" src="jsb-24-03.js"></script>
   </head>
   <body>
      <h1>The const keyword</h1>
      <table id="temps">
         <tr>
            <th>City</th>
            <th><div>Temperature</div><div>(Fahrenheit)</div></th>
         </tr>
      </table>
   </body>
</html>
```

CSS: jsp-24-03.css

```
.cold
{
   font-weight: bold;
   color: blue;
}

td
{
   text-align: center;
}
```

JavaScript: jsb-24-03.js

```
// initialize when the page has loaded
addEvent(window, "load", showData);

const FREEZING_F = 32;

var cities = ["London", "Moscow", "New York", "Tokyo", "Sydney"];

var cityTempsF = [33, 12, 20, 40, 75];

function showData()
{
   var theTable = document.getElementById("temps");

      if (theTable)
      {
         var newRowElem;
         var newCellElem;
         var newText;
         // Insert the rows after the existing row
         var newRowPosition = 1;
```

```
      // put the city temperature data in the table
      for (var i = 0; i < cities.length; i++)
      {
         // create a new row
         newRowElem = theTable.insertRow(newRowPosition);

         // create the new city cell in the new row and its text
         newCellElem = newRowElem.insertCell(0);
         newText = document.createTextNode(cities[i]);
         newCellElem.appendChild(newText);

         // create the new temp cell in the new row and its text
         newCellElem = newRowElem.insertCell(1);
         newText = document.createTextNode(cityTempsF[i]);
         newCellElem.appendChild(newText);

         // style the cells with cold data
            if (cityTempsF[i] < FREEZING_F)
            {
               newCellElem.className = "cold";
            }
         newRowPosition += 1;
      }
   }
}
```

The `const` keyword likely will be adopted in the next version of the ECMA-262 standard and will become an official part of the JavaScript vernacular in future browsers. In the meantime, it enjoys full support in Mozilla-based browsers.

var

Compatibility: WinIE3+, MacIE3+, NN2+, Moz+, Safari+, Opera+, Chrome+

Before using any variable, you should declare it (and optionally, initialize it with a value) via the `var` statement. If you omit the `var` keyword, the variable is automatically assigned as a global variable within the current document. To keep a variable local to a function, you must declare or initialize the variable with the `var` keyword inside the function's braces.

If you do not assign a value to a variable, it evaluates to `null`. Because a JavaScript variable is not limited to one variable type during its lifetime, you don't need to initialize a variable to an empty string or zero unless that initial value helps your scripting. (For example, if you initialize a variable as an empty string, you can then use the add-by-value operator (+=) to append string values to that variable in a future statement in the document.)

To save statement lines, you can declare, and/or initialize, multiple variables with a single `var` statement. Separate each `varName=value` pair with a comma, as in

```
var name, age, height;  // declare as null
var color = "green", temperature = 85.6; // initialize
```

Variable names (also known as identifiers) must be one contiguous string of characters, and the first character must be a letter. Many punctuation symbols are banned, but the underscore character is

valid and is often used to separate multiple words in a long variable name. All variable names (like most identifiers in JavaScript) are case-sensitive, so you must name a particular variable identically throughout the variable's scope.

By convention, variable names are written in camelCase, as in

```
var colorCode = "green";
```

WinIE Objects

Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-, Opera-, Chrome-

For better or worse, Microsoft prides itself on the integration between web browser functionality and the Windows operating system. The linkage between browser and OS is most apparent in IE's facilities for accessing ActiveX objects. Microsoft has fashioned several such objects for access to scripters — again, provided the deployment is intended only for Windows versions of Internet Explorer. Some objects also exist as a way to expose some Visual Basic Script (VBScript) functionality to JavaScript. Because these objects are more within the realm of Windows and ActiveX programming, the details and quirks of working with them from WinIE is best left to other venues. But in case you are not familiar with these facilities, the following discussions introduce the basic set of WinIE objects. You can find more details at the Microsoft Developer Network (MSDN) web site at `http://msdn.microsoft.com/`.

The objects mentioned here are the `ActiveXObject`, `Dictionary`, `Enumerator`, and `VBArray` objects. Microsoft documents these objects as if they were part of the native JScript language. However, you can be sure that they will remain proprietary to Internet Explorer, if not exclusively for Windows-only versions.

Note

JScript is Microsoft's proprietary take on JavaScript, supported by Internet Explorer. JScript is essentially the same as JavaScript with a few Windows-specific extras thrown in, such as support for ActiveX objects. ■

ActiveXObject

`ActiveXObject` is a generic object that allows your script to open and access what Microsoft sometimes calls *automation objects*. An automation object is an executable program that might run on the client or be served from a server. This can include local applications, such as applications from the Microsoft Office suite, executable DLLs (dynamic-link libraries), and so on.

Use the constructor for the `ActiveXObject` to obtain a reference to the object according to the following syntax:

```
var objRef = new ActiveXObject(appName.className[, remoteServerName]);
```

This JScript syntax is the equivalent of the VBScript `CreateObject()` method. You need to know a bit about Windows programming to determine the application name and the classes or types available for that application. For example, to obtain a reference to an Excel worksheet, use this constructor:

```
var mySheet = new ActiveXObject("Excel.Sheet");
```

Once you have a reference to the desired object, you must also know the names of the properties and methods of the object you'll be addressing. You can access much of this information via Microsoft's developer tools, such as Visual Studio .NET, or the tools that come with Visual Basic .NET. These

tools enable you to query an object to discover its properties and methods. Unfortunately, an `ActiveXObject`'s properties are not enumerable through a typical JavaScript `for...in` property inspector.

Accessing an `ActiveXObject`, especially one on the client, involves some serious security considerations. The typical security setup for an IE client prevents scripts from accessing client applications, at least not without asking the user if it's okay to do so. While it's foolhardy to state categorically that one cannot perform surreptitious inspection or damage to a client without the user's knowledge (hackers find holes from time to time), it is highly unlikely. In a corporate environment, where some level of access to all clients is desirable, the client may be set up to accept instructions to work with ActiveX objects when they come from trusted sources. The bottom line is that unless you are well versed in Windows programming, don't expect the `ActiveXObject` to become some kind of magic portal that enables you to invade the privacy or security of unsuspecting users.

Dictionary

Although the `Dictionary` object is very helpful to VBScript authors, JavaScript already provides the equivalent functionality natively. A `Dictionary` object behaves very much like a JavaScript array that has string index values (similar to a Java hash table), although numeric index values are also acceptable in the `Dictionary`. Indexes are called *keys* in this environment. VBScript arrays do not have this facility natively, so the `Dictionary` object supplements the language for the sake of convenience. Unlike a JavaScript array, however, you must use the various properties and methods of the `Dictionary` object to add, access, or remove items from it.

You create a `Dictionary` object via the `ActiveXObject` as follows:

```
var dict = new ActiveXObject("Scripting.Dictionary");
```

You must create a separate `Dictionary` object for each array. Table 24-2 lists the properties and methods of the `Dictionary` object. After you create a blank `Dictionary` object, populate it via the `Add()` method for each entry. For example, the following statements create a `Dictionary` object to store U.S. state capitals:

```
var stateCaps = new ActiveXObject("Scripting.Dictionary");
stateCaps.Add("Illinois", "Springfield");
```

You can then access an individual item via the `Key` property (which, thanks to its VBScript heritage, looks more like a JavaScript method). One convenience of the `Dictionary` object is the `Keys()` method, which returns an array of all the keys in the dictionary — something that a string-indexed JavaScript array could use.

Enumerator

An `Enumerator` object provides JavaScript with access to collections that otherwise do not allow direct access to their items, via index number or name. This object isn't necessary when working with DOM collections, such as `document.all`, because you can use the `item()` method to obtain a reference to any member of the collection. But if you are scripting ActiveX objects, some of these objects' methods or properties may return collections that cannot be accessed through this mechanism or the JavaScript `for...in` property inspection technique. Instead, you must wrap the collection inside an `Enumerator` object.

To wrap a collection in an `Enumerator`, invoke the constructor for the object, passing the collection as the parameter:

```
var myEnum = new Enumerator(someCollection);
```

TABLE 24-2

Dictionary Object Properties and Methods

Property	Description
`Count`	Integer number of entries in the dictionary (read-only)
`Item("key")`	Reads or writes a value for an entry whose name is `key`
`Key("key")`	Assigns a new key name to an entry
Method	**Description**
`Add("key", value)`	Adds a value associated with a unique key name
`Exists("key")`	Returns Boolean `true` if `key` exists in dictionary
`Items()`	Returns `VBArray` of values in dictionary
`Keys()`	Returns `VBArray` of keys in dictionary
`Remove("key")`	Removes `key` and its value
`RemoveAll()`	Removes all entries

This enumerator instance must be accessed, via one of its four methods, to position a "pointer" to a particular item and then extract a copy of that item. In other words, you don't access a member directly (that is, by diving into the collection with an item number to retrieve). Instead, you move the pointer to the desired position and then read the item value. As you can see from the list of methods in Table 24-3, this object is truly intended for looping through the collection. Pointer control is limited to positioning it at the start of the collection and incrementing its position along the collection by one:

```
var val;
myEnum.moveFirst();
for (; !myEnum.atEnd(); myEnum.moveNext())
{
   val = myEnum.item();
   // more statements that work on value
}
```

VBArray

The `VBArray` object provides JavaScript access to Visual Basic *safe arrays*. Such an array is read-only and is commonly returned by ActiveX objects. Such arrays can be composed in VBScript sections of client-side scripts. Visual Basic arrays, by their very nature, can have multiple dimensions. For example, the following code creates a three-by-two VB array:

```
<script type="text/vbscript">
   Dim myArray(2, 1)
   myArray(0, 0) = "A"
   myArray(0, 1) = "a"
```

```
   myArray(1, 0) = "B"
   myArray(1, 1) = "b"
   myArray(2, 1) = "C"
   myArray(2, 2) = "c"
</script>
```

TABLE 24-3

Enumerator Object Methods

Method	Description
`atEnd()`	Returns `true` if pointer is at end of collection
`item()`	Returns value at current pointer position
`moveFirst()`	Moves pointer to first position in collection
`moveNext()`	Moves pointer to next position in collection

Once you have a valid VB array, you can convert it to an object that the JScript interpreter can't choke on:

```
<script type="text/javascript">
   var theVBArray = new VBArray(myArray);
</script>
```

Global variables from one script language block can be accessed by another block, even in a different language. But at this point, the array is not yet in the form of a JavaScript array. You can either convert it to such via the `VBArray.toArray()` method, or access information about the `VBArray` object through its other methods (described briefly in Table 24-4). Once you convert a `VBArray` to a JavaScript array, you can then iterate through the values, as with any JavaScript array.

TABLE 24-4

VBArray Object Methods

Method	Description
`dimensions()`	Returns number of dimensions of the original array
`getItem(dim1[, dim2[,...dimN]])`	Returns value at array location defined by dimension addresses
`ibound(dim)`	Returns lowest index value for a given dimension
`toArray()`	Returns JavaScript array version of `VBArray`
`ubound(dim)`	Returns highest index value for a given dimension

When you use the `toArray()` method, and the source array has multiple dimensions, values from dimensions after the first "row" are simply appended to the JavaScript array with no nesting structures.

Part IV

Document Objects Reference

IN THIS PART

Chapter 25
Document Object Model Essentials

Chapter 26
Generic HTML Element Objects

Chapter 27
Window and Frame Objects

Chapter 28
Location and History Objects

Chapter 29
Document and Body Objects

Chapter 30
Link and Anchor Objects

Chapter 31
Image, Area, Map, and Canvas Objects

Chapter 32
Event Objects

CHAPTER 25

Document Object Model Essentials

IN THIS CHAPTER

Object models versus browser versions

Proprietary model extensions

Structure of the W3C DOM

Scripting trends

Without question, the biggest challenge facing client-side web scripters is the sometimes-baffling array of document object models (DOMs) that have competed for our attention throughout the short history of scriptable browsers. Netscape got the ball rolling in Navigator 2 with the first object model. By the time version 4 browsers came around, the original object model had gained not only some useful cross-browser features, but also a host of features that were unique to Navigator or Internet Explorer. The object models were diverging, causing no end of headaches for page authors whose scripts had to run on as many browsers as possible. A ray of hope emerged from the standards process of the World Wide Web Consortium (W3C) in the form of a DOM recommendation. The DOM brought forward much of the original object model, plus new ways of consistently addressing every object in a document. The goal of this chapter is to put each of the object models into perspective and help you understand how modern browsers have alleviated most of the object model compatibility problems. But before we get to those specifics, let's examine the role of the object model in designing scripted applications.

The Object Model Hierarchy

The tutorial chapters of Part II introduced the fundamental ideas behind a document object hierarchy in scriptable browsers. In other object-oriented environments, object hierarchy plays a much greater role than it does in JavaScript-able browsers. (In JavaScript, you don't have to worry about related terms, such as classes, inheritance, and instances.) Even so, you cannot ignore the hierarchy concept because some of your code relies on your ability to write references to objects that depend on their positions within the hierarchy.

Calling these objects *JavaScript objects* is not correct. They are really browser document objects: You just happen to use the JavaScript language to bring them to life. Some scripters of Microsoft Internet Explorer use the VBScript language to script the very same document objects. Technically speaking, JavaScript objects apply to data types and other core language objects separate from the document.

Hierarchy as road map

For the programmer, the primary role of the document object hierarchy is to provide a way for scripts to reference a particular object among all the objects that a browser window can contain. The hierarchy acts as a road map that the script can use to know precisely which object to address.

Consider, for a moment, a scene in which you and your friend Tony are in a high-school classroom. It's getting hot and stuffy as the afternoon sun pours in through the wall of windows on the west side of the room. You ask Tony, "Would you please open a window?" and motion your head toward a particular window in the room. In programming terms, you've issued a command to an object (whether or not Tony appreciates the comparison). This human interaction has many advantages over anything you can do in programming. First, by making eye contact with Tony before you speak, he knows that he is the intended recipient of the command. Second, your body language passes along some parameters with that command, pointing ever so subtly to a particular window on a particular wall.

If, instead, you are in the principal's office using the public address system, and you broadcast the same command ("Would you please open a window?"), no one knows what you mean. Issuing a command without directing it to an object is a waste of time, because every object thinks, "That can't be meant for me." To accomplish the same goal as your one-on-one command, the broadcast command has to be something like "Would Tony Jeffries in Room 312 please open the middle window on the west wall?"

Let's convert this last command to JavaScript *dot syntax* form (see Chapter 6, "Browser and Document Objects"). Recall from the tutorial that a reference to an object starts with the most global point of view and narrows to the most specific point of view. From the point of view of the principal's office, the location hierarchy of the target object is

```
room312.Jeffries.Tony
```

You can also say that Tony's knowledge about how to open a window is one of Tony's methods. The complete reference to Tony and his method then becomes

```
room312.Jeffries.Tony.openWindow()
```

Your job isn't complete yet. The method requires a parameter detailing which window to open. In this case, the window you want is the middle window of the west wall of Room 312. Or, from the hierarchical point of view of the principal's office, it becomes

```
room312.westWall.middleWindow
```

This object road map is the parameter for Tony's `openWindow()` method. Therefore, the entire command that comes over the PA system is

```
room312.Jeffries.Tony.openWindow(room312.westWall.middleWindow)
```

If, instead of barking out orders while sitting in the principal's office, you attempt the same task through radio from an orbiting space shuttle to all the inhabitants on Earth, imagine how laborious your object hierarchy would be. The complete reference to Tony's `openWindow()` method and the window that you want opened wolud have to be mighty long to distinguish the desired objects from the billions of objects within the space shuttle's view.

The point is that the smaller the scope of the object-oriented world you're programming, the more you can assume about the location of objects. For client-side JavaScript, the scope is no wider than

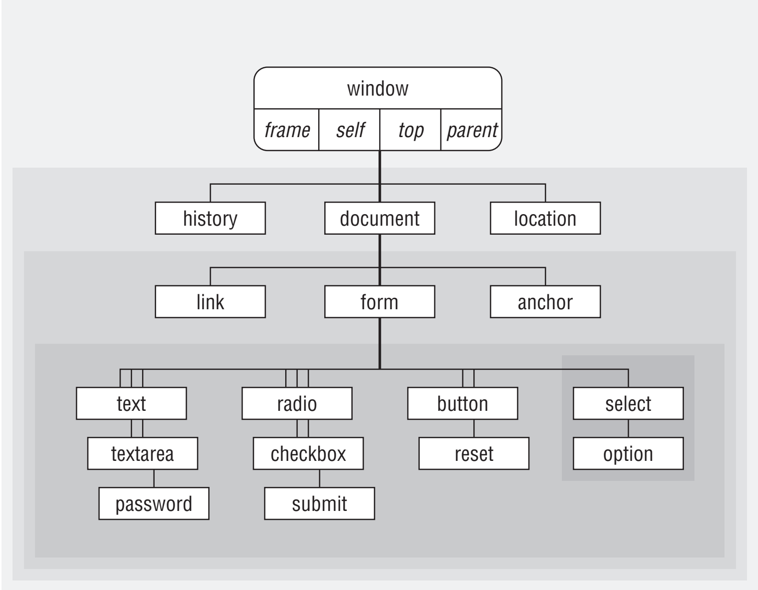
the browser itself. In other words, every object that a JavaScript script can work with resides within the browser application. With few exceptions, a script does not access anything about your computer hardware, operating system, other applications, desktop, or any other stuff beyond the browser program.

The first browser document object road map

Figure 25-1 shows the lowest-common-denominator document object hierarchy that is implemented in all scriptable browsers, including scriptable legacy browsers such as IE3 and NN2. Notice that the `window` object is the topmost object in the entire scheme. Everything you script in JavaScript is in the browser's window.

FIGURE 25-1

The lowest-common-denominator browser document object hierarchy.



Pay attention to the shading of the concentric rectangles. Every object in the same shaded area is at the same level relative to the `window` object. When a line from an object extends to the next-darker shaded rectangle, that object contains all the objects in darker areas. At most, one of these lines exists between levels. The `window` object contains the `document` object; the `document` object contains a form object; a form object contains many different kinds of form control elements.

How Document Objects Are Born

Most of the objects that a browser creates for you are established when an HTML document loads into the browser. The same kind of HTML code you use to create links, anchors, and input elements tells a JavaScript-enhanced browser to create those objects in memory. The objects are there whether or not your scripts call them into action.

The only visible differences to the HTML code for defining those objects are the one or more optional attributes specifically dedicated to JavaScript. By and large, these attributes specify the event you want the user interface element to react to and what JavaScript should do when the user takes that action. By relying on the document's HTML code to perform the object generation, you can spend more time figuring out how to do things with those objects, or how to have them do things for you.

Bear in mind that objects are created in their load order. If you create a multiframe environment, a script in one frame cannot communicate with another frame's objects until both frames load. This trips up a lot of scripters who create multiframe and multiwindow sites (for more information, see Chapter 27, "Window and Frame Objects").

But interestingly enough, you are not confined to just those objects that the document's HTML code generates. As you'll see later in this chapter, the W3C DOM allows you to add and delete objects from the object hierarchy. In fact, if you want to create HTML code that contains no JavaScript, you'll see how you can do that as well.

Object Properties

A property generally defines a particular current setting of an object. The setting may reflect a visible attribute of an object, such as the state of a checkbox (selected or not); it may also contain information that is not so obvious, such as the action and method of a submitted form.

Document objects have most of their initial properties assigned by the attribute settings of the HTML tags that generate the objects. Thus, a property may be a word (for example, a name) or a number (for example, a size). A property can also be an array, such as an array of images contained by a document. If the HTML does not include all attributes, the browser usually fills in a default value for both the attribute and the corresponding JavaScript property.

A Note to Experienced Object-Oriented Programmers

Although the basic object model hierarchy appears to have a class/subclass relationship, many of the traditional aspects of a true object-oriented environment don't apply to the model. The original JavaScript document object hierarchy is a *containment hierarchy,* not an *inheritance hierarchy.* No object inherits properties or methods of an object higher up the chain. Neither is there any automatic message passing from object to object in any direction. Therefore, you cannot invoke a window's method by sending a message to it through the `document` or a form object. All object references must be explicit.

Predefined document objects are generated only when the HTML code containing their definitions loads into the browser. In Chapter 23, ''Function Objects and Custom Objects,'' you learned how to create your own objects, but those objects do not present new visual elements on the page that go beyond what HTML, Java applets, and plug-ins can portray.

Inheritance *does* play a role, as you will see later in this chapter, in the object model defined by the W3C. This newer hierarchy is of a more general nature to accommodate requirements of XML as well as HTML. But the containment hierarchy for HTML objects, as described in this section, is still valid in W3C DOM–compatible browsers.

Property values may be changed in scripts, regardless of the DOM you are using; the difference is the code you write. Later in this chapter, we'll give you specifics for the examples you'll see now. Where applicable, we'll show the examples in DOM0 as well as the W3C DOM.

When used in script statements, property names are case-sensitive. Therefore, if you see a property name listed as `bgColor`, you must use it in a script statement with that exact combination of lowercase and uppercase letters. But when you set an initial value of a property by way of an HTML attribute, the attribute name (like all of HTML) is not case sensitive. Thus, `<BODY BGCOLOR="white">` and `<body bgcolor="white">` both set the same `bgColor` property value. Although XHTML won't validate correctly if you use anything but lowercase letters for tag and attribute names, most browsers will load the page anyway because they continue to be case insensitive for markup, regardless of the HTML or XHTML version you specify for the page's `DOCTYPE`. The case for property names is not influenced by the case of the markup attribute name. You can validate your HTML at `http://validator.w3.org/`.

Each property determines its own read/write status. Some properties are read-only, whereas you can change others on the fly by assigning a new value to them. For example, to put some new text into a text box object, you assign a string to the object's `value` property:

```
document.forms[0].phone.value = "555-1212";
```

or

```
document.getElementById("myPhoneTextBox").value = "555-1212";
```

When an object contained by the document exists (that is, its HTML is loaded into the document), you can also add one or more custom properties to that object. This can be helpful if you want to associate some additional data with an object for later retrieval. To add such a property, you may simply specify it in the same way that you assign a value to it:

```
document.forms[0].phone.delimiter = "-";
```

or

```
document.getElementById("myPhoneTextBox").value = "-";
```

You'll see later in the chapter that there are other ways to add properties to an object. Any property you set survives as long as the document remains loaded in the window and scripts do not overwrite the object. Be aware, however, that reloading the page usually destroys custom properties.

Object Methods

An object's *method* is a command that a script can give to that object. Some methods return values, but that is not a prerequisite for a method. Also, not every object has methods defined for it. In a majority of cases, invoking a method from a script causes some action to take place. The resulting action may be obvious (such as resizing a window) or something more subtle (such as sorting an array in memory).

All methods have parentheses after them, and methods always appear at the end of an object's reference. When a method accepts or requires parameters, the parameter values go inside the parentheses (with multiple parameters separated by commas).

Although an object has its methods predefined by the object model, you can also assign one or more additional methods to an object that already exists (that is, after its HTML is loaded into the document). To do this, a script in the document (or in another window or frame accessible by the document) must define a JavaScript function and then assign that function to a new property name

of the object. In the following example, written to take advantage of modern browser features, the `fullScreen()` function invokes two `window` object methods. By assigning the function reference to the new `window.maximize` property, we define a `maximize()` method for the `window` object. Thus, a button's event handler can call that method directly.

```
// define the function
function fullScreen()
{
   this.moveTo(0,0);
   this.resizeTo(screen.availWidth, screen.availHeight);
}
// assign the function to a custom property
window.maximize = fullScreen;
...
<!-- invoke the custom method -->
<input type="button" value="Maximize Window" onclick="window.maximize()" />
```

In this last example, DOM0 is used. Using the W3C DOM, the function and its assignment to a custom property will be coded exactly the same. The difference is how the `onclick` event handler is identified. In DOM0, the event is defined directly as an attribute of the object in the HTML: on the `input` tag in the case above. In the W3C DOM the event may be defined within the script, rather than on the HTML element itself:

```
// initialize when the page has loaded
addEvent(window, "load", initialize);

function initialize()
{
   // add an event to the button
   addEvent(document.getElementById("theButton"), click);
}
```

Note

The function to assign event handlers throughout the code in this chapter and much of the book is `addEvent()`, a cross-browser event handler explained in detail in Chapter 32.

The `addEvent()` function is part of the script file `jsb-global.js`, which is located on the accompanying CD-ROM in the `Content/` folder, where it is accessible to all chapters' scripts. ■

Object Event Handlers

An *event handler* specifies how an object reacts to an event that is triggered by a user action (for example, a button click) or a browser action (for example, the completion of a document load). Going back to the earliest JavaScript-enabled browser, event handlers were defined inside HTML tags as extra attributes. They included the name of the attribute, followed by an equal sign (working as an assignment operator) and a string containing the script statement(s) or function(s) to execute when the event occured (see Chapter 7, "Scripts and HTML Documents").

Although event handlers are commonly defined in an object's HTML tag, you also have the power to assign or change an event handler just as you assign or change the property of an object. The value of an event handler property looks like a function definition. For example, given this HTML definition

```
<input type="text" name="entry" onfocus="doIt()" />
```

the value of the object's `onfocus` (all lowercase) property is

```
function onfocus()
{
   doIt();
}
```

You can, however, assign an entirely different function to an event handler by assigning a function reference to the property. Such references don't include the parentheses that are part of the function's definition. (You see this again in Chapter 23 when you assign functions to object properties.)

Using the same text field definition you just looked at, you can assign a different function to the event handler, because, based on user input elsewhere in the document, you want the field to behave differently when it receives the focus. If you define a function like this

```
function doSomethingElse()
{
   //statements
}
```

You can then assign the function to the field with either:

```
document.formName.entry.onfocus = doSomethingElse;
```

or

```
document.getElementById("theEntryID").onfocus = doSomethingElse;
```

Because the new function reference is written in JavaScript, you must observe case for the function name. Additionally, you are best served across all browsers by sticking with all-lowercase event handler names as properties.

If your scripts create new element objects dynamically, you can assign event handlers to these objects by way of event handler properties. For example, the following code uses W3C DOM syntax to create a new button input element and assign an `onclick` event handler that invokes a function defined elsewhere in the script:

```
var newElem = document.createElement("input");
newElem.type = "button";
newElem.value = "Click Here";
newElem.onclick = doIt;
document.getElementById("theFormId").appendChild(newElem);
```

Object Model Smorgasbord

A survey of the entire evolution of scriptable browsers since NN2 and IE3 reveals six distinct DOM families. Even if your job entails developing content for just one current browser version, you may be surprised that family members from more than one DOM inhabit your authoring space.

Studying the evolution of the object model is extremely valuable for newcomers to scripting. It is too easy to learn the latest object model gadgets in your current browser, only to discover that your heroic scripting efforts are lost on earlier browsers accessing your pages. Even if you plan on supporting only modern browsers, a cursory knowledge of object model history is a useful part of your JavaScript knowledge base. Therefore, take a look at the six major object model types and how they came into

being. Table 25-1 lists the object model families and the browser versions that support them. Note that a release number is only listed when the level of support for an object model type leaps forward (for example, going from “limited” support to “most” support). Later in this chapter are some guidelines you can follow to help you choose the object model(s) that best suit your users’ appetites.

TABLE 25-1

Object Model Families

Model	Browser Support
Basic Object Model	NN2+, IE3+, Moz1+, FF2+, Safari1+, Opera+, Chrome+
Basic Plus Images	NN3+, MacIE3.01+, IE4+, Moz1+, FF2+, Safari1+, -, Opera+, Chrome+
NN4 Extensions	NN4
IE4 Extensions	IE4+ (some features in all versions require Win32 OS)
IE5 Extensions	IE5+ (some features in all versions require Win32 OS), Opera 9.62+ (very limited)
W3C DOM (I and II)	IE5+ (partial), Moz1 (most), FF2+ (most), Safari1 (partial), Safari1.3+ (most), Opera9.62+ (most), Chrome1+ (most)

It’s important to realize that even though browsers have come a long way toward providing unified support for web standards, we’re not quite there yet. As of this writing, no current browser fully and accurately supports Levels I and II of the W3C DOM. Firefox 2, Safari 1.3/2, Opera 9, and Chrome 1 have all closed the compatibility gap considerably, but some issues, not severely impacting HTML authoring, remain.

Basic Object Model

The first scriptable browser, Netscape Navigator 2, implemented a very basic DOM. Figure 25-1, which you saw earlier in the chapter, provides a visual guide to the objects that were exposed to scripting. The hierarchical structure starts with the window and drills inward toward the document, forms, and form control elements. A document is a largely immutable page on the screen. Only elements that are by nature interactive — links and form elements, such as text fields and buttons — are treated as objects with properties, methods, and event handlers.

The heavy emphasis on form controls opened numerous possibilities that were radical ideas at the time. Because a script could inspect the values of form controls, forms could be prevalidated on the client. If the page included a script that performed some calculations, data entry and calculated results were displayed via editable text fields.

Additional objects that exist outside the document — `window`, `history`, and `location` objects — provide scriptable access to simple yet practical properties of the browser that loads the page. The most global view of the environment is the `navigator` object, which includes properties about the browser brand and version.

When Internet Explorer 3 arrived on the scene, the short life of Navigator 2 was nearing its end. Even though NN3 was already widely available in prerelease form, Internet Explorer 3 implemented the basic object model from NN2 (plus one `window` object property from NN3). Therefore, despite the browser version number discrepancy, NN2 and IE3 were essentially the same with respect to their DOMs. For a brief moment in Internet Time, there was nearly complete harmony between Microsoft and Netscape DOMs — albeit at a very simple level.

Basic Object Model Plus Images

A very short time after Internet Explorer 3 was released, Netscape released Navigator 3 with an object model that built upon the original version. A handful of existing objects — especially the `window` object — gained new properties, methods, and/or event handlers. Scripts could also communicate with Java applets as objects. But the biggest new object on the scene was the `Image` object, along with the array of image objects exposed to the `document` object.

Most of the properties for a Navigator 3 image object gave read-only access to values typically assigned to attributes in the `<img>` tag. But you could modify one property — the `src` property — after the page loaded. Scripts could swap out images within the fixed image rectangle. Although these new image objects didn't have mouse-related event handlers, nesting an image inside a link (which had `onmouseover` and new `onmouseout` event handlers) let scripters implement image rollovers to liven up a page.

As more new scripters investigated the possibilities of adding JavaScript to their pages, frustration ensued when the image swapping they implemented for Navigator 3 failed to work in Internet Explorer 3. Although you could easily script around the lack of an image object to prevent script errors in Internet Explorer 3, the lack of this cool page feature disappointed many. Had they also taken into account the installed base of Navigator 2 in the world, they would have been disappointed there, too. To confuse matters even more, the Macintosh version of Internet Explorer 3.01 (the second release of the Internet Explorer for Mac browser) implemented scriptable image objects.

Despite these rumblings of compatibility problems to come, the object model implemented in Navigator 3 eventually became the baseline reference for future DOMs. With few exceptions, code written for this object model runs on all browsers from Navigator 3 and Internet Explorer 4, on through the latest versions of both brands and other modern browsers.

Navigator 4–Only Extensions

The next browser released to the world was Netscape Navigator 4. Numerous additions to the existing objects put more power into the hands of scripters. You could move and resize browser windows within the context of script-detectable `screen` object properties (for example, how big the user's screen was). Two concepts that represented new thinking about the object model were an enhanced event model and the layer object.

Event capture model

Navigator 4 added many new events to the repertoire. Keyboard events and more mouse events (`onmousedown` and `onmouseup`) allowed scripts to react to more user actions on form control elements and links. All of these events worked as they did in previous object models in which event

handlers were typically assigned as attributes to an element's tag (although you could also assign event handlers as properties in script statements). To facilitate some of the Dynamic HTML (DHTML) potential in the rest of the Navigator 4 object model, the event model was substantially enhanced.

At the root of the system is the idea that when a user performs some physical action on an event-aware object (for example, clicking a form button), the event reaches that button from the top down, through the document object hierarchy. If you have multiple objects that share an event handler, it may be more convenient to capture that event in just one place — the `window` or `document` object level — rather than assigning the same event handler to all the elements. The default behavior of Navigator 4 allowed the event to reach the target object, just as it had in earlier browsers. But you could also turn on *event capture* in the `window`, `document`, or layer object. When captured, the event could be handled at the upper level, preprocessed before being passed onto its original target, or redirected to another object altogether.

Whether or not you capture events, the Navigator 4 event model produces an `event` object (with a lowercase `e` to distinguish it from the static `Event` object) for each event. That object contains properties that reveal more information about the specific event, such as the keyboard character pressed for a keyboard event or the position of a click event on the page. Any event handler can inspect `event` object properties to learn more about the event, and then process the event accordingly.

Layers

Perhaps the most radical addition to the Navigator 4 object model was a new object that reflected an entirely new HTML element: the `layer` element. A *layer* is a container that is capable of holding its own HTML document, yet it exists in a plane in front of the main document. You can move, size, and hide a layer under script control. This new element allowed, for the first time, overlapping elements in an HTML page.

To accommodate the layer object in the document object hierarchy, Netscape defined a nesting hierarchy such that a layer was contained by a document. As a result, the `document` object acquired a property (`document.layers`) that was an array of layer objects in the document. This array exposed only the first level of layer(s) in the current `document` object.

Each layer had its own `document` object because each layer could load an external HTML document if desired. As a positionable element, a layer object had numerous properties and methods that allowed scripts to move, hide, show, and change its stacking order.

Unfortunately for Netscape, the W3C did not agree to make the `<layer>` tag part of the HTML 4 specification. As such, it is an orphan element that exists only in Navigator 4 (not implemented in Moz1 or later). The same goes for the scripting of the layer object and its nested references.

Internet Explorer 4+ Extensions

Microsoft broke important new ground with the release of Internet Explorer 4, which came several months after the release of Navigator 4. The main improvements were in the exposure of all HTML elements, scripted support of Cascading Style Sheets (CSS), and a new event model. Some other additions were available only on Windows 32-bit operating system platforms.

HTML element objects

The biggest change to the object model world was that every HTML element became a scriptable object, while still supporting the original object model. Microsoft invented the `document.all` array

(also called a *collection*). This array contains references to every element in the document, regardless of element nesting. If you assign a unique identifier (name) to the `id` attribute of an element, you can reference the element by the following syntax:

```
document.all.elementID
```

In most cases, you can also drop the `document.all.` part of the reference and begin with only the element ID.

Every element object has an entirely new set of properties and methods that give scripters a level of control over document content unlike anything seen before. These properties and methods are explored in more detail in Chapter 26, "Generic HTML objects," but several groups of properties deserve special mention here.

Four properties (`innerHTML`, `innerText`, `outerHTML`, and `outerText`) provide read/write access to the actual content within the body of a document. This means that you no longer have to use text boxes to display calculated output from scripts. You can modify content inside paragraphs, table cells, or anywhere, on the fly. The browser's rendering engine immediately reflows a document when the dimensions of an element's content change. That feature puts the *Dynamic* in *Dynamic HTML*. To those of us who scripted the static pages of earlier browsers, this feature — taken for granted today — was nothing short of a revelation.

The series of offset properties are related to the position of an element on the page. These properties are distinct from the kind of positioning performed by CSS. Therefore, you can get the dimensions and location of any element on the page, making it easier to move positionable content atop elements that are part of the document and that may appear in various locations due to the browser window's current size.

Finally, the `style` property is the gateway to CSS specifications defined for the element. It is important that the script can modify the numerous properties of the `style` object. Therefore, you can modify font specifications, colors, borders, and the positioning properties after the page loads. The dynamic reflow of the page takes care of any layout changes that the alteration requires (for example, adjusting to a bigger font size).

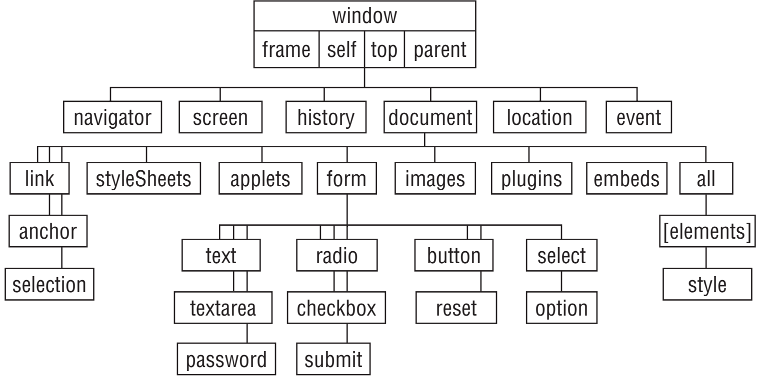
Element containment hierarchy

Although Internet Explorer 4 still recognizes the element hierarchy of the original object model (see Figure 25-1), the DOM for Internet Explorer 4 does not extend this kind of hierarchy fully into other elements. If it did, it would mean that `td` elements inside a table might have to be addressed via its next outer `tr` or `table` element (just as a form control element must be addressed through its containing `form` element). Figure 25-2 shows how all HTML elements are grouped under the `document` object. The `document.all` array flattens the containment hierarchy, as far as referencing objects goes. A reference to the most deeply nested TD element is still `document.all.cellID`. The highlighted pathway from the `window` object is the predominant reference path used when working with the Internet Explorer 4 document object hierarchy.

Element containment in Internet Explorer 4, however, is important for other reasons. Because an element can inherit some style sheet attributes from an element that contains it, you should devise a document's HTML by embedding every piece of content in a container. Paragraph elements are text containers (with start and end tags), not tall line breaks between text chunks. Internet Explorer 4 introduced the notion of a parent-child relationship between a container and elements nested within it. Also, the position of an element may be calculated relative to the position of its next outermost positioning context.

FIGURE 25-2

The Internet Explorer 4 document object hierarchy.



The bottom line here is that element containment doesn't have anything to do with object references (like the original object model). It has everything to do with the *context* of an element relative to the rest of the page's content.

Cascading Style Sheets

By arriving a bit later to market with its version 4 browser than Netscape, Microsoft benefited from having the CSS Level 1 specification more fully developed before the browser's release. Therefore, the implementation is far more complete than that of Navigator 4 (though it is not 100 percent compatible with the standard).

The scriptability of style sheet properties is a bit at odds with the first-generation CSS specification, which seemed to ignore the potential of scripting styles with JavaScript. Many CSS attribute names are hyphenated words (for example, `text-align`, `z-index`), even though hyphens are not allowed in identifier names in JavaScript. This necessitated conversion of the multiword CSS attribute names to camelCase JavaScript property names. Therefore, `text-align` becomes `textAlign`, and `z-index` becomes `zIndex`. You can access all of these properties through an element's `style` property:

```
document.all.elementID.style.stylePropertyName
```

or

```
document.getElementById("myElement").style.stylePropertyName
```

One byproduct of the scriptability of style sheets in Internet Explorer 4 and later is what you might call the *phantom page syndrome*. This occurs when the layout of a page is handled after the primary HTML for the page has downloaded to the browser. As the page loads, not all content may be visible, or it may be in a visual jumble. An `onload` event handler in the page then triggers scripts to set styles or content for the page. Elements jump around to get to their final resting places. This may be disconcerting to some users who at first see a link to click, but by the time the cursor reaches the click location, the page has reflowed, thereby moving the link somewhere else on the page.

Note

For Internet Explorer users with 32-bit Windows operating systems, Internet Explorer 4 includes some extra features in the object model that can enhance presentations. *Filters* are style sheet additives that offer a variety of visual effects on body text. For example, you can add a drop shadow or a glowing effect to text simply by applying filter styles to the text, or you can create the equivalent of a slide presentation by placing the content of each slide in a positioned `div` element. Although filters follow the CSS syntax, they are not part of the W3C specification. ■

Event bubbling

Just as Netscape invented an event model for Navigator 4, so did Microsoft for Internet Explorer 4. Unfortunately for cross-browser scripters, the two event models are quite different. Instead of events trickling down the hierarchy to the target element, an Internet Explorer event starts at the target element and, unless instructed otherwise, bubbles up through the element containment hierarchy to reach the `window` object eventually. At any object along the way, an event handler can perform additional processing on that event, if desired. Therefore, if you want a single event handler to process all click events for the page, assign the event handler to the `body` or `window` object so the events reach those objects (provided that the event bubbling isn't canceled by some other object along the containment hierarchy).

Internet Explorer also has an `event` object (a property of the `window` object) that contains details about the event, such as the key pressed for a keyboard event and the location of a mouse event. Names for these properties are entirely different from the event object properties of Navigator 4.

Despite what seems like incompatible, if not completely opposite, event models in Navigator 4 and Internet Explorer 4, you can make a single set of scripts handle events in both browsers; see Chapter 32, "Event Objects" and Chapter 59, "Application: Cross-Browser DHTML Map Puzzle" (on the CD-ROM) for examples. The Internet Explorer 4 event model continues to be the only model supported by Internet Explorer through version 7.

Internet Explorer 5+ Extensions

With the release of Internet Explorer 5, Microsoft built more onto the proprietary object model it launched with Internet Explorer 4. Although the range of objects remained pretty much the same, the number of properties, methods, and event handlers for the objects increased dramatically. Some of those additions were added to meet some of the specifications of the W3C DOM (discussed in the next section), occasionally causing a bit of incompatibility with Internet Explorer 4. But Microsoft also pushed ahead with features only for Windows users that may not necessarily become industry standards: DHTML behaviors and HTML applications.

A *DHTML behavior* is a chunk of script — saved as an external file — that defines some action (usually, a change of one or more style properties) that you can apply to any kind of element. The goal is to create a reusable component that you can load into any document whose elements require that behavior. As an example of a DHTML behavior, you can define a behavior that turns an element's text to red whenever the cursor rolls atop it, and reverts the text to black when the cursor rolls out. When you assign the behavior to an element in the document (through CSS-like rule syntax), the element picks up that behavior and responds to the user accordingly. You can apply that same behavior to any element(s) in the document.

Cross-Reference

You can see an example of a DHTML behavior in Chapter 26, in the description of the `addBehavior()` method; you can read an extended discussion in Chapter 50 (on the CD-ROM). ■

HTML applications (*HTAs,* in Microsoft parlance) are HTML files that include an XML element known as the `hta:application` element. You can download an HTA to Internet Explorer 5 or later from the server as though it were a web page (although its file extension is `.hta` rather than `.htm` or `.html`). A user can also install an HTA on a client machine so that it behaves very much like an application, with a desktop icon and significant control over the look of the window. HTAs are granted greater security privileges on the client so that this application can behave more like a regular program. In fact, you can elect to turn off the system menu bar and use DHTML techniques to build your own menu bar for the application. Implementation details of HTAs are beyond the scope of this book, but you should be aware of their existence. More information is available at `http://msdn.microsoft.com`.

The W3C DOM

Conflicting browser object models from Netscape and Microsoft made life difficult for developers. Scripters craved a standard that would serve as a common denominator, much as HTML and CSS standards did for content and styles. The W3C took up the challenge of creating a DOM standard: the W3C DOM.

The charter of the W3C DOM working group was to create a DOM that could be applied to both HTML and XML documents. Because an XML document can have tags of virtually any name (as defined by a Document Type Definition or XML Schema), it has no intrinsic structure or fixed vocabulary of elements, as an HTML document does. As a result, the DOM specification had to accommodate the known structure of HTML (as defined in the HTML 4 specification), as well as the unknown structure of an XML document.

To make this work effectively, the working group divided the DOM specification into two sections. The first, called the *Core DOM,* defines specifications for the basic document structure that HTML and XML documents share. This includes notions of a document containing elements that have tag names and attributes; an element is capable of containing zero or more other elements. The second part of the DOM specification addresses the elements and other characteristics that apply only to HTML. The HTML portion inherits all the features of the Core DOM, while providing a measure of backward compatibility to object models already implemented in legacy browsers and providing a framework for new features.

It is important for veteran scripters to recognize that the W3C DOM does not specify all features from existing browser object models. Many features of the Internet Explorer 4 (and later) object model are not part of the W3C DOM specification. This means that if you are comfortable in the Internet Explorer environment and wish to shift your focus to writing for the W3C DOM spec, you have to change some practices, as highlighted in this chapter. In many respects, especially with regard to DHTML applications, the W3C DOM is an entirely new DOM with new concepts that you must grasp before you can successfully script in the environment.

By the same token, you should be aware that whereas Mozilla-based browsers go to great lengths to implement all of DOM Level 1 and most of Level 2, Microsoft (for whatever reason) features only a partial implementation of the W3C DOM through Internet Explorer 5.5. Although IE6 and

later versions implement more W3C DOM features, some important parts — notably, W3C DOM events — are missing. Other modern browsers, such as Chrome 1+, Safari 1.3+, and Opera 9+, provide comprehensive W3C DOM support and have largely closed the gap to compete with Mozilla in terms of supporting the W3C DOM.

DOM levels

Like most W3C specifications, one version is rarely enough. The job of the DOM working group was too large to be swallowed whole in one sitting. Therefore, the DOM is a continually evolving specification. The timeline of specification releases rarely coincides with browser releases. Therefore, it is very common for any given browser release to include only some of the most recent W3C version.

The first formal specification, DOM Level 1, was released well after NN4 and IE4 shipped. The HTML portion of Level 1 includes the so-called DOM Level 0 (there is no published standard by that name). This is essentially the object model as implemented in Navigator 3 (and for the most part in Internet Explorer 3, plus image objects). Perhaps the most significant omission from Level 1 is an event model (it ignores even the simple event model implemented in NN2 and IE3).

DOM Level 2 builds on the work of Level 1. In addition to several enhancements of both the Core and HTML portions of Level 1, Level 2 adds significant new sections (published as separate modules) on the event model, ways of inspecting a document's hierarchy, XML namespaces, text ranges, style sheets, and style properties. Some modules of the Level 3 DOM have reached Recommendation status. Although IE is still a ways off from implementation to any significant degree, the other major browsers have at least partially implemented some of the modules, including some that are still in working draft status.

What stays the same

By adopting DOM Level 0 as the starting point of the HTML portion of the DOM, the W3C provided a way for a lot of existing script code to work even in a W3C DOM-compatible browser. Every object you see in the original object model, starting with the `document` object (see Figure 25-1) plus the image object, are in DOM Level 0. Almost all of the same object properties and methods are also available.

More important, when you consider the changes to referencing other elements in the W3C DOM, we're lucky that the old ways of referencing objects — such as forms, form control elements, and images — still work. Had the working group started with a clean slate, it is unlikely that the `document` object would have been given properties consisting of arrays of forms, links, and images.

The only potential problems you could encounter with your existing code have to do with a handful of properties that used to belong to the `document` object. In the new DOM, four style-related properties of the `document` object (`alinkColor`, `bgColor`, `linkColor`, and `vlinkColor`) become properties of the `body` object (referenced as `document.body`). In addition, the three link color properties pick up new names in the process (`aLink`, `link`, and `vLink`). It appears, however, that for now, IE6 and Moz1 maintain backward compatibility with the older document object color properties.

Also note that the DOM specification concerns itself only with the document and its content. Objects such as `window`, `navigator`, and `screen` are not part of the DOM specification through Level 2. Scripters are still at the mercy of browser makers for compatibility in these areas.

What isn't available

As mentioned earlier, the W3C DOM is not simply a restatement of existing browser specifications. Many convenience features of the Internet Explorer and Netscape Navigator object models do not appear in the W3C DOM. If you developed DHTML content in Internet Explorer 4 or later or in Navigator 4, you had to learn how to get along without some of these conveniences.

The Navigator 4 experiment with the `<layer>` tag was not successful in the W3C process. As a result, both the tag and the scripting conventions surrounding it do not exist in the W3C DOM. To some scripters' relief, the `document.layerName` referencing scenario (even more complex with nested layers) disappears from the object model. A positioned element is treated as just another element with some special style sheet attributes that enable you to move it anywhere on the page, stack it amid other positioned elements, or hide it from view.

Among popular Internet Explorer 4+ features missing from the W3C DOM are the `document.all` collection of HTML elements and four element properties that facilitate dynamic content: `innerHTML`, `innerText`, `outerHTML`, and `outerText`. W3C provides a new way for acquiring an array of all elements in a document, but generating HTML content to replace existing content or to be inserted in a document requires a tedious sequence of statements (see the section "New DOM concepts," later in this chapter). All the new browsers, however, have implemented the `innerHTML` property for HTML element objects, and with the exception of Firefox, have implemented the other three properties.

New HTML practices

Exploitation of DHTML possibilities in the W3C DOM relies on modern HTML practices that by now have been adopted by the majority of HTML authors. At the core of these practices (espoused by the HTML 4 specification) is making sure that all content is within an HTML container of some kind. Therefore, instead of using the `<p>` tag as a separator between blocks of running text, surround each paragraph of the running text with a `<p>...</p>` tag set. If you don't do it, the browser treats each `<p>` tag as the beginning of a paragraph and ends the paragraph element just before the next `<p>` tag or other block-level element.

Although browsers continue to accept the omission of certain end tags (for `td`, `tr`, and `li` elements, for instance) for backward compatibility, it is best to get into the habit of supplying these end tags if for no other reason than that they help you visualize where an element's sphere of influence truly begins and ends. Applying this as a best practice also means your code will validate.

Any element that you intend to script — whether to change its content or its style — should have an identifier assigned to the element's `id` attribute. The identifier should be a unique value within the context of the current web page. Form control elements still require `name` attributes if you submit the form content to a server. But you can freely assign the same or a different identifier value to a control's `id` attribute. Scripts can use either the `id` or the `document.formReference.elementName` reference to reach a control object. Identifiers are essentially the same as the values you assign to the `name` attributes of form and form input elements. Following the same rules for the `name` attribute value, an `id` identifier must be a single word (no whitespace); it cannot begin with a numeral (to prevent conflicts in JavaScript); and it should avoid punctuation symbols except for the underscore character.

New DOM concepts

With the W3C DOM comes several concepts that may be new to you unless you have worked extensively with the terminology of tree hierarchies. Concepts that have the most impact on your scripting are new ways of referencing elements and nodes.

Element referencing

Script references to objects in the DOM Level 0 are observed in the W3C DOM for backward compatibility. Therefore, a form input element whose `name` attribute is assigned the value `userName` is addressed just as it always is

```
document.forms[0].userName
```

or

```
document.formName.userName
```

But because all elements of a document are exposed to the `document` object, you can use the `document` object method designed to access any element whose ID is assigned. The method is `document.getElementById()`, and the sole parameter is a string version of the identifier of the object whose reference you want to get. To help put this in context with what you may have used with the Internet Explorer 4 object model, consider the following HTML paragraph tag:

```
<p id="myParagraph">...</p>
```

In Internet Explorer 4 or later, you can reference this element with

```
var elem = document.all.myParagraph;
```

Although the `document.all` collection is not implemented in the W3C DOM, the `document` object method (available in Internet Explorer 5 and later, Mozilla, Safari, and others) `getElementById()` enables you to access any element by its ID:

```
var elem = document.getElementById("myParagraph");
```

This method is considered the appropriate technique for referencing an element based upon its ID. Unfortunately for scripters, this method is difficult to type because it is case sensitive, so watch out for that ending lowercase *d*.

A hierarchy of nodes

The issue surrounding containers (described earlier) comes into play for the underlying architecture of the W3C DOM. Every element or free-standing chunk of text in an HTML (or XML) document is an object that is contained by its next outermost container. Let's look at a simple HTML document to see how this system works. Listing 25-1 is formatted to show the containment hierarchy of elements and string chunks.

LISTING 25-1

A Simple HTML Document

```
<html>
   <head>
      <title>
         A Simple Page
      </title>
```

continued

LISTING 25-1 *(continued)*

```
   </head>

   <body>
      <p id="paragraph1">
         This is the
         <em id="emphasis1">
            one and only
         </em>
         paragraph on the page.
      </p>
   </body>
</html>
```

What you don't see in the listing is a representation of the `document` object. The `document` object exists automatically when this page loads into a browser. It is important that the `document` object encompasses everything you see in Listing 25-1. Therefore, the `document` object has a single nested element: the `html` element. The `html` element in turn has two nested elements: `head` and `body`. The `head` element contains the `title` element, whereas the `title` element contains a chunk of text. Down in the `body` element, the `p` element contains three pieces: a string chunk, the `em` element, and another string chunk.

According to W3C DOM terminology, each container, stand-alone element (such as a `br` element), or text chunk is known as a *node* — a fundamental building block of the W3C DOM. Nodes have parent-child relationships when one container holds another. As in real life, parent-child relationships extend only between adjacent generations, so a node can have zero or more children. However, the number of third-generation nodes further nested within the family tree does not influence the number of children associated with a parent. Therefore, in Listing 25-1, the `html` node has two child nodes: `head` and `body`, which are *siblings* that have the same parent. The `body` element has one child (`p`), even though that child contains three children (two text nodes and an `em` element node).

If you draw a hierarchical tree diagram of the document in Listing 25-1, it should look like the illustration in Figure 25-3.

FIGURE 25-3

Tree diagram of nodes for the document in Listing 25-1.

```
document
+--<html>
    +--<head>
     |   +--<title>
     |       +--"A Simple Page"
     +--<body>
         +--<p ID="paragraph1">
             +--"This is the  "
             +--<em ID="emphasis1">
             |   +--"one and only"
             +--" paragraph on the page."
```

Note

If the document's source code contains a Document Type Definition (in a `DOCTYPE` element) above the `<html>` tag, the browser treats that `DOCTYPE` node as a sibling of the HTML element node. In that case, the root `document` node contains two child nodes. ■

The W3C DOM (through Level 2) defines 12 different types of nodes, 7 of which have direct application in HTML documents. These seven types of nodes appear in Table 25-2; the rest apply to XML. Of the 7 types, the three most common are the document, element, and text types. All W3C DOM browsers (including Internet Explorer 5 and later, Mozilla, Safari, and others) implement the three common node types, whereas Mozilla implements all of them, IE6 implements all but one, and Safari 1 implements all but two.

Applying the node types of Table 25-2 to the node diagram in Figure 25-3, you can see that the simple page consists of one document node, six element nodes, and four text nodes.

TABLE 25-2

W3C DOM HTML-Related Node Types

Type	Number	nodeName	nodeValue	Description	IE	Moz	Safari
Element	1	*tag name*	`Null`	Any HTML or XML tagged element	6+	1+	1+
Attribute	2	*attribute name*	*attribute value*	A name-value attribute pair in an element	6+	1+	1+
Text	3	`#text`	*text content*	A text fragment contained by an element	6+	1+	1+
Comment	8	`#comment`	*comment text*	HTML comment	6+	1+	4+
Document	9	`#document`	`Null`	Root `document` object	6+	1+	1+
DocumentType	10	`DOCTYPE`	`Null`	DTD specification	No	1+	4+
Fragment	11	`#document-fragment`	`Null`	Series of one or more nodes outside the document	6+	1+	1+

Node properties

A node has many properties, most of which are references to other nodes related to the current node. Table 25-3 lists all properties shared by all node types in DOM Level 2.

Note

You can find all the properties shown in Table 25-3, that also show themselves to be implemented in Internet Explorer 6+ or Moz1+, in Chapter 26, in the listing of properties that all HTML element objects have in common. That's because an HTML element, as a type of node, inherits all of the properties of the prototypical node. ■

TABLE 25-3

Node Object Properties (W3C DOM Level 2)

Property	Value	Description	IE6Win+	IE5Mac+	Moz1	Safari1
`nodeName`	String	Varies with node type (see Table 25-2)	Yes	Yes	Yes	Yes
`nodeValue`	String	Varies with node type (see Table 25-2)	Yes	Yes	Yes	Yes
`nodeType`	Integer	Constant representing each type	Yes	Yes	Yes	Yes
`parentNode`	Object	Reference to next outermost container	Yes	Yes	Yes	Yes
`childNodes`	Array	All child nodes in source order	Yes	Yes	Yes	Yes
`firstChild`	Object	Reference to first child node	Yes	Yes	Yes	Yes
`lastChild`	Object	Reference to last child node	Yes	Yes	Yes	Yes
`previous Sibling`	Object	Reference to sibling node up in source order	Yes	Yes	Yes	Yes
`nextSibling`	Object	Reference to sibling node next in source order	Yes	Yes	Yes	Yes
`attributes`	NodeMap	Array of attribute nodes	Yes	Some	Yes	Yes
`ownerDocument`	Object	Containing document object	Yes	Yes	Yes	Yes
`namespaceURI`	String	URI to namespace definition (element and attribute nodes only)	Yes	No	Yes	Yes
`Prefix`	String	Namespace prefix (element and attribute nodes only)	Yes	No	Yes	Yes
`localName`	String	Applicable to namespace-affected nodes	Yes	No	Yes	Yes

To help you see the meanings of the key node properties, Table 25-4 shows the property values of several nodes in the simple page shown in Listing 25-1. For each node column, find the node in Figure 25-3, and then follow the list of property values for that node, comparing the values against the actual node structure in Figure 25-3.

TABLE 25-4

Properties of Selected Nodes for a Simple HTML Document

Properties	Nodes			
	document	html	p	"one and only"
nodeType	9	1	1	3
nodeName	#document	html	p	#text
nodeValue	Null	null	null	"one and only"
parentNode	Null	document	body	em
previousSibling	Null	null	null	null
nextSibling	Null	null	null	null
childNodes	Html	head body	"This is the" em "paragraph on the page."	(none)
firstChild	Html	head	"This is the"	null
lastChild	Html	body	"paragraph on the page."	null

The `nodeType` property is an integer that is helpful in scripts that iterate through an unknown collection of nodes. Most content in an HTML document is of type 1 (an HTML element) or 3 (a text node), with the outermost container, the document, of type 9. A node's `nodeName` property is either the name of the node's tag (for an `HTML` element) or a constant value (preceded by a # [hash mark], as shown in Table 25-2). And, while it may surprise some, the `nodeValue` property is `null` except for the text node type, in which case the value is the actual string of text of the node. In other words, for `HTML` elements, the W3C DOM does not expose a container's HTML as a string.

The Object-Oriented W3C DOM

If you are familiar with concepts of object-oriented (OO) programming, you will appreciate the OO tendencies in the way the W3C defines the DOM. The `Node` object includes sets of properties (see Table 25-3) and methods (see Table 25-5) that are inherited by every object based on the `Node`. Most of the objects that inherit the `Node`'s behavior have their own properties and/or methods that define their specific behaviors. Figure 25-4 shows (in W3C DOM terminology) the inheritance tree from the `Node` root object. Most items are defined in the Core DOM, whereas items shown in boldface are from the HTML DOM portion.

continued

continued

FIGURE 25-4

W3C DOM Node object inheritance tree.

```
Node
+--Document
|   +--HTMLDocument
+--CharacterData
|   +--Text
|   |  +--CDATASection
|   +--Comment
+--Attr
+--Element
|   +--HTMLElement
|        +--  (Each specific HTML element)
+--DocumentType
+--DocumentFragment
+--Notation
+--Entity
+--Entity Reference
+--ProcessingInstruction
```

You can see from the preceding figure that individual HTML elements inherit properties and methods from the generic HTML element, which inherits from the Core `Element` object, which in turn inherits from the basic `Node`.

It isn't important to know the `Node` object inheritance to script the DOM. But it does help in understanding the ECMA Script Language Binding appendix of the W3C DOM recommendation, as well as how a simple element object winds up with so many properties and methods associated with it.

It is doubtful that you will use all of the relationship-oriented properties of a node, primarily because there is some overlap in how you can reach a particular node from any other. The `parentNode` property is important because it is a reference to the current node's immediate container. Although the `firstChild` and `lastChild` properties point directly to the first and last children inside a container, most scripts generally use the `childNodes` property with array notation inside a `for` loop to iterate through child nodes. If there are no child nodes, the `childNodes` array has a length of zero.

Node methods

Actions that modify the HTML content of a node in the W3C DOM world primarily involve the methods defined for the prototype `Node`. Table 25-5 shows the methods and their support in the W3C DOM–capable browsers.

The important methods for modifying content are `appendChild()`, `insertBefore()`, `removeChild()`, and `replaceChild()`. Note, however, that all of these methods assume that the point of view for the action is from the parent of the nodes being affected by the methods.

For example, to delete an element (using removeChild()), you don't invoke that method on the element being removed, but on its parent element. This leaves open the possibility of creating a library of utility functions that obviate having to know too much about the precise containment hierarchy of an element. A simple function that lets a script appear to delete an element actually does so from its parent:

```
function removeElement(elemID)
{
   var elem = document.getElementById(elemID);
   elem.parentNode.removeChild(elem);
}
```

TABLE 25-5

Node Object Methods (W3C DOM Level 2)

Method	Description	IE5+	Moz1	Safari 1
appendChild(newChild)	Adds child node to end of current node	Yes	Yes	Yes
cloneNode(deep)	Grabs a copy of the current node (optionally with children)	Yes	Yes	Yes
hasChildNodes()	Determines whether current node has children (Boolean)	Yes	Yes	Yes
insertBefore(new, ref)	Inserts new child in front of another child	Yes	Yes	Yes
removeChild(old)	Deletes one child	Yes	Yes	Yes
replaceChild(new, old)	Replaces an old child with a new one	Yes	Yes	Yes
isSupported(feature, version)	Determines whether the node supports a particular feature	No	Yes	Yes

If this seems like a long way to go to accomplish the same result as setting the outerHTML property of an Internet Explorer 4 or later object to empty, you are right. Although some of this convolution makes sense for XML, the W3C working group doesn't seem to have HTML scripters' best interests in mind, unfortunately. All is not lost, however, as you see later in this chapter.

Generating new node content

The final point about the node structure of the W3C DOM focuses on the similarly gnarled way scripters must go about generating content that they want to add or replace on a page. For text-only changes (for example, the text inside a table cell), there is both an easy and a hard way to perform the task. For HTML changes, there is only the hard way (plus a handy workaround discussed later). Let's look at the hard way first and then pick up the easy way for text changes.

To generate a new node in the DOM, you look to the variety of methods that are defined for the Core DOM's `document` object (and are therefore inherited by the HTML `document` object). A node creation method is defined for nearly every node type in the DOM. The two important ones for HTML documents are `createElement()` and `createTextNode()`. The first generates an element with whatever tag name (string) you pass as a parameter; the second generates a text node with whatever text you pass.

When you first create a new element, it exists only in the browser's memory and not as part of the document containment hierarchy. Moreover, the result of the `createElement()` method is a reference to an empty element, except for the name of the tag. For example, to create a new `p` element, use

```
var newElem = document.createElement("p");
```

The new element has no ID, attributes, or any content. To assign some attributes to this element, you can use the `setAttribute()` method (a method of every element object) or assign a value to the object's corresponding property. For example, to assign an identifier to the new element, use either

```
newElem.setAttribute("id", "newP");
```

or

```
newElem.id = "newP";
```

Both ways are perfectly legal. Even though the element has an ID at this point, it is not yet part of the document, so you cannot retrieve it via the `document.getElementById()` method.

To add some content to the paragraph, you next generate a text node as a separate object:

```
var newText = document.createTextNode("This is the second paragraph.");
```

Again, this node is just sitting around in memory waiting for you to apply it as a child of some other node. To make this text the content of the new paragraph, you can append the node as a child of the paragraph element that is still in memory:

```
newElem.appendChild(newText);
```

If you were able to inspect the HTML that represents the new paragraph element, it would look like the following:

```
<p id="newP">This is the second paragraph.</p>
```

The new paragraph element is ready for insertion into a document. Using the document shown in Listing 25-1, you can append it as a child of the `body` element:

```
document.body.appendChild(newElem);
```

At last, the new element is part of the document containment hierarchy. Now you can reference it just like any other element in the document.

Replacing node content

The addition of the paragraph shown in the last section requires a change to a portion of the text in the original paragraph (since the first paragraph is no longer the one and only paragraph on the page). As mentioned earlier, you can perform text changes via the `replaceChild()` method or by assigning new text to a text node's `nodeValue` property. Let's see how each approach works to change the text of the first paragraph's `em` element from one and only to first.

To use `replaceChild()`, a script first must generate a valid text node with the new text:

```
var newText = document.createTextNode("first");
```

The next step is to use the `replaceChild()` method. But recall that the point of view for this method is the parent of the child being replaced. The child here is the text node inside the `em` element, so you must invoke the `replaceChild()` method on the `em` element. Also, the `replaceChild()` method requires two parameters. The first parameter is the new node; the second is a reference to the node to be replaced. Because the script statements get pretty long with the `getElementById()` method, an intermediate step grabs a reference to the text node inside the `em` element:

```
var oldChild = document.getElementById("emphasis1").childNodes[0];
```

Now the script is ready to invoke the `replaceChild()` method on the `em` element, swapping the old text node with the new:

```
document.getElementById("emphasis1").replaceChild(newText, oldChild);
```

If you want to capture the old node before it disappears, be aware that the `replaceChild()` method returns a reference to the replaced node (which is only in memory at this point and not part of the document node hierarchy). You can assign the method statement to a variable and use that old node somewhere else, if needed.

This may seem like a long way to go; it is, especially if the HTML you are generating is complex. Fortunately, you can take a simpler approach for replacing text nodes. All it requires is a reference to the text node being replaced. You can assign that node's `nodeValue` property its new string value:

```
document.getElementById("emphasis1").childNodes[0].nodeValue = "first";
```

When an element's content is entirely text (for example, a table cell that already has a text node in it), this is the most streamlined way to swap text on the fly using W3C DOM syntax. This doesn't work for the creation of the second paragraph text earlier in this chapter because the text node did not exist yet. The `createTextNode()` method had to create it explicitly.

Also remember that a text node does not have any inherent style associated with it. The style of the containing HTML element governs the style of the text. If you want to change not only the text node's text, but also how it looks, you have to modify the `style` property of the text node's parent element. Browsers that perform these kinds of content swaps and style changes automatically reflow the page to accommodate changes in the size of the content.

To summarize, Listing 25-2 is a live version of the modifications made to the original document shown in Listing 25-1. The new version includes a button and a script that makes the changes described throughout this discussion of nodes. Reload the page to start over.

LISTING 25-2

Adding/Replacing DOM Content

```
<html>
<head>
   <title>A Simple Page</title>
   <script type="text/javascript">
   function modify()
   {
      var newElem = document.createElement("p");
      newElem.id = "newP";
      var newText = document.createTextNode("This is the second paragraph.");
      newElem.appendChild(newText);
      document.body.appendChild(newElem);
      document.getElementById("emphasis1").childNodes[0].nodeValue = "first";
   }
   </script>
</head>

<body>
   <button onclick="modify()">Add/Replace Text</button>

   <p id="paragraph1">This is the <em id="emphasis1">one and
   only</em> paragraph on the page.</p>
</body>
</html>
```

Cross-Reference

Chapter 26 details node properties and methods that are inherited by all HTML elements. Most are implemented in all modern W3C DOM browsers. Also look to the reference material for the `document` object in Chapter 29 for other valuable W3C DOM methods. ■

A de facto standard: innerHTML

Microsoft was the first to implement the `innerHTML` property of all element objects starting with Internet Explorer 4. Although the W3C DOM has not supported this property, scripters frequently find it more convenient to modify content dynamically by way of a string containing HTML markup than by creating and assembling element and text nodes. As a result, most modern W3C DOM browsers, including Moz1 and Safari 1, support the read/write `innerHTML` property of all element objects as a de facto standard.

When you assign a string containing HTML markup to the `innerHTML` of an existing element, the browser automatically inserts the newly rendered elements into the document node tree. You may also use `innerHTML` with unmarked text to perform the equivalent of the Internet Explorer–only `innerText` property.

Despite the apparent convenience of the innerHTML property compared with the step-by-step process of manipulating element and text node objects, browsers operate on nodes much more efficiently than on assembly of long strings. This is one case where less JavaScript code does not necessarily translate to greater efficiency.

Static W3C DOM HTML objects

The Moz1+ DOM (but unfortunately, not Internet Explorer 5 or later) adheres to the core JavaScript notion of prototype inheritance with respect to the object model. When a page loads into a Moz1+ browser, the browser creates HTML objects based on the prototypes of each object defined by the W3C DOM. For example, if you use The Evaluator Sr. (discussed in Chapter 4, "JavaScript Essentials") to see what kind of object the myP paragraph object is — enter document.getElementById("myP") in the top text box and click the Evaluate button — it reports that the object is based on the HTMLParagraphElement object of the DOM. Every instance of a p element object in the page inherits its default properties and methods from HTMLParagraphElement (which in turn inherits from HTMLElement, Element, and Node objects — all detailed in the JavaScript binding appendix of the W3C DOM specification).

You can use scripting to add properties to the prototypes of some of these static objects. To do so, you must use new features added to Moz1+ that are a part of JavaScript 1.5 and ECMAScript 5. Two new methods, __defineGetter__() and __defineSetter__(), enable you to assign functions to a custom property of an object.

Note

These methods are Mozilla specific. To prevent their collision with standardized implementations of these features in ECMAScript, the underscore characters on either side of the method name are pairs of underscore characters. ■

The functions execute whenever the property is read (the function assigned via the __defineGetter__() method) or modified (the function assigned through the __defineSetter__() method). The common way to define these functions is in the form of an anonymous function (see Chapter 23). The formats for the two statements that assign these behaviors to an object prototype are as follows:

```
object.prototype.__defineGetter__("propName",
      function([param1[,...[,paramN]]]) {
  // statements
  return returnValue;
})
object.prototype.__defineSetter__("propName",
      function([param1[,...[,paramN]]]) {
  // statements
  return returnValue;
})
```

The example in Listing 25-3 demonstrates how to add a read-only property to every HTML element object in the current document. The property, called childNodeDetail, returns an object. The object has two properties: one for the number of element child nodes and one for the number of text child nodes. Note that the this keyword in the function definition is a reference to the object for which the property is calculated. And because the function runs each time a script statement reads

the property, any scripted changes to the content after the page loads are reflected in the returned property value.

LISTING 25-3

Adding a Read-Only Prototype Property to All HTML Element Objects

```
<script type="text/javascript">
if (HTMLElement)
{
   HTMLElement.prototype.__defineGetter__("childNodeDetail", function()
   {
      var result = {elementNodes:0, textNodes:0 }
      for (var i = 0; i < this.childNodes.length; i++)
      {
         switch (this.childNodes[i].nodeType)
         {
            case 1:
               result.elementNodes++;
               break;
            case 3:
               result.textNodes++;
               break;
         }
      }
      return result;
   })
}
</script>
```

To access the property, use it like any other property of the object. For example:

```
var BodyNodeDetail = document.body.childNodeDetail;
```

The returned value in this example is an object, so you use regular JavaScript syntax to access one of the property values:

```
var BodyElemNodesCount = document.body.childNodeDetail.elementNodes;
```

Bidirectional event model

Despite the seemingly conflicting event models of NN4 (trickle down) and IE4 (bubble up), the W3C DOM event model (defined in Level 2) manages to employ both event propagation models. This gives the scripter the choice of where along an event's propagation path the event gets processed. To prevent conflicts with existing event model terminology, the W3C model invents many new terms for properties and new methods for events. Some coding probably requires W3C DOM–specific handling in a page aimed at multiple object models.

The W3C event model also introduces a new concept called the event listener. An *event listener* is essentially a mechanism that instructs an object to respond to a particular kind of event — very much

like the way the event handler attributes of HTML tags respond to events. But the DOM recommendation points out that it prefers a more script-oriented way of assigning event listeners: the addEventListener() method available for every node in the document hierarchy. Through this method, you advise the browser whether to force an event to bubble up the hierarchy (the default behavior that is also in effect if you use the HTML attribute type of event handler) or to be captured at a higher level.

Functions invoked by the event listener receive a single parameter consisting of the event object whose properties contain contextual details about the event (details such as the position of a mouse click, character code of a keyboard key, or a reference to the target object). For example, if a form includes a button whose job is to invoke a calculation function, the W3C DOM prefers the following way of assigning the event handler:

```
document.getElementById("calcButton").addEventListener("click",
doCalc, false);
```

The addEventListener() method takes three parameters. The first parameter is a string of the event to listen for; the second is a reference to the function to be invoked when that event fires; and the third parameter is a Boolean value. When you set this Boolean value to true, it turns on event capture whenever this event is directed to this target. The function then takes its cue from the event object passed as the parameter:

```
function doCalc(evt)
{
   // get shortcut reference to input button's form
   var form = evt.target.form;
   var results = 0;
   // other statements to do the calculation //
   form.result.value = results;
}
```

To modify an event listener, you use the removeEventListener() method to get rid of the old listener, and then employ addEventListener() with different parameters to assign the new one.

Preventing an event from performing its default action is also a different procedure in the W3C event model than in Internet Explorer. In Internet Explorer 4 (as well as Navigator 3 and 4), you can cancel the default action by allowing the event handler to evaluate to return false. Although this still works in Internet Explorer 5 and later, Microsoft includes another property of the window.event object, called returnValue. Setting that property to false anywhere in the function invoked by the event handler also kills the event before it does its normal job. But the W3C event model uses a method of the event object, preventDefault(), to keep the event from its normal task. You can invoke this method anywhere in the function that executes when the event fires.

Detailed information about an event is contained in an event object that must be passed to an event handler function where details may be read. If you assign event handlers via the W3C DOM addEventListener() method or an event handler property, the event object is passed automatically as the sole parameter to the event handler function. Include a parameter variable to catch the incoming parameter:

```
function swap(evt)
{
   // statements here to work with W3C DOM event object
}
```

But if you assign events through a tag attribute, you must explicitly pass the event object in the call to the function:

```
<a href="http://www.example.com" onmouseover="swap(event)">
```

Unfortunately, as of Internet Explorer 8 for Windows and Internet Explorer 5 for Macintosh, the W3C DOM event model has yet to be supported by Microsoft. You can, however, make the Internet Explorer and W3C event models work together if you assign event handlers by way of object properties or tag attributes, and throw in a little object detection as described later in this chapter, and in more detail in Chapter 32.

Scripting Trends

Although browser scripting had humble beginnings as a way to put some intelligence into form controls, the capabilities of the JavaScript language and DOM have inspired many a web developer to create what are essentially applications. Popular implementations of web-based email systems use extensive scripting and background communication with the server to keep pages updated quickly without the need to fetch and re-render the complete page each time you delete a message from the inbox list. It's not uncommon for large projects to involve multiple scripters (along with specialists in CSS, server programming, artists, and writers). Wrangling all the code can be a chore.

Separating content from scripting

Those who use CSS to style their sites have learned that separating style definitions from the HTML markup makes a huge improvement in productivity when it comes time to change colors or font specifications throughout a site. Instead of having to modify hundreds of `<font>` tag specifications scattered around the site, all it takes is a simple tweak of a single CSS rule in one `.css` file for that change to be implemented immediately across the board.

The notion of using HTML purely for a page's structure has also impacted scripting. It is rare these days for a professional scripter to put an event handler attribute inside an HTML tag. That would be considered too much mixing of content with behavior. In other words, the HTML markup should be able to stand on its own so that those with nonscriptable browsers (including those with vision or motor disabilities who use specialized browsers) can still get the fundamental information provided by the page. Any scripting that impacts the display or behavior of the page is added to the page after the HTML markup has loaded and rendered. Even assigning events to elements is done by script after the page load.

Script code is more commonly linked into a page from an external `.js` file. This isn't part of the separation of content and scripts trend, but a practice that offers many benefits, such as the same code being instantly usable on multiple pages. Additionally, when projects involve many code chefs, scripters can work on their code while writers work on the HTML and designers work on their external CSS code.

Note

You will see lots of examples in this book that use event handler attributes inside tags, and scripts embedded within the page. This approach is used primarily for simplicity in demonstrating a language or DOM feature. ■

Using the W3C DOM where possible

Basic support for W3C DOM element referencing and content manipulation has been implemented in mainstream browsers for so long that you can be assured that composing scripts for that model will work for the bulk of your visitors. That's not to say you can assume that every visitor is equipped that way, but the hassles that scripters used to endure to support conflicting object models are behind us for the most part. The days of writing extensive branching code for IE and Netscape are not-so-fond memories.

You still want to use object detection techniques to guard against the occasional old browser that stops by. That's where the technique of assigning event handlers by scripts can save a lot of headaches.

Except for some initializations that might occur while the page loads, most script execution in a web page occurs at the instigation of an event: A user clicks a button, types something in a text box, chooses from a `select` element, and so on. You can prevent older browsers from tripping up on W3C DOM syntax by doing your fundamental object detection right in the event assignment code, as in the following simplified example:

```
function setUpEvents()
{
   if (document.getElementById)
   {
      // statements to bind events to elements
   }
}
window.onload = setUpEvents;
```

Now, browsers that don't have even minimum support for the W3C DOM won't generate script errors when users click or type, because those events won't be assigned for those browsers. Then scripts that survive your object detection query can also modify the page, as you saw in Listing 4-2 in Chapter 4.

Handling events

You will still find some places where the W3C DOM isn't enough. This is particularly true in processing events, where Internet Explorer (at least through version 8) does not support the W3C DOM way of getting details about an event to the event handler function. The W3C DOM automatically passes the `event` object as a parameter to a handler function. In the Internet Explorer model, the `event` object is a property of the `window` object. Therefore, your functions have to equalize the differences where necessary. For example, to obtain a single variable representing the `event` object, regardless of browser, you can use a construction similar to the following:

```
function calculate(evt)
{
   evt = (evt) ? evt : window.event;
   // more statements to process event
}
```

Additional branching is necessary to inspect important details of the event. For example, the Internet Explorer event object property pointing to the object that received the event is called `srcElement`, whereas the W3C DOM version is called `target`. Again, a little bit of equalizing code in the event handler function can handle the disparity. When your script has a reference to the element receiving

the event, you can start using W3C DOM properties and methods of the element, because Internet Explorer supports those. You can find more details on event objects in Chapter 32.

Standards Compatibility Modes (DOCTYPE Switching)

Both Microsoft and Netscape/Mozilla discovered that they had, over time, implemented CSS features in ways that ultimately differed from the published standards that came later (usually after much wrangling among working-group members). To compensate for these differences and make a clean break to be compatible with the standards, the major browser makers decided to let the page author's choice of `<!DOCTYPE>` header element details determine whether the document was designed to follow the old way (sometimes called *quirks mode*) or the standards-compatible way. The tactic, known informally as *DOCTYPE switching,* is implemented in Internet Explorer 6 and later, Internet Explorer 5 for the Mac, all Mozilla-based browsers, and all WebKit-based browsers.

Although most of the differences between the two modes are small, there are some significant differences between the two modes in Internet Explorer 6 and later, particularly when styles or DHTML scripts rely on elements designed with borders, margins, and padding. Microsoft's original box model measured the dimensions of elements in a way that differed from the eventual CSS standard.

To place the affected browsers in CSS standards–compatible mode, you should include a `<!DOCTYPE>` element at the top of every document that specifies any of the following details (the first `<!DOCTYPE>` statement is for HTML5):

```
<!DOCTYPE html>

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN"
        "http://www.w3.org/TR/REC-html40/loose.dtd">

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Frameset//EN"
        "http://www.w3.org/TR/REC-html40/frameset.dtd">

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0//EN"
        "http://www.w3.org/TR/REC-html40/strict.dtd">

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
        "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Frameset//EN"
        "http://www.w3.org/TR/xhtml1/DTD/xhtml1-frameset.dtd">

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
        "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
        "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
```

Be aware, however, that older versions of Internet Explorer for Windows, such as Internet Explorer 5 or Internet Explorer 5.5, are ignorant of the standards-compatible mode and will use the old Microsoft quirks mode regardless of the `<!DOCTYPE>` setting. But using a standards-compliant mode DOCTYPE is more likely to force your content and style sheets to render similarly across the latest browsers.

Where to Go from Here

These past two chapters provided an overview of the core language and object model issues that anyone designing pages that use JavaScript must confront. The goal here is to stimulate your own thinking about how to embrace or discard levels of compatibility with your pages as you balance your desire to generate cool pages and serve your audience. From here on, the difficult choices are up to you.

To help you choose the objects, properties, methods, and event handlers that best suit your requirements, all the chapters in Part III and the rest of Part IV provide in-depth references to the DOM and core JavaScript language features. Observe the compatibility ratings for each language term very carefully to help you determine which features best suit your audience's browsers. Most example listings are complete HTML pages that you can load in various browsers to see how they work. Many others invite you to explore how things work through The Evaluator Sr. (see Chapter 4). Play around with the files, making modifications to build your own applications or expanding your working knowledge of JavaScript in the browser environment.

The language and object models have grown in the handful of years they have been in existence. The amount of language vocabulary has increased astronomically. It takes time to drink it all in and feel comfortable that you are aware of the powers available to you. Don't worry about memorizing the vocabulary. It's more important to acquaint yourself with the features and come back later when you need the implementation details.

Be patient. Be persistent. The reward will come.

CHAPTER

26

Generic HTML Element Objects

IN THIS CHAPTER

- **Working with HTML element objects**
- **Common properties and methods**
- **Event handlers of all element objects**

The object model specifications implemented in Internet Explorer 4+ and W3C (Mozilla-based and WebKit-based) browsers feature a large set of scriptable objects that represent what we often call *generic* HTML elements. Generic elements can be divided into two groups. One group, such as the `band strike` elements, defines font styles to be applied to enclosed sequences of text. The need for these elements (and the objects that represent them) is all but gone due to more page designers using style sheets. The second group of elements assigns context to content within their start and end tags. Examples of contextual elements include `h1`, `blockquote`, and the ubiquitous `p` element. Although browsers sometimes have consistent visual ways of rendering contextual elements by default (for example, the large bold font of an `<h1>` tag), the specific rendering is not the intended purpose of the tags. No formal standard dictates that text within an `em` element must be italicized: The style simply has become the custom since the very early days of browsers.

All of these generic elements share a large number of scriptable properties, methods, and event handlers. The sharing extends not only among generic elements, but also among virtually every renderable element — even if it has additional, element-specific properties, methods, and/or event handlers that we cover in depth in other chapters of this reference. Rather than repeat the details of these shared properties, methods, and event handlers for each object throughout this reference, we describe them in detail only in this chapter (unless there is a special behavior, bug, or trick associated with the item in some object described elsewhere). In succeeding reference chapters, each object description includes a list of the object's properties, methods, and event handlers, but we do not list shared items over and over (making it hard to find items that are unique to a particular element). Instead, you see a pointer back to this chapter for the items in common with generic HTML element objects.

Generic Objects

Table 26-1 lists all of the objects that we treat in this reference as *generic* objects. All of these objects share the properties, methods, and event handlers described in succeeding sections and have no special items that require additional coverage elsewhere in this book.

TABLE 26-1

Generic HTML Element Objects

Formatting Objects	Contextual Objects
b	acronym
big	address
center	cite
i	code
nobr	dfn
rt	del
ruby	div
s	em
small	ins
strike	kbd
sub	listing
sup	p
tt	plaintext
u	pre
wbr	samp
	span
	strong
	var
	xmp

Properties	Methods	Event Handlers
accessKey	addBehavior()	onactivate
all[]	addEventListener()	onafterupdate
attributes[]	appendChild()	onbeforecopy
baseURI	applyElement()	onbeforecut
behaviorUrns[]	attachEvent()	onbeforedeactivate
canHaveChildren	blur()	onbeforeeditfocus
canHaveHTML	clearAttributes()	onbeforepaste

Properties	Methods	Event Handlers
childNodes[]	click()	onbeforeupdate
children	cloneNode()	onblur
cite	compareDocumentPosition()	oncellchange
className	componentFromPoint()	onclick
clientHeight	contains()	oncontextmenu
clientLeft	createControlRange()	oncontrolselect
clientTop	detachEvent()	oncopy
clientWidth	dispatchEvent()	oncut
contentEditable	doScroll()	ondataavailable
currentStyle	dragDrop()	ondatasetchanged
dateTime	fireEvent()	ondatasetcomplete
dataFld	focus()	ondblclick
dataFormatAs	getAdjacentText()	ondeactivate
dataSrc	getAttribute()	ondrag
dir	getAttributeNode()	ondragend
disabled	getAttributeNodeNS()	ondragenter
document	getAttributeNS()	ondragleave
filters[]	getBoundingClientRect()	ondragover
firstChild	getClientRects()	ondragstart
height	getElementsByTagName()	ondrop
hideFocus	getElementsByTagNameNS()	onerrorupdate
id	getExpression()	onfilterchange
innerHTML	getFeature()	onfocus
innerText	getUserData()	onfocusin
isContentEditable	hasAttribute()	onfocusout
isDisabled	hasAttributeNS()	onhelp
isMultiLine	hasAttributes()	onkeydown
isTextEdit	hasChildNodes()	onkeypress

continued

TABLE 26-1 *(continued)*

Properties	Methods	Event Handlers
lang	insertAdjacentElement()	onkeyup
language	insertAdjacentHTML()	onlayoutcomplete
lastChild	insertAdjacentText()	onlosecapture
length	insertBefore()	onmousedown
localName	isDefaultNamespace()	onmouseenter
namespaceURI	isEqualNode()	onmouseleave
nextSibling	isSameNode()	onmousemove
nodeName	isSupported()	onmouseout
nodeType	item()	onmouseover
nodeValue	lookupNamespaceURI()	onmouseup
offsetHeight	lookupPrefix()	onmousewheel
offsetLeft	mergeAttributes()	onmove
offsetParent	normalize()	onmoveend
offsetTop	releaseCapture()	onmovestart
offsetWidth	removeAttribute()	onpaste
outerHTML	removeAttributeNode()	onpropertychange
outerText	removeAttributeNS()	onreadystatechange
ownerDocument	removeBehavior()	onresize
parentElement	removeChild()	onresizeend
parentNode	removeEventListener()	onresizestart
parentTextEdit	removeExpression()	onrowenter
prefix	removeNode()	onrowexit
previousSibling	replaceAdjacentText()	onrowsdelete
readyState	replaceChild()	onrowsinserted
recordNumber	replaceNode()	onscroll
runtimeStyle	scrollIntoView()	onselectstart
scopeName	setActive()	
scrollHeight	setAttribute()	
scrollLeft	setAttributeNode()	

Properties	Methods	Event Handlers
scrollTop	setAttributeNodeNS()	
scrollWidth	setAttributeNS()	
sourceIndex	setCapture()	
style	setExpression()	
tabIndex	setUserData()	
tagName	swapNode()	
tagUrn	tags()	
textContent	toString()	
title	urns()	
uniqueID		
unselectable		
width		

Syntax

To access element properties or methods, use this:

```
(IE4+)     [document.all.]objectID.property | method([parameters])
(IE5+/W3C) document.getElementById(objectID).property | method([parameters])
```

Note

It's important to note that unless you have the specific need of supporting IE4, which is highly unlikely at this point in time, you should rely solely on the latter approach of referencing element properties and methods via the `getElementById()` **method.** ■

About these objects

All objects listed in Table 26-1 are document object model (DOM) representations of HTML elements that influence either the font style or the context of some HTML content. The large set of properties, methods, and event handlers associated with these objects also applies to virtually every other DOM object that represents an HTML element. Discussions about object details in this chapter apply to dozens of other objects described in succeeding chapters of this reference section.

Properties

accessKey

Value: One-character string Read/Write

Compatibility: WinIE4+, MacIE4+, NN7+, Moz-, Safari+, Opera+, Chrome+

***elementObject*.accessKey**

For many elements, you can specify a keyboard character (letter, numeral, or punctuation symbol) that — when typed as an Alt+key combination (on the Win32 OS platform), a Ctrl+key combination (on the MacOS), or a Shift+Esc+key combination (on Opera) — brings focus to that element. An element that has focus is the one that is set to respond to keyboard activity. If the newly focused element is out of view in the document's current scroll position, the document is scrolled to bring that focused element into view (also see the `scrollIntoView()` method). The character you specify can be an uppercase or lowercase value, but these values are not case sensitive. If you assign the same letter to more than one element, the user can cycle through all elements associated with that `accessKey` value.

Internet Explorer gives some added powers to the `accessKey` property in some cases. For example, if you assign an `accessKey` value to a `label` element object, the focus is handed to the form element associated with that label. Also, when elements such as buttons have focus, pressing the spacebar acts the same as clicking the element with a mouse.

Exercise some judgment in selecting characters for `accessKey` values. If you assign a letter that is normally used to access one of the browser's built-in menus (for example, Alt+F for the File menu in IE), that `accessKey` setting overrides the browser's normal behavior. To users who rely on keyboard access to menus, your control over that key combination can be disconcerting. In other browsers your `accessKey` assignment will not override the browser's normal behavior.

Example

Listing 26-1 shows an example of how to use the `accessKey` property to manipulate the keyboard interface for navigating a web page. When you load the script in Listing 26-1, adjust the height of the browser window so that you can see nothing below the second dividing rule. Enter any character in the Settings portion of the page and press Enter. (The Enter key may cause your computer to beep.) Pressing the Enter key does not do anything visible in your browser; it just assigns the character you entered into the textbox to the appropriate element. Then hold down the Alt (Windows) or Ctrl (Mac) key (or Shift+Esc keys in Opera) while pressing the same keyboard key. The element from below the second divider should come into view.

Note

The property assignment event handling technique employed throughout the code in this chapter and much of the book is a deliberate simplification to make the code more readable. It is generally better to use the more modern approach of binding events using the `addEventListener()` (NN6+/Moz/W3C) or `attachEvent()` (IE5+) method. A modern cross-browser event handling technique is explained in detail in Chapter 32. ■

LISTING 26-1

Controlling the accessKey Property

HTML: jsb26-01.html

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>accessKey Property</title>
```

```
      <script type="text/javascript" src="../jsb-global.js"></script>
      <script type="text/javascript" src="jsb26-01.js"></script>
   </head>
   <body>
      <h1>accessKey Property Lab</h1>
      <hr />
      Settings:<br />
      <form name="input">
        Assign an accessKey value to the Button below and press Return:
        <input type="text" size="2" maxlength="1"
               onkeypress="assignKey('button', this)" />
        <br />
        Assign an accessKey value to the Text Box below and press Return:
        <input type="text" size="2" maxlength="1"
               onkeypress="return assignKey('text', this)" />
        <br />
        Assign an accessKey value to the Table below (IE5.5+ only)
        and press Return:
        <input type="text" size="2" maxlength="1"
               onkeypress="return assignKey('table', this)" />
      </form>
      <br />
      Then press Alt (Windows) or Control (Mac) or ShiftESC (Opera) + the key
      <br />
      <em>Size the browser window to view nothing lower than this line.</em>
      <hr />
      <form name="output" onsubmit="return false">
         <input type="button" id="access1" value="Standard Button" />
         <input type="text" id="access2" />
      </form>
      <table id="myTable" cellpadding="10" border="2">
         <tr>
            <th>Quantity</th>
            <th>Description</th>
            <th>Price</th>
         </tr>
         <tbody bgcolor="red">
            <tr>
               <td width="100">4</td>
               <td>Primary Widget</td>
               <td>$14.96</td>
            </tr>
            <tr>
               <td>10</td>
               <td>Secondary Widget</td>
               <td>$114.96</td>
            </tr>
         </tbody>
      </table>
   </body>
</html>
```

continued

LISTING 26-1 *(continued)*

JavaScript: jsb26-01.js

```
function assignKey(type, elem)
{
   if (window.event)       // IE, Opera, Safari, Chrome
   {
      if (window.event.keyCode == 13)
      {
         switch (type)
         {
            case "button":
               document.getElementById("access1").accessKey = elem.value;
               break;
            case "text":
               document.getElementById("access2").accessKey = elem.value;
               break;
            case "table":
               document.getElementById("myTable").accessKey = elem.value;
         }
         return false;
      }
   }
}
```

Related Item: `scrollIntoView()` method

all[]

Value: Array of nested element objects Read-Only

Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari+ (partial), Opera+, Chrome+ (partial)

Originally exclusive to Internet Explorer, the `all` property is a collection (array) of every HTML element and (in IE5+) XML tag within the scope of the current object. Opera now supports it fully, while WebKit-based browsers (such as Safari and Chome) support it only as a property of the document object. Even so, you should strongly consider using the `document.getElementById()` method (as described in Chapter 29), which is the official W3C and cross-browser approach for referencing elements. The `document.getElementById()` method is supported in IE5+.

Items in this array appear in source-code order, and the array is oblivious to element containment among the items. For HTML element containers, the source-code order is dependent on the position of the start tag for the element; end tags are not counted. But for XML tags, end tags appear as separate entries in the array.

Every `document.all` collection contains objects for the `html`, `head`, `title`, and `body` element objects even if the actual HTML source code omits the tags. The object model creates these objects for every document that is loaded into a window or frame. Although the `document.all` reference may be the most common usage, the `all` property is available for any container element. For example, `document.forms[0].all` exposes all elements defined within the first form of a page.

You can access any element that has an identifier assigned to its `id` attribute by that identifier in string form (as well as by index integer). Rather than use the performance-costly `eval()` function to convert a string to an object reference, use the string value of the name as an array index value:

```
var paragraph = document.all["myP"];
```

Internet Explorer enables you to use either square brackets or parentheses for single collection index values. Thus, the following two examples evaluate identically:

```
var paragraph = document.all["myP"];
var paragraph = document.all("myP");
```

In the rare case that two or more elements within the `all` collection have the same ID, the syntax for the string index value returns a collection of just those identically named elements. But you can use a second argument (in parentheses) to signify the integer of the initial collection and thus single out a specific instance of that named element:

```
var secondRadio = document.all("group0",1);
```

As a more readable alternative, you can use the `item()` method (described later in this chapter) to access the same kinds of items within a collection:

```
var secondRadio = document.all.item("group0",1);
```

Also see the `tags()` method (later in this chapter) as a way to extract a set of elements from an `all` collection that matches a specific tag name.

Example

Use The Evaluator (see Chapter 4) to experiment with the `all` collection. Enter the following statements one at a time in the lower text box, and review the results in the text area for each:

```
document.all
myTable.all
myP.all
```

If you encounter a numbered element within a collection, you can explore that element to see which tag is associated with it. For example, if one of the results for the `document.all` collection says `document.all.8=[object]`, enter the following statement in the topmost text box:

```
document.all[8].tagName
```

Related Items: `item()`, `tags()`, `document.getElementById()` methods

attributes[]

Value: Array of attribute object references Read-Only

Compatibility: WinIE5+, MacIE5+, NN6+, Moz+, Safari+, Opera+, Chrome+

The `attributes` property consists of an array of attributes specified for an element. In IE5+, the `attributes` array contains an entry for every possible property that the browser has defined for its elements — even if the attribute is not set explicitly in the HTML tag. Also, any attributes that

you add later via script facilities such as the `setAttribute()` method are not reflected in the `attributes` array. In other words, the IE5+ `attributes` array is fixed, using default values for all properties except those that you explicitly set as attributes in the HTML tag.

Mozilla browsers' `attributes` property returns an array that is a named node map (in W3C DOM terminology) — an object that has its own properties and methods to read and write attribute values. For example, you can use the `getNamedItem(attrName)` and `item(index)` methods on the array returned from the `attributes` property to access individual attribute objects via W3C DOM syntax.

IE5 and Mozilla had different ideas about what an attribute object should be. Mozilla is dependent on the W3C DOM node inheritance model, discussed in Chapter 25. In the past, the larger set of properties in the W3C DOM node were solely supported by Mozilla. Now, as you can see, IE supports more of the W3C DOM. Table 26-2 shows the variety of properties of an attribute object as defined by the two object models. The WebKit-based browsers offer the same property support as Mozilla, while Opera lies in between.

TABLE 26-2

Attribute Object Properties

Property	IE5	Moz	Description
`attributes`	Yes	Yes	Array of nested attribute objects (null)
`childNodes`	No	Yes	Child node array
`firstChild`	No	Yes	First child node
`lastChild`	No	Yes	Last child node
`localName`	No	Yes	Name within current namespace
`name`	Yes	Yes	Attribute name
`nameSpaceURI`	Yes	Yes	XML namespace URI
`nextSibling`	Yes	Yes	Next sibling node
`nodeName`	Yes	Yes	Attribute name
`nodeType`	Yes	Yes	Node type (2)
`nodeValue`	Yes	Yes	Value assigned to attribute
`ownerDocument`	Yes	Yes	Document object reference
`ownerElement`	Yes	Yes	Element node reference
`parentNode`	Yes	Yes	Parent node reference
`prefix`	Yes	Yes	XML namespace prefix
`previousSibling`	Yes	Yes	Previous sibling node
`specified`	Yes	Yes	Whether attribute is explicitly specified (Boolean)
`value`	Yes	Yes	Value assigned to attribute

The most helpful property of an attribute object is the Boolean `specified` property. In IE, this lets you know whether the attribute is explicitly specified in the element's tag. Because Mozilla returns only explicitly specified attributes in the `attributes` array, the value in Mozilla is always `true`. Most of the time, however, you'll probably use an element object's `getAttribute()` and `setAttribute()` methods to read and write attribute values.

Example

Use The Evaluator (see Chapter 4) to examine the values of the `attributes` array for some of the elements in that document. Enter each of the following expressions in the bottom text box, and see the array contents in the Results text area for each:

```
document.body.attributes
document.getElementById("myP").attributes
document.getElementById("myTable").attributes
```

If you have both IE5+ and a W3C DOM–compatible browser, compare the results you get for each of these expressions. To view the value of a single attribute in WinIE5+ by accessing the `attributes` array, enter the following statement in the top text box:

```
document.getElementById("myForm2").attributes["name"].value
```

For W3C browsers, IE6+, and MacIE5, use the W3C DOM syntax:

```
document.getElementById("myForm2").attributes.getNamedItem("name").value
```

Related Items: `getAttribute()`, `mergeAttributes()`, `removeAttribute()`, `setAttribute()` methods

baseURI

Value: Full URI string — Read-Only

Compatibility: WinIE-, MacIE-, NN6+, Moz+, Safari+, Opera+, Chrome+

This property reveals the full path to the source from which the element was served. The property is handy in applications that import XML data, in which case the source of an XML element is likely different from the HTML page in which it is being processed.

behaviorUrns[]

Value: Array of behavior URN strings — Read-Only

Compatibility: WinIE5+, MacIE-, NN-, Moz-, Safari-, Opera-, Chrome-

The Internet Explorer `behaviorUrns` property is designed to provide a list of addresses, in the form of *URNs (Uniform Resource Names)*, of all behaviors assigned to the current object. If there are no behaviors, the array has a length of zero. In practice, however, IE5+ always returns an array of empty strings. Perhaps the potential exposure of URNs by script was deemed to be a privacy risk.

Related Item: `urns()` method

canHaveChildren

Value: Boolean — Read-Only

Compatibility: WinIE5+, MacIE-, NN-, Moz-, Safari-, Opera-, Chrome-

***elementObject*.canHaveChildren**

Useful in some dynamic content situations, the `canHaveChildren` property reveals whether a particular element is capable of containing a child (nested) element. Most elements that have start and end tags (particularly the generic elements covered in this chapter) can contain nested elements. A nested element is referred to as a *child* of its parent container.

Example

Listing 26-2 shows an example of how to use the `canHaveChildren` property to visually identify elements on a page that can have nested elements. This example uses color to demonstrate the difference between an element that can have children and one that cannot. The first button sets the `color` style property of every visible element on the page to red. Thus, elements (including the normally non-childbearing ones such as `hr` and `input`) are affected by the color change. But if you reset the page and click the largest button, only those elements that can contain nested elements receive the color change.

LISTING 26-2

Reading the canHaveChildren Property

HTML: jsb26-02.html

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>canHaveChildren Property</title>
      <script type="text/javascript" src="../jsb-global.js"></script>
      <script type="text/javascript" src="jsb26-02.js"></script>
   </head>
   <body>
      <h1>canHaveChildren Property Lab</h1>
      <hr />
      <form name="input">
         <input type="button" value="Color All Elements" onclick="colorAll()" />
         <br />
         <input type="button" value="Reset" onclick="history.go(0)" />
         <br />
         <input type="button" value="Color Only Elements That Can Have Children"
                onclick="colorChildBearing()" />
      </form>
      <br />
      <hr />
      <form name="output">
         <input type="checkbox" checked="checked" />Your basic checkbox
         <input type="text" name="access2" value="Some textbox text." />
      </form>
      <table id="myTable" cellpadding="10" border="2">
         <tr>
            <th>Quantity</th>
            <th>Description</th>
            <th>Price</th>
         </tr>
```

```
      <tbody>
         <tr>
            <td width="100">4</td>
            <td>Primary Widget</td>
            <td>$14.96</td>
         </tr>
         <tr>
            <td>10</td>
            <td>Secondary Widget</td>
            <td>$114.96</td>
         </tr>
      </tbody>
   </table>
  </body>
</html>
```

JavaScript: jsb26-02.js

```
function colorAll()
{
   var elems = document.getElementsByTagName("*");
   for (var i = 0; i < elems.length; i++)
   {
      elems[i].style.color = "red";
   }
}

function colorChildBearing()
{
   var elems = document.getElementsByTagName("*");
   for (var i = 0; i < elems.length; i++)
   {
      if (elems[i].canHaveChildren)
      {
         elems[i].style.color = "red";
      }
   }
}
```

Related Items: `childNodes`, `firstChild`, `lastChild`, `parentElement`, `parentNode` properties; `appendChild()`, `hasChildNodes()`, `removeChild()` methods

canHaveHTML

Value: Boolean Read-Only and Read/Write

Compatibility: WinIE5+, MacIE-, NN-, Moz-, Safari-, Opera-, Chrome-

Not all HTML elements are containers of HTML content. The `canHaveHTML` property lets scripts find out whether a particular object can accept HTML content, such as for insertion or replacement by object methods. The value for a `p` element, for example, is `true`. The value for a `br` element is `false`. The property is read-only for all elements except HTML Components, in which case it is read/write.

Example

Use The Evaluator (see Chapter 4) in WinIE5+ to experiment with the `canHaveHTML` property. Enter the following statements in the top text box, and observe the results:

```
document.getElementById("input").canHaveHTML
document.getElementById("myP").canHaveHTML
```

The first statement returns `false` because an `input` element (the top text box, in this case) cannot have nested HTML. But the `myP` element is a `p` element that gladly accepts HTML content.

Related Items: `appendChild()`, `insertAdjacentHTML()`, `insertBefore()` methods

childNodes[]

Value: Array of node objects Read-Only

Compatibility: WinIE5+, MacIE5+, NN6+, Moz+, Safari+, Opera+, Chrome+

The `childNodes` property consists of an array of node objects contained by the current object. Note that child nodes consist of both element objects and text nodes. Therefore, depending on the content of the current object, the number of `childNodes` and children collections may be different.

Caution

If you use the `childNodes` array in a `for` loop that iterates through a sequence of HTML (or XML) elements, watch out for the possibility that the browser treats source-code whitespace (blank lines between elements and even simple carriage returns between elements) as text nodes. This potential problem affects MacIE5 and W3C-based browsers. If present, these extra text nodes occur primarily surrounding block elements. ■

Note

Most looping activity through the `childNodes` array aims to examine, count, or modify element nodes within the collection. If that is your script's goal, test each node returned by the `childNodes` array, and verify that the `nodeType` property is `1` (element) before processing that node; otherwise, skip the node. The skeletal structure of such a loop follows:

```
for (var i = 0; i < myElem.childNodes.length; i++)
{
   if (myElem.childNodes[i].nodeType == 1)
   {
      statements to work on element node i
   }
}
```

The presence of these phantom text nodes also impacts the nodes referenced by the `firstChild` and `lastChild` properties, described later in this chapter. ■

Example

Listing 26-3 contains an example of how you might code a function that walks the child nodes of a given node. The `walkChildNodes()` function shown in the listing accumulates and returns a hierarchical list of child nodes from the point of view of the document's HTML element (the default) or any element whose ID you pass as a string parameter. This function is embedded in The Evaluator so

that you can inspect the child node hierarchy of that page or (when using `evaluator.js` for debugging as described in Chapter 48 on the CD-ROM) the node hierarchy within any page you have under construction. Try it out in The Evaluator by entering the following statements in the top text box:

```
walkChildNodes()
walkChildNodes(document.getElementById("myP"))
```

The results of this function show the nesting relationships among all child nodes within the scope of the initial object. It also shows the act of drilling down to further `childNodes` collections until all child nodes are exposed and catalogued. Text nodes are labeled accordingly. The first 15 characters of the actual text are placed in the results to help you identify the nodes when you compare the results against your HTML source code.

LISTING 26-3

Collecting Child Nodes

```
<!-- xmp tags added for display of HTML in a browser window -->
<xmp>
   function walkChildNodes(objRef, n)
   {
      var obj;
      if (objRef)
      {
         if (typeof objRef == "string")
         {
            obj = document.getElementById(objRef);
         }
         else
         {
            obj = objRef;
         }
      }
      else
      {
         obj = (document.body.parentElement) ?
            document.body.parentElement : document.body.parentNode;
      }
      var output = "";
      var indent = "";
      var i, group, txt;
      if (n)
      {
         for (i = 0; i < n; i++)
         {
            indent += "+---";
         }
      }
      else
```

continued

LISTING 26-3 *(continued)*

```
      {
         n = 0;
         output += "Child Nodes of <" + obj.tagName;
         output += ">\n====================\n";
      }
      group = obj.childNodes;
      for (i = 0; i < group.length; i++)
      {
         output += indent;
         switch (group[i].nodeType)
         {
            case 1:
               output += "<" + group[i].tagName;
               output += (group[i].id) ? " ID=" + group[i].id : "";
               output += (group[i].name) ? " NAME=" + group[i].name : "";
               output += ">\n";
               break;
            case 3:
               txt = group[i].nodeValue.substr(0,15);
               output += "[Text:\"" + txt.replace(/[\r\n]/g,"<cr>");
                  if (group[i].nodeValue.length > 15)
                  {
                     output += "...";
                  }
               output += "\"]\n";
               break;
            case 8:
               output += "[!COMMENT!]\n";
               break;
            default:
               output += "[Node Type = " + group[i].nodeType + "]\n";
         }
         if (group[i].childNodes.length > 0)
         {
           output += walkChildNodes(group[i], n+1);
         }
      }
      return output;
   }
</xmp>
```

Related Items: nodeName, nodeType, nodeValue, parentNode properties; cloneNode(), hasChildNodes(), removeNode(), replaceNode(), swapNode() methods

children

Value: Array of element objects Read-Only

Compatibility: WinIE4+, MacIE4+, NN-, Moz+, Safari+, Opera+, Chrome+

The `children` property consists of an array of element objects contained by the current object. Unlike the `childNodes` property, `children` does not take into account text nodes, but focuses strictly on the HTML (and XML) element containment hierarchy from the point of view of the current object. Children exposed to the current object are immediate children only. If you want to get all element objects nested within the current object (regardless of how deeply nested they are), use the `all` collection instead.

Example

Listing 26-4 shows how you can use the `children` property to walk the child nodes of a given node. This function accumulates and returns a hierarchical list of child elements from the point of view of the document's HTML element (the default) or any element whose ID you pass as a string parameter. This function is embedded in The Evaluator so that you can inspect the parent–child hierarchy of that page or (when using `evaluator.js` for debugging as described in Chapter 48 on the CD-ROM) the element hierarchy within any page you have under construction. Try it out in The Evaluator by entering the following statements in the top text box:

```
walkChildren("myTable")
```

Notice in this example that the `walkChildren()` function is called with the name of an element instead of a call to `document.getElementId()`. This reveals the flexibility of the `walkChildren()` function and how it can operate on either an object or the name of an object (element). The `walkChildNodes()` function in Listing 26-3 offers the same flexibility.

The results of the `walkChildren()` function show the nesting relationships among all parent and child elements within the scope of the initial object. It also shows the act of drilling down to further `children` collections until all child elements are exposed and cataloged. The element tags also display their `id` and/or `name` attribute values if any are assigned to the elements in the HTML source code.

LISTING 26-4

Collecting Child Elements

```
<!-- xmp tags added for display of HTML in a browser window -->
<xmp>
   function walkChildren(objRef, n)
   {
      var obj;
      if (objRef)
      {
         if (typeof objRef == "string")
         {
            obj = document.getElementById(objRef);
         }
         else
         {
            obj = objRef;
         }
```

continued

LISTING 26-4 *(continued)*

```
      }
      else
      {
         obj = document.body.parentElement;
      }
      var output = "";
      var indent = "";
      var i, group;
      if (n)
      {
         for (i = 0; i < n; i++)
         {
            indent += "+---";
         }
      }
      else
      {
         n = 0;
         output += "Children of <" + obj.tagName;
         output += ">\n====================\n";
      }
      group = obj.children;
      for (i = 0; i < group.length; i++)
      {
         output += indent + "<" + group[i].tagName;
         output += (group[i].id) ? " ID=" + group[i].id : "";
         output += (group[i].name) ? " NAME=" + group[i].name : "";
         output += ">\n";
         if (group[i].children.length > 0)
         {
            output += walkChildren(group[i], n+1);
         }
      }
      return output;
   }
</xmp>
```

Related Items: `canHaveChildren`, `firstChild`, `lastChild`, `parentElement` properties; `appendChild()`, `removeChild()`, `replaceChild()` methods

cite

Value: URL string   Read/Write

Compatibility: WinIE6+, MacIE-, NN6+, Moz+, Safari+, Opera+, Chrome+

The `cite` property contains a URL that serves as a reference to the source of an element, as in the author of a quote. The property is intended to apply to only the `blockquote`, `q`, `del`, and

ins element objects, but IE supports it in a wider range of text content objects. This may or may not be a mistake, so it's probably not a safe bet to use the property outside its intended elements.

className

Value: String Read/Write

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

A *class name* is an identifier that is assigned to the class attribute of an element. To associate a cascading style sheets (CSS) rule with several elements in a document, assign the same identifier to the class attributes of those elements, and use that identifier (preceded by a period) as the CSS rule's selector. An element's className property enables the application of different CSS rules to that element under script control. Listing 26-5 shows an example of such a script.

Example

The style of an element toggles between on and off in Listing 26-5 by virtue of setting the element's className property alternatively to an existing style sheet class selector name and an empty string. When you set the className to an empty string, the default behavior of the h1 element governs the display of the first header. A click of the button forces the style sheet rule to override the default behavior in the first h1 element.

LISTING 26-5

Working with the className Property

HTML: jsb26-05.html

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>className Property</title>
      <style type="text/css">
         .special
         {
            font-size:16pt; color:red;
         }
      </style>
      <script type="text/javascript" src="../jsb-global.js"></script>
      <script type="text/javascript" src="jsb26-05.js"></script>
   </head>
   <body>
      <h1>className Property Lab</h1>
      <hr />
      <form name="input">
         <input type="button" value="Toggle Class Name"
               onclick="toggleSpecialStyle('head1')" />
      </form>
      <br />
```

continued

LISTING 26-5 *(continued)*

```
      <h1 id="head1">ARTICLE I</h1>
      <p>Congress shall make no law respecting an establishment of religion, or
         prohibiting the free exercise thereof; or abridging the freedom of
         speech, or of the press; or the right of the people peaceably to
         assemble, and to petition the government for a redress of grievances.
      </p>
      <h1>ARTICLE II</h1>
      <p>A well regulated militia, being necessary to the security of a free
         state, the right of the people to keep and bear arms, shall not be
         infringed.
      </p>
   </body>
</html>
```

JavaScript: jsb26-05.js

```
function toggleSpecialStyle(elemID)
{
   var elem = (document.all) ? document.all(elemID) :
               document.getElementById(elemID);
      if (elem.className == "")
      {
         elem.className = "special";
      }
      else
      {
         elem.className = "";
      }
}
```

You can also create multiple versions of a style rule with different class selector identifiers and apply them at will to a given element.

Related Items: `rule`, `stylesheet` objects (Chapter 38); `id` property

`clientHeight`
`clientWidth`

Value: Integer   Read-Only

Compatibility: WinIE4+, MacIE4+, NN7, Moz1.0.1+, Safari+, Opera+, Chrome+

These two properties by and large reveal the pixel height and width of the content within an element whose style sheet rule includes height and width settings. In theory, these measures do not take into account any margins, borders, or padding that you add to an element by way of style sheets. In practice, however, different combinations of borders, margins, and padding influence these values in unexpected ways. One of the more reliable applications of the `clientHeight` property enables you to discover, for example, where the text of an overflowing element ends. To read the rendered

dimensions of an element, you are better served across browsers with the `offsetHeight` and `offsetWidth` properties.

For the `document.body` object, the `clientHeight` and `clientWidth` properties return the inside height and width of the window or frame (plus or minus a couple of pixels). These take the place of desirable, but nonexistent, window properties in IE.

Unlike earlier versions, Internet Explorer 5+ expanded the number of objects that employ these properties to include virtually all objects that represent HTML elements. Values for these properties in Mozilla-based browsers are zero except for `document.body`, which measures the browser's current content area.

Example

Listing 26-6 for IE includes an example of how to size content dynamically on a page based on the client-area width and height. This example calls upon the `clientHeight` and `clientWidth` properties of a `div` element that contains a paragraph element. Only the width of the `div` element is specified in its style sheet rule, which means that the paragraph's text wraps inside that width and extends as deeply as necessary to show the entire paragraph. The `clientHeight` property describes that depth. The `clientHeight` property then calculates where a logo image should be positioned immediately after `div`, regardless of the length of the text. As a bonus, the `clientWidth` property helps the script center the image horizontally with respect to the paragraph's text.

LISTING 26-6

Using clientHeight and clientWidth Properties

HTML: jsb26-06.html

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>clientHeight and clientWidth Properties</title>
      <script type="text/javascript" src="../jsb-global.js"></script>
      <script type="text/javascript" src="jsb26-06.js"></script>
   </head>
   <body>
      <button onclick="showLogo()">Position and Show Logo Art</button>
      <div id="myDIV" style="width:200px">
         <p>Lorem ipsum dolor sit amet, consectetaur adipisicing elit, sed do
            eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim
            adminim veniam, quis nostrud exercitation ullamco laboris nisi ut
            aliquip ex ea commodo consequat. Duis aute irure dolor in
            reprehenderit involuptate velit esse cillum dolore eu fugiat nulla
            pariatur. Excepteur sint occaecat cupidatat non proident.
         </p>
      </div>
      <div id="logo" style="position:absolute; width:120px; visibility:hidden">
         <img alt="image" src="logo.gif" />
      </div>
   </body>
</html>
```

continued

LISTING 26-6 *(continued)*

```
JavaScript: jsb26-01.js
function showLogo()
{
   var paragraphW = document.getElementById("myDIV").clientWidth;
   var paragraphH = document.getElementById("myDIV").clientHeight;
   // correct for Windows/Mac discrepancies
   var paragraphTop = (document.getElementById("myDIV").clientTop) ?
      document.getElementById("myDIV").clientTop :
      document.getElementById("myDIV").offsetTop;
   var logoW = document.getElementById("logo").style.pixelWidth;
   // center logo horizontally against paragraph
   document.getElementById("logo").style.pixelLeft = (paragraphW-logoW) / 2;
   // position image immediately below end of paragraph
   document.getElementById("logo").style.pixelTop = paragraphTop + paragraphH;
   document.getElementById("logo").style.visibility = "visible";
}
```

To assist in the vertical positioning of the logo, the `offsetTop` property of the `div` object provides the position of the start of the `div` with respect to its outer container (the `body`). Unfortunately, MacIE uses the `clientTop` property to obtain the desired dimension. That measure (assigned to the `paragraphTop` variable), plus the `clientHeight` of the `div`, provides the top coordinate of the image.

Related Items: `offsetHeight`, `offsetWidth` properties

clientLeft
clientTop

Value: Integer Read-Only

Compatibility: WinIE4+, MacIE4+, NN-, Moz+, Safari+, Opera+, Chrome+

The purpose and names of the `clientLeft` and `clientTop` properties are confusing at best. Unlike the `clientHeight` and `clientWidth` properties, which apply to the content of an element, the `clientLeft` and `clientTop` properties return essentially no more information than the thickness of a border around an element — provided that the element is positioned. If you do not specify a border or do not position the element, the values are zero (although the `document.body` object can show a couple of pixels in each direction without explicit settings). If you are trying to read the left and top coordinate positions of an element, the `offsetLeft` and `offsetTop` properties are more valuable in WinIE; as shown in Listing 26-6, however, the `clientTop` property returns a suitable value in MacIE. Virtually all elements have the `clientLeft` and `clientTop` properties in IE5+, whereas support in MacIE is less consistent.

Related Items: `offsetLeft`, `offsetTop` properties

contentEditable

Value: Boolean Read/Write

Compatibility: WinIE5.5+, MacIE-, NN-, Moz+, Safari+, Opera+, Chrome+

IE5.5 introduced the concept of editable HTML content on a page. Element tags can include a `contentEditable` attribute whose value is echoed via the `contentEditable` property of the element. The default value for this property is `inherit`, which means that the property inherits whatever setting this property has in the hierarchy of HTML containers outward to the body. If you set the `contentEditable` property to `true`, that element and all nested elements set to inherit the value become editable; conversely, a setting of `false` turns off the option to edit the content. Some browsers automatically provide a visual cue for editable elements by giving an editable element a glowing blue border.

Example

Listing 26-7 demonstrates how to use the `contentEditable` property to create a very simple poetry editor. When you click the button of a freshly loaded page, the `toggleEdit()` function captures the opposite of the current editable state via the `contentEditable` property of the `div` that is subject to edit. You switch on editing for that element in the next statement by assigning the new value to the `contentEditable` property of the `div`. For added impact, turn the text of the `div` to red to provide additional user feedback about what is editable on the page. The button label is also switched to one that indicates the action invoked by the next click of that button.

LISTING 26-7

Using the contentEditable Property

HTML: jsb26-07.html

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Editable Content</title>
      <style type="text/css">
         .normal
         {
            color: black;
         }
         .editing
         {
            color: red;
         }
      </style>
      <script type="text/javascript" src="../jsb-global.js"></script>
      <script type="text/javascript" src="jsb26-07.js"></script>
   </head>
   <body>
      <h1>Poetry Editor</h1>
      <hr />
      <p>Turn on editing to modify the following text:</p>
      <div id="noneditableText">
         Roses are red,
         <br />
         Violets are blue.
      </div>
```

continued

LISTING 26-7 *(continued)*

```
      <div id="editableText">
         Line 3,
         <br />
         Line 4.
      </div>
      <p>
         <button id="editBtn" onclick="toggleEdit()"
                 onfocus="this.blur()">Enable Editing</button>
      </p>
   </body>
</html>
```

JavaScript: jsb26-07.js

```
function toggleEdit()
{
   var newState = document.getElementById("editableText").contentEditable;
   // While contentEditable is boolean, different browsers behave differently
      if (newState == false || newState == "false" || newState == "inherit")
      {
         newState = true;
      }
      else
      {
         newState = false;
      }
   document.getElementById("editableText").contentEditable = newState;
   document.getElementById("editableText").className = (newState) ?
        "editing" : "normal";
   document.getElementById("editBtn").innerHTML = (newState) ?
        "Disable Editing" : "Enable Editing";
}
```

Related Item: `isContentEditable` property

currentStyle

Value: `style` object Read-Only

Compatibility: WinIE5+, MacIE5+, NN-, Moz-, Safari-, Opera-, Chrome-

Every element has style attributes applied to it, even if those attributes are the browser's default settings. Because an element's `style` object reflects only those properties whose corresponding attributes are explicitly set via CSS statements, you cannot use the `style` property of an `element` object to view default style settings applied to an element. That's where the `currentStyle` property comes in.

This property returns a read-only `style` object that contains values for every possible `style` property applicable to the element. If a `style` property is explicitly set via CSS statement or script adjustment, the current reading for that property is also available here. Thus, a script can inquire about any property to determine whether it should change to meet some scripted design goal.

For example, if you surround some text with an `<em>` tag, the browser by default turns that text into an italic font style. This setting is not reflected in the element's `style` object (`fontStyle` property) because the italic setting was not set via CSS; by contrast, the `element` object's `currentStyle.fontStyle` property reveals the true, current `fontStyle` property of the element as italic.

Example

To change a `style` property setting, access it via the element's `style` object. Use The Evaluator (see Chapter 4) to compare the properties of the `currentStyle` and `style` objects of an element. For example, an unmodified copy of The Evaluator contains an `em` element whose ID is `"myEM"`. Enter `document.getElementById("myEM").style` in the bottom property listing text box and press Enter. Notice that most of the property values are empty. Now enter `document.getElementById("myEM").currentStyle` in the property listing text box and press Enter. Every property has a value associated with it.

Related Items: `runtimeStyle`, `style` objects (Chapter 38); `window.getComputedStyle()` for W3C DOM browsers (Chapter 27)

dateTime

Value: Date string Read-Only

Compatibility: WinIE6+, MacIE-, NN6+, Moz+, Safari+, Opera+, Chrome+

The `dateTime` property contains a date/time value that is used to establish a timestamp for an element. Similar to the `cite` property, the `dateTime` property is intended to apply to a lesser number of element objects (`del` and `ins`) than is actually supported in IE. This may or may not be a mistake, so it's probably not a safe bet to use the property outside its intended elements.

dataFld
dataFormatAs
dataSrc

Value: String Read/Write

Compatibility: WinIE4+, MacIE5, NN-, Moz-, Safari-, Opera-, Chrome-

The `dataFld`, `dataFormatAs`, and `dataSrc` properties (along with more element-specific properties such as `dataPageSize` and `recordNumber`) are part of the Internet Explorer data-binding facilities based on ActiveX controls. The Win32 versions of IE4 and later have several ActiveX objects built into the browsers that facilitate direct communication between a web page and a data source. Data sources include text files, XML data, HTML data, and external databases (MacIE supports text files only). Data binding is a very large topic, much of which extends more to discussions about Microsoft Data Source Objects (DSOs), ODBC, and JDBC — subjects well beyond the scope of this book. But data binding is a powerful tool and can be of use even if you are not a database guru. Therefore, this discussion of the three primary properties — `dataFld`, `dataFormatAs`, and `dataSrc` — briefly covers data binding through Microsoft's *Tabular Data Control DSO*. This allows any page to access, sort, display, and filter (but not update) data downloaded into a web page from an external text file (commonly, comma- or tab-delimited data).

You can load data from an external text file into a document with the help of the *Tabular Data Control* (TDC). You retrieve the data by specifying the TDC object within an `<object>` tag set and specifying additional parameters, such as the URL of the text file and field delimiter characters. The `object`

element can go anywhere within the body of your document. (We tend to put it at the bottom of the code so that all normal page rendering happens before the control loads.) Retrieving the data simply brings it into the browser and does not, on its own, render the data on the page.

If you haven't worked with embedded objects in IE, the classid attribute value might seem a bit strange. The most perplexing part to some is the long value of numeric data signifying the *Globally Unique Identifier* (GUID) for the object, which is IE's way of uniquely identifying objects. You must enter this value exactly as shown in the following example for the proper ActiveX TDC to run. The HTML syntax for this object is as follows:

```
<object id="objName" classid="clsid:333C7BC4-460F-11D0-BC04-0080C7055A83">
   <param name="DataURL" value="URL">
   [additional optional parameters]
</object>
```

Table 26-3 lists the parameters available for the TDC. Only the DataURL parameter is required. Other parameters — such as FieldDelim, UseHeader, RowDelim, and EscapeChar — may be helpful, depending on the nature of the data source.

TABLE 26-3

Tabular Data Control Parameters

Parameter	Description
CharSet	Character set of the data source file. Default is latin1.
DataURL	URL of data source file (relative or absolute).
EscapeChar	Character used to escape delimiter characters that are part of the data. Default is empty. A common value is "\".
FieldDelim	Delimiter character between fields within a record. Default is comma (,). For a Tab character, use a value of .
Language	ISO language code of source data. Default is en-us.
TextQualifier	Optional character surrounding a field's data. Default is empty.
RowDelim	Delimiter character between records. Default is newline (NL).
UseHeader	Set to true if the first row of data in the file contains field names. Default is false.

The value you assign to the object element's id attribute is the identifier that your scripts use to communicate with the data after the page and data completely load. Therefore, you can have as many uniquely named TDCs loaded in your page as there are data source files you want to access at the same time.

The initial binding of the data to HTML elements usually comes when you assign values to the datasrc and datafld attributes of the elements. The datasrc attribute points to the dso identifier (matching the id attribute of the object element, preceded by a hash symbol), whereas the datafld attribute points to the name of the field whose data should be extracted. When you use

data binding with an interactive element such as a table, multiple records are displayed in consecutive rows of the table (more about this in a moment).

Adjust the `dataSrc` and `dataFld` properties if you want the same HTML element (other than a table) to change the data that it displays. These properties apply to a subset of HTML elements that can be associated with external data: `a`, `applet`, `body`, `button`, `div`, `frame`, `iframe`, `img`, `input` (most types), `label`, `marquee`, `object`, `param`, `select`, `span`, and `textarea` objects.

In some cases, your data source may store chunks of HTML-formatted text for rendering inside an element. Unless directed otherwise, the browser renders a data source field as plain text — even if the content contains HTML formatting tags. But if you want the HTML to be observed during rendering, you can set the `dataFormatAs` property (or, more likely, the `dataformatas` attribute of the tag) to `HTML`. The default value is `text`.

Example

Listing 26-8 is a simple document that has two TDC objects associated with it. The external files are different formats of the U.S. Bill of Rights document. One file is a traditional, tab-delimited data file consisting of only two records. The first record is a tab-delimited sequence of field names (named `"Article1"`, `"Article2"`, and so on). The second record is a tab-delimited sequence of article content defined in HTML:

```
<h1>ARTICLE I</h1>
<p>Congress shall make...</p>
```

The second file is a raw-text file consisting of the full Bill of Rights with no HTML formatting attached.

When you load Listing 26-8, only the first article of the Bill of Rights appears in a blue-bordered box. Buttons enable you to navigate to the previous and next articles in the series. Because the data source is a traditional, tab-delimited file, the `nextField()` and `prevField()` functions calculate the name of the next source field and assign the new value to the `dataFld` property. All of the data is already in the browser after the page loads, so cycling through the records is as fast as the browser can reflow the page to accommodate the new content.

LISTING 26-8

Binding Data to a Page

HTML: jsb26-08.html

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Data Binding</title>
      <style type="text/css">
         #display
         {
            width:500px; border:10px ridge blue; padding:20px;
         }
         .hiddenControl
```

continued

LISTING 26-8 *(continued)*

```
         {
            display:none;
         }
      </style>
      <script type="text/javascript" src="../jsb-global.js"></script>
      <script type="text/javascript" src="jsb26-08.js"></script>
   </head>
   <body>
      <h1>U.S. Bill of Rights</h1>
      <form>
         <input type="button" value="Toggle Complete/Individual"
                onclick="toggleComplete()" />
         <span id="buttonWrapper" class="">
            <input type="button" value="Prev" onclick="prevField()" />
            <input type="button" value="Next" onclick="nextField()" />
         </span>
      </form>
      <div id="display" datasrc="#rights_html" datafld="Article1"
           dataformatas="HTML">
      </div>
      <object id="rights_html"
              classid="clsid:333C7BC4-460F-11D0-BC04-0080C7055A83">
         <param name="DataURL" value="Bill of Rights.txt" />
         <param name="UseHeader" value="True" />
         <param name="FieldDelim" value="&#09;" />
      </object>
      <object id="rights_raw"
              classid="clsid:333C7BC4-460F-11D0-BC04-0080C7055A83">
         <param name="DataURL" value="Bill of Rights (no format).txt" />
         <param name="FieldDelim" value="\" />
         <param name="RowDelim" value="\" />
      </object>
   </body>
</html>
```

JavaScript: jsb26-08.js

```
function nextField()
{
   var elem = document.getElementById("display");
   var fieldName = elem.dataFld;
   var currFieldNum = parseInt(fieldName.substring(7, fieldName.length),10);
   currFieldNum = (currFieldNum == 10) ? 1 : ++currFieldNum;
   elem.dataFld = "Article" + currFieldNum;
}
function prevField()
{
   var elem = document.getElementById("display");
   var fieldName = elem.dataFld;
   var currFieldNum = parseInt(fieldName.substring(7, fieldName.length),10);
```

```
   currFieldNum = (currFieldNum == 1) ? 10 : --currFieldNum;
   elem.dataFld = "Article" + currFieldNum;
}

function toggleComplete()
{
   if (document.getElementById("buttonWrapper").className == "")
   {
      document.getElementById("display").dataSrc = "#rights_raw";
      document.getElementById("display").dataFld = "column1";
      document.getElementById("display").dataFormatAs = "text";
      document.getElementById("buttonWrapper").className = "hiddenControl";
   }
   else
   {
      document.getElementById("display").dataSrc = "#rights_html";
      document.getElementById("display").dataFld = "Article1";
      document.getElementById("display").dataFormatAs = "HTML";
      document.getElementById("buttonWrapper").className = "";
   }
}
```

Another button on the page enables you to switch between the initial piecemeal version of the document and the unformatted version in its entirety. To load the entire document as a single record, the `FieldDelim` and `RowDelim` parameters of the second `object` element eliminate their default values by replacing them with characters that don't appear in the document at all. And because the external file does not have a field name in the file, the default value (`column1` for the lone column in this document) is the data field. Thus, in the `toggleComplete()` function, the `dataSrc` property is changed to the desired `object` element ID; the `dataFld` property is set to the correct value for the data source; and the `dataFormatAs` property is changed to reflect the different intention of the source content (to be rendered as HTML or as plain text). When the display shows the entire document, you can hide the two radio buttons by assigning a `className` value to the `span` element that surrounds the buttons. The `className` value is the identifier of the class selector in the document's style sheet. When the `toggleComplete()` function resets the `className` property to empty, the default properties (normal inline display style) take hold.

One further example demonstrates the kind of power available to the TDC under script control. Listing 26-9 displays table data from a tab-delimited file of Academy Awards information. The data file has eight columns of data, and each column heading is treated as a field name: Year, Best Picture, Best Director, Best Director Film, Best Actress, Best Actress Film, Best Actor, and Best Actor Film. For the design of the page, only five fields from each record appear: Year, Film, Director, Actress, and Actor. Notice in the listing that the HTML for the table and its content is bound to the data source object and the fields within the data.

The dynamic part of this example is apparent in how you can sort and filter the data, after it is loaded into the browser, without further access to the original source data. The TDC object features `Sort` and `Filter` properties that enable you to act on the data currently loaded in the browser. The simplest kind of sorting indicates on which field (or fields, via a semicolon-delimited list of field names) the entire data set should be sorted. Leading the name of the sort field is either a plus (to indicate ascending) or minus (descending) symbol. After setting the `data` object's `Sort` property, invoke its

Reset() method to tell the object to apply the new property. The data in the bound table is immediately redrawn to reflect any changes.

Similarly, you can tell a data collection to display records that meet specific criteria. In Listing 26-9, two select lists and a pair of radio buttons provide the interface to the Filter property's settings. For example, you can filter the output to display only those records in which the Best Picture was the same picture of the winning Best Actress's performance. Simple filter expressions are based on field names:

```
dataObj.Filter = "Best Picture" = "Best Actress Film";
```

LISTING 26-9

Sorting Bound Data

HTML: jsb26-09.html

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Data Binding - Sorting</title>
      <style type="text/css">
         thead
         {
            background-color:yellow; text-align:center;
         }
      </style>
      <script type="text/javascript" src="../jsb-global.js"></script>
      <script type="text/javascript" src="jsb26-09.js"></script>
   </head>
   <body>
      <h1>Academy Awards 1978-2005</h1>
      <form>
        <p>Sort list by year
          <a href="javascript:sortByYear('normal')">from newest to oldest</a> or
          <a href="javascript:sortByYear('reverse')">from oldest to newest</a>.
        </p>
        <p>Filter listings for records whose
           <select name="filter1" onchange="filterInCommon(this.form)">
               <option value="Best Picture">Best Picture</option>
               <option value="Best Director Film">Best Director's Film</option>
               <option value="Best Actress Film">Best Actress' Film</option>
               <option value="Best Actor Film">Best Actor's Film</option>
           </select>
           <input type="radio" name="operator" checked="checked"
                   onclick="filterInCommon(this.form)" />is
           <input type="radio" name="operator"
                   onclick="filterInCommon(this.form)" />is not
           <select name="filter2" onchange="filterInCommon(this.form)">
```

```
            <option value="Best Picture">Best Picture</option>
            <option value="Best Director Film">Best Director's Film</option>
            <option value="Best Actress Film">Best Actress' Film</option>
            <option value="Best Actor Film">Best Actor's Film</option>
         </select>
      </p>
     </form>
     <table datasrc="#oscars" border="1" align="center">
        <thead>
           <tr>
              <td>Year</td>
              <td>Film</td>
              <td>Director</td>
              <td>Actress</td>
              <td>Actor</td>
           </tr>
        </thead>
        <tr>
           <td><div id="col1" datafld="Year"></div></td>
           <td><div id="col2" datafld="Best Picture"></div></td>
           <td><div id="col3" datafld="Best Director"></div></td>
           <td><div id="col4" datafld="Best Actress"></div></td>
           <td><div id="col5" datafld="Best Actor"></div></td>
        </tr>
     </table>
     <object id="oscars" classid="clsid:333C7BC4-460F-11D0-BC04-0080C7055A83">
        <param name="DataURL" value="Academy Awards.txt" />
        <param name="UseHeader" value="True" />
        <param name="FieldDelim" value="&#09;" />
     </object>
  </body>
</html>
```

JavaScript: jsb26-09.js

```
function sortByYear(type)
{
   oscars.Sort = (type == "normal") ? "-Year" : "+Year";
   oscars.Reset();
}
function filterInCommon(form)
{
   var filterExpr1 = form.filter1.options[form.filter1.selectedIndex].value;
   var filterExpr2 = form.filter2.options[form.filter2.selectedIndex].value;
   var operator = (form.operator[0].checked) ? "=" : "<>";
   var filterExpr = filterExpr1 + operator + filterExpr2;
   oscars.Filter = filterExpr;
   oscars.Reset();
}
```

Related Items: `recordNumber`, `table.dataPageSize` properties

dir

Value: `"ltr" | "rtl"` Read/Write

Compatibility: WinIE5+, MacIE5+, NN6+, Moz+, Safari+, Opera+, Chrome+

The `dir` property (based on the `dir` attribute of virtually every text-oriented HTML element) controls whether an element's text is rendered left to right (the default) or right to left. By and large, this property (and HTML attribute) is necessary only when you need to override the default directionality of a language's character set as defined by the Unicode standard.

Example

Changing this property value in a standard U.S. version of a browser only makes the right margin the starting point for each new line of text (in other words, the characters are not rendered in reverse order). You can experiment with this in The Evaluator by entering the following statements in the expression evaluation field:

```
document.getElementById("myP").dir = "rtl"
```

Related Item: `lang` property

disabled

Value: Boolean Read/Write

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

Though only form elements have a `disabled` property in IE4 and IE5, this property is associated with every HTML element in IE5.5+. W3C DOM browsers apply the property only to form control and `style` element objects. Disabling an HTML element (like form elements) usually gives the element a dimmed look, indicating that it is not active. A disabled element does not receive any events. It also cannot receive focus, either manually or by script. But a user can still select and copy a disabled body text element.

Note

If you disable a form control element, the element's data is not submitted to the server with the rest of the form elements. If you need to keep a form control locked down but still submit it to the server, use the `form` element's `onsubmit` event handler to enable the form control right before the form is submitted. ■

Example

Use The Evaluator (see Chapter 4) to experiment with the `disabled` property on both form elements (IE4+ and W3C) and regular HTML elements (WinIE5.5+). For IE4+ and W3C browsers, see what happens when you disable the output text area by entering the following statement in the top text box:

```
document.forms[0].output.disabled = true
```

The text area is disabled for user entry, although you can still set the field's `value` property via script (which is how the `true` returned value got there).

If you have WinIE5.5+, disable the `myP` element by entering the following statement in the top text box:

```
document.getElementById("myP").disabled = true
```

The sample paragraph's text turns gray.

Related Item: `isDisabled` property

document

Value: `document` object Read-Only

Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-, Opera+, Chrome-

In the context of HTML element objects as exposed in IE4+/Opera, the `document` property is a reference to the document that contains the object. Though it is unlikely that you will need to use this property, `document` may come in handy for complex scripts and script libraries that handle objects in a generic fashion and do not know the reference path to the document containing a particular object. You might need a reference to the document to inspect it for related objects. The W3C version of this property is `ownerDocument`.

Example

The following simplified function accepts a parameter that can be any object in a document hierarchy. The script finds out the reference of the object's containing document for further reference to other objects:

```
function getCompanionFormCount(obj)
{
   var ownerDoc = obj.document;
   return ownerDoc.forms.length;
}
```

Because the `ownerDoc` variable contains a valid reference to a `document` object, the `return` statement uses that reference to return a typical property of the document object hierarchy.

Related Item: `ownerDocument` property

filters[]

Value: Array Read-Only

Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-, Opera-, Chrome-

Filters are IE-specific style sheet add-ons that offer a greater variety of font rendering (such as drop shadows) and transitions between hidden and visible elements. Each filter specification is a `filter` object. The `filters` property contains an array of `filter` objects defined for the current element. You can apply filters to the following set of elements: `bdo`, `body`, `button`, `fieldset`, `img`, `input`, `marquee`, `rt`, `ruby`, `table`, `td`, `textarea`, `th`, and positioned `div` and `span` elements. See Chapter 38 for details about style sheet filters.

Related Item: `filter` object

firstChild
lastChild

Value: Node object reference Read-Only

Compatibility: WinIE5+, MacIE5+, NN6+, Moz+, Safari+, Opera+, Chrome+

W3C DOM-based DOMs are built around an architecture known as a *node map*. Each object defined by HTML is a node in the map. A node has relationships with other nodes in the document — relationships described in family terms of parents, siblings, and children.

A *child node* is an element that is contained by another element. The container is the parent of such a child. Just as an HTML element can contain any number of child elements, so can a parent object have zero or more children. A list of those children (returned as an array) can be read from an object by way of its `childNodes` property:

```
var nodeArray = document.getElementById("elementID").childNodes;
```

Though you can use this array (and its `length` property) to get a reference to the first or last child node, the `firstChild` and `lastChild` properties offer shortcuts to those positions. These are helpful when you wish to insert a new child before or after all of the others, and you need a reference point for methods that add elements to the document's node list.

Example

Listing 26-10 contains an example of how to use the `firstChild` and `lastChild` properties to access child nodes. These two properties come in handy in this example, which adds and replaces `li` elements to an existing `ol` element. You can enter any text you want to appear at the beginning or end of the list. Using the `firstChild` and `lastChild` properties simplifies access to the ends of the list. For the functions that replace child nodes, the example uses the `replaceChild()` method. Alternatively for IE4+, you can modify the `innerText` property of the objects returned by the `firstChild` or `lastChild` property. This example is especially interesting to watch when you add items to the list: The browser automatically renumbers items to fit the current state of the list.

Caution

See the discussion of the `childNodes` property earlier in this chapter for details about the presence of phantom nodes in some browser versions. The problem may influence your use of the `firstChild` and `lastChild` properties. ■

Note

As handy as it may be, in a strict W3C approach to JavaScript, you wouldn't use the `innerHTML` property because it isn't officially part of the W3C standard. However, it is often too powerful a convenience property to ignore, as much of the code throughout this book is a testament. The book does show the W3C node manipulation alternative to `innerHTML` in some examples. Refer to Chapter 29 for a thorough explanation and examples of the W3C alternative to `innerHTML`. ■

LISTING 26-10

Using firstChild and lastChild Properties

HTML: jsb26-10.html

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>firstChild and lastChild Properties</title>
      <script type="text/javascript" src="../jsb-global.js"></script>
      <script type="text/javascript" src="jsb26-10.js"></script>
   </head>
   <body>
      <h1>firstChild and lastChild Property Lab</h1>
      <hr />
      <form>
         <label>Enter some text to add to or replace in the OL element:</label>
         <br />
         <input type="text" name="input" size="50" />
         <br />
         <input type="button" value="Insert at Top"
                onclick="prepend(this.form)" />
         <input type="button" value="Append to Bottom"
                onclick="append(this.form)" />
         <br />
         <input type="button" value="Replace First Item"
                onclick="replaceFirst(this.form)" />
         <input type="button" value="Replace Last Item"
                onclick="replaceLast(this.form)" />
      </form>
      <ol id="myList">
         <li>Initial Item 1</li>
         <li>Initial Item 2</li>
         <li>Initial Item 3</li>
         <li>Initial Item 4</li>
      </ol>
   </body>
</html>
```

JavaScript: jsb26-10.js

```
// helper function for prepend() and append()
function makeNewLI(txt)
{
   var newItem = document.createElement("LI");
   newItem.innerHTML = txt;
   return newItem;
}
function prepend(form)
```

continued

LISTING 26-10 *(continued)*

```
{
   var newItem = makeNewLI(form.input.value);
   var firstLI = document.getElementById("myList").firstChild;
   document.getElementById("myList").insertBefore(newItem, firstLI);
}
function append(form)
{
   var newItem = makeNewLI(form.input.value);
   var lastLI = document.getElementById("myList").lastChild;
   document.getElementById("myList").appendChild(newItem);
}
function replaceFirst(form)
{
   var newItem = makeNewLI(form.input.value);
   var firstLI = document.getElementById("myList").firstChild;
   document.getElementById("myList").replaceChild(newItem, firstLI);
}
function replaceLast(form)
{
   var newItem = makeNewLI(form.input.value);
   var lastLI = document.getElementById("myList").lastChild;
   document.getElementById("myList").replaceChild(newItem, lastLI);
}
```

Related Items: `nextSibling`, `parentElement`, `parentNode`, `previousSibling` properties; `appendChild()`, `hasChildNodes()`, `removeChild()`, `removeNode()`, `replaceChild()`, `replaceNode()` methods

height
width

Value: Integer or percentage string Read/Write and Read-Only

Compatibility: WinIE4+, MacIE4+, NN4+, Moz+, Safari+, Opera+, Chrome+

The `height` and `width` properties described here are not the identically named properties that belong to an element's style. Rather, these properties reflect the values normally assigned to `height` and `width` attributes of elements such as `img`, `applet`, `table`, and so on. As such, these properties are accessed directly from the object (for example, `document.getElementById("myTable").width`) rather than through the `style` object (for example, `document.getElementById("myDIV").style.width`). Only elements for which the HTML 4.*x* standard provides `height` and `width` attributes have the corresponding properties.

Values for these properties are either integer pixel values (numbers or strings) or percentage values (strings only). If you need to perform some math on an existing percentage value, use the `parseInt()` function to extract the numeric value for use with math calculations. If an element's `height` and `width` attributes are set as percentage values, you can use the `offsetHeight` and `offsetWidth` properties in many modern browsers to get the rendered pixel dimensions.

Property values are read/write for the `image` object in most recent browser versions because you can resize an `image` object in after the page loads. Properties are read/write for some other objects (such as the `table` object) — but not necessarily all others that support these properties.

In general, you cannot set the value of these properties to something less than is required to render the element. This is particularly true of a table. If you attempt to set the `height` value to less than the amount of pixels required to display the table as defined by its style settings, your changes have no effect (even though the property value retains its artificially low value). For other objects, however, you can set the size to anything you like, and the browser scales the content accordingly (images, for example). If you want to see only a segment of an element (in other words, to crop the element), use a style sheet to set the element's clipping region.

Example

The following example demonstrates how to use the `width` property by increasing the width of a table by 10 percent:

```
var tableW = parseInt(document.getElementById("myTable").width);
document.getElementById("myTable").width = (tableW * 1.1) + "%";
```

Because the initial setting for the `width` attribute of the `table` element is set as a percentage value, the script calculation extracts the number from the percentage width string value. In the second statement, the old number is increased by 10 percent and turned into a percentage string by appending the percentage symbol to the value. The resulting string value is assigned to the `width` property of the table.

Related Items: `clientHeight`, `clientWidth` properties; `style.height`, `style.width` properties

hideFocus

Value: Boolean Read/Write

Compatibility: WinIE5.5+, MacIE-, NN-, Moz-, Safari-, Opera-, Chrome-

In IE for Windows, button types of form controls and links display a dotted rectangle around some part of the element whenever that element has focus. If you set the `tabindex` attribute or `tabIndex` property of any other kinds of elements in IE5+, they, too, display that dotted line when given focus. You can still let an element receive focus but hide that dotted line by setting the `hideFocus` property of the element object to `true` (default value is `false`).

Hiding focus does not disable the element. In fact, if the element about to receive focus is scrolled out of view, the page scrolls to bring the element into view. Form controls that respond to keyboard action (for example, pressing the spacebar to check or uncheck a checkbox control) also continue to work as normal. For some designers, the focus rectangle harms the design goals of the page. The `hideFocus` property gives them more control over the appearance while maintaining consistency of operation with other pages. There is no corresponding HTML attribute for a tag, so you can use an `onload` event handler in the page to set the `hideFocus` property of desired objects after the page loads.

Example

Use The Evaluator (see Chapter 4) to experiment with the `hideFocus` property in WinIE5.5+. Enter the following statement in the top text box to assign a `tabIndex` value to the `myP` element so that by default, the element receives focus and the dotted rectangle:

```
document.getElementById("myP").tabIndex = 1
```

Press the Tab key several times until the paragraph receives focus. Now disable the focus rectangle:

```
document.getElementById("myP").hideFocus = true
```

If you now press the Tab key several times, the dotted rectangle does not appear around the paragraph. To prove that the element still receives focus, scroll the page down to the bottom so that the paragraph is not visible (you may have to resize the window). Click one of the focusable elements at the bottom of the page and then press the Tab key slowly until the Address field toolbar has focus. Press the Tab key once. The page scrolls to bring the paragraph into view, but there is no focus rectangle around the element.

Related Items: `tabIndex` property; `srcollIntoView()` method

id

Value: String (See text)

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

The `id` property returns the identifier assigned to an element's `id` attribute in the HTML code. A script cannot modify the ID of an existing element or assign an ID to an element that lacks one. But if a script creates a new element object, an identifier may be assigned to it by way of the `id` property.

Example

Rarely do you need to access this property in a script — unless you write an authoring tool that iterates through all elements of a page to extract the IDs assigned by the author. You can retrieve an object reference when you know the object's `id` property (via the `document.getElementById(elemID)` method). But if for some reason your script doesn't know the ID of, say, the second paragraph of a document, you can extract that ID as follows:

```
var elemID = document.getElementsByTagName("p")[1].id;
```

Related Item: `className` property

innerHTML
innerText

Value: String Read/Write

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

One way that Internet Explorer exposes the contents of an element is through the `innerHTML` and `innerText` properties. (NN6+/Moz offer only the `innerHTML` property.) All content defined by these inner properties consists of document data that is contained by an element's start and end tags but does not include the tags themselves (see the `outerText` and `outerHTML` properties). Setting these inner properties is a common way to modify a portion of a page's content after the page loads.

The `innerHTML` property contains not only the text content for an element as seen on the page, but also every bit of HTML tagging that is associated with that content. (If there are no tags in the content, the text is rendered as is.) For example, consider the following bit of HTML source code:

```
<p id="paragraph1">"How <em>are</em> you?" he asked.</p>
```

The value of the paragraph object's innerHTML property (document.getElementById ("paragraph1").innerHTML) is

```
"How <em>are</em> you?" he asked.
```

The browser interprets any HTML tags included in a string you assign to an element's innerHTML property as tags. This also means that you can introduce entirely new nested elements (or child nodes in the modern terminology) by assigning a slew of HTML content to an element's innerHTML property. The document's object model adjusts itself to the newly inserted content.

By contrast, the innerText property knows only about the text content of an element container. In the example you just saw, the value of the paragraph's innerText property (document .getElementById("paragraph1").innerText) is

```
"How are you?" he asked.
```

It's important to remember that if you assign a string to the innerText property of an element, and that string contains HTML tags, the tags and their angle brackets appear in the rendered page and are not interpreted as live tags.

The W3C DOM Level 3 adds a textContent property that serves as the standard equivalent of innerText. IE does not support textContent.

Do not modify the innerHTML property to adjust the HTML for frameset, html, head, or title objects. You may modify table constructions through either innerHTML or the various table-related methods that create or delete rows, columns, and cells (see Chapter 41). It is also safe to modify the contents of a cell by setting its innerHTML or innerText property.

When the HTML you insert includes a <script> tag, be sure to include the defer attribute to the opening tag. This goes even for scripts that contain function definitions, which you might consider to be deferred automatically.

The innerHTML property is not supported by the W3C DOM, but it does share widespread support in all modern browsers. You could argue that a pure W3C DOM node manipulation approach is more structured than just assigning HTML code to innerHTML, but the ease of making a single property assignment has so far won out in the practicality of everyday scripting. Whenever possible, the examples in this book use the W3C approach to alter the HTML code for a node, but there are several instances where innerHTML is simply too concise an option to resist.

Example

Listing 26-11 contains an example of how to use the innerHTML and innerText properties to alter dynamically the content within a page. The page generated in the listing contains an h1 element label and a paragraph of text. The purpose is to demonstrate how the innerHTML and innerText properties differ in their intent. Two text boxes contain the same combination of text and HTML tags that replaces the inner content of the paragraph's label.

If you apply the default content of the first text box to the innerHTML property of the label1 object, the italic style is rendered as such for the first word. In addition, the text in parentheses is rendered with the help of the small style sheet rule assigned by virtue of the surrounding <span> tags. But if you apply that same content to the innerText property of the label object, the tags are rendered as is.

Use this as a laboratory to experiment with some other content in both text boxes. See what happens when you insert a <br> tag within some text in both text boxes.

LISTING 26-11

Using innerHTML and innerText Properties

HTML: jsb26-11.html

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>innerHTML and innerText Properties</title>
      <style type="text/css">
         h1
         {
            font-size:18pt; font-weight:bold;
            font-family:"Comic Sans MS", Arial, sans-serif;
         }
         .small
         {
            font-size:12pt; font-weight:400; color:gray;
         }
      </style>
      <script type="text/javascript" src="../jsb-global.js"></script>
      <script type="text/javascript" src="jsb26-11.js"></script>
   </head>
   <body>
      <form>
         <p>
            <input type="text" name="HTMLInput"
                   value="&lt;I&gt;First&lt;/I&gt; Article &lt;span ↩
                       class='small'&gt;(of ten)&lt;/span&gt;"
                   size="50" />
             <input type="button" value="Change Heading HTML"
                    onclick="setGroupLabelAsHTML(this.form)" />
         </p>
         <p>
            <input type="text" name="textInput"
                   value="&lt;I&gt;First&lt;/I&gt; Article &lt;span ↩
                       class='small'&gt;(of ten)&lt;/span&gt;"
                   size="50" />
             <input type="button" value="Change Heading Text"
                    onclick="setGroupLabelAsText(this.form)" />
         </p>
      </form>
      <h1 id="label1">
         ARTICLE I
      </h1>
      <p>Congress shall make no law respecting an establishment of religion, or
         prohibiting the free exercise thereof; or abridging the freedom of
         speech, or of the press; or the right of the people peaceably to
         assemble, and to petition the government for a redress of grievances.
      </p>
```

```
   </body>
</html>
```

JavaScript: jsb26-11.js

```
function setGroupLabelAsText(form)
{
   var content = form.textInput.value;
      if (content)
      {
         document.getElementById("label1").innerText = content;
      }
}
function setGroupLabelAsHTML(form)
{
   var content = form.HTMLInput.value;
      if (content)
      {
         document.getElementById("label1").innerHTML = content;
      }
}
```

Related Items: `outerHTML`, `outerText`, `textContent` properties; `replaceNode()` method

isContentEditable

Value: Boolean Read-Only

Compatibility: WinIE5.5+, MacIE-, NN-, Moz-, Safari 1.2+, Opera+, Chrome+

The `isContentEditable` property returns a Boolean value indicating whether a particular element object is set to be editable (see the discussion of the `contentEditable` property earlier in this chapter). This property is helpful because if a parent element's `contentEditable` property is set to `true`, a nested element's `contentEditable` property likely is set to its default value `inherit`. But because its parent is editable, the `isContentEditable` property of the nested element returns `true`.

Example

Use The Evaluator (see Chapter 4) to experiment with both the `contentEditable` and `isContentEditable` properties on the `myP` and nested `myEM` elements (reload the page to start with a known version). Check the current setting for the `myEM` element by typing the following statement in the top text box:

```
myEM.isContentEditable
```

This value is `false` because no element upward in the element containment hierarchy is set to be editable yet. Next, turn on editing for the surrounding `myP` element:

```
myP.contentEditable = true
```

At this point, the entire `myP` element is editable because its child element is set, by default, to inherit the edit state of its parent. Prove it by entering the following statement in the top text box:

```
myEM.isContentEditable
```

Although the `myEM` element is shown to be editable, no change has accrued to its `contentEditable` property:

```
myEM.contentEditable
```

This property value remains the default `inherit`.

Related Item: `contentEditable` property

isDisabled

Value: Boolean Read-Only

Compatibility: WinIE5.5+, MacIE-, NN-, Moz-, Safari-, Opera-, Chrome-

The `isDisabled` property returns a Boolean value that indicates whether a particular element object is set to be disabled (see the discussion of the `disabled` property earlier in this chapter). This property is helpful; if a parent element's `disabled` property is set to `true`, a nested element's `disabled` property likely is set to its default value of `false`. But because its parent is disabled, the `isDisabled` property of the nested element returns `true`. In other words, the `isDisabled` property returns the actual disabled status of an element regardless of its `disabled` property.

Example

Use The Evaluator (see Chapter 4) to experiment with both the `disabled` and `isDisabled` properties on the `myP` and nested `myEM` elements (reload the page to start with a known version). Check the current setting for the `myEM` element by typing the following statement in the top text box:

```
myEM.isDisabled
```

This value is `false` because no element upward in the element containment hierarchy is set for disabling yet. Next, disable the surrounding `myP` element:

```
myP.disabled = true
```

At this point, the entire `myP` element (including its children) is disabled. Prove it by entering the following statement in the top text box:

```
myEM.isDisabled
```

Although the `myEM` element is shown as disabled, no change has accrued to its `disabled` property:

```
myEM.disabled
```

This property value remains the default `false`.

Related Item: `disabled` property

isMultiLine

Value: Boolean Read-Only

Compatibility: WinIE5.5+, MacIE-, NN-, Moz-, Safari-, Opera-, Chrome-

The `isMultiLine` property returns a Boolean value that reveals whether the element object is capable of occupying or displaying more than one line of text. It is important that this value does not reveal whether the element actually occupies multiple lines; rather, it indicates the potential of doing so. For example, a text `input` element cannot wrap to multiple lines, so its `isMultiLine` property is `false`. However, a `button` element can display multiple lines of text for its label, so it reports `true` for the `isMultiLine` property.

Example

Use The Evaluator (see Chapter 4) to read the `isMultiLine` property for elements on that page. Try the following statements in the top text box:

```
document.body.isMultiLine
document.forms[0].input.isMultiLine
myP.isMultiLine
myEM.isMultiLine
```

All but the text field form control report that they are capable of occupying multiple lines.

isTextEdit

Value: Boolean Read-Only

Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-, Opera-, Chrome-

The `isTextEdit` property reveals whether an object can have a WinIE `TextRange` object created with its content (see Chapter 33 on the CD-ROM). You can create `TextRange` objects from only a limited selection of objects in IE4+ for Windows: `body`, `button`, button, image and radio type `input`, and `textarea`. This property always returns `false` in MacIE.

Example

Good coding practice dictates that your script check for this property before invoking the `createTextRange()` method on any object. A typical implementation is as follows:

```
if (document.getElementById("myObject").isTextEdit)
{
   var myRange = document.getElementById("myObject").createTextRange();
   // [more statements that act on myRange]
}
```

Related Items: `createRange()` method; `TextRange` object (Chapter 33 on the CD-ROM)

lang

Value: ISO language code string Read/Write

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

The `lang` property governs the written language system used to render an element's text content when overriding the default browser's language system. The default value for this property is an empty string unless the corresponding `lang` attribute is assigned a value in the element's tag. Modifying the property value by script control does not appear to have any effect in the current browser implementations.

Example

Values for the `lang` property consist of strings containing valid ISO language codes. Such codes have, at minimum, a primary language code (for example, `"fr"` for French) plus an optional region specifier (for example, `"fr-ch"` for Swiss French). The code to assign a Swiss German value to an element looks like the following:

```
document.getElementById("specialSpan").lang = "de-ch";
```

language

Value: String Read/Write

Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-

IE4+'s architecture allows for multiple scripting engines to work with the browser. Two engines are included with the basic Windows version browser: JScript (compatible with JavaScript) and Visual Basic Scripting Edition (VBScript). The default scripting engine is JScript. But if you wish to use VBScript or some other scripting language in statements that are embedded within event handler attributes of a tag, you can specifically direct the browser to apply the desired scripting engine to those script statements by way of the `language` attribute of the tag. The `language` property provides scripted access to that property. Unless you intend to modify the event handler HTML code and replace it with a statement in VBScript (or any other non-JScript-compatible language installed with your browser), you do not need to modify this property (or read it, for that matter).

Valid values include `JScript`, `javascript`, `vbscript`, and `vbs`. Third-party scripting engines have their own identifier for use with this value. Internet Explorer 5 observes `language="xml"` as well. Be aware that this attribute was deprecated in HTML4.01, so depending on the `DOCTYPE` you use, code using this attribute will not validate. You can accomplish the same objective with `type` (see Chapter 4).

Related Item: `script` element object

lastChild

(See `firstChild`)

length

Value: Integer Read-Only and Read/Write

Compatibility: WinIE3+, MacIE3+, NN2+, Moz+, Safari+, Opera+, Chrome+

The `length` property returns the number of items in an array or collection of objects. Its most common application is as a boundary condition in a `for` loop. Though arrays and collections commonly use integer values as index values (always starting with zero), the `length` value is the actual number of items in the group. Therefore, to iterate through all items of the group, the condition expression should include a less-than (<) symbol rather than a less-than-or-equal (<=) symbol, as in the following:

```
for (var i = 0; i < someArray.length; i++)
{
  // [statements for the loop to process]
}
```

For decrementing through an array (in other words, starting from the last item in the array and working toward the first), the initial expression must initialize the counting variable as the length minus one:

```
for (var i = someArray.length - 1; i >= 0; i--)
{
  // [statements for the loop to process]
}
```

For most arrays and collections, the `length` property is read-only and governed solely by the number of items in the group. But in more recent versions of the browsers, you can assign values to some object arrays (`areas`, `options`, and the `select` object) to create placeholders for data assignments. See the discussions of the `area`, `select`, and `option` element objects for details. A plain JavaScript array can also have its `length` property value modified by script to either trim items from the end of the array or reserve space for additional assignments. See Chapter 18 for more about the `Array` object.

Example

You can try the following sequence of statements in the top text box of The Evaluator to see how the `length` property returns values (and sets them for some objects). Note that some statements work in only some browser versions.

```
(All browsers)  document.forms.length
(All browsers)  document.forms[0].elements.length
(All browsers)    document.images.length
(NN4+)          document.layers.length
(All browsers)          document.all.length
(IE5+, W3C)     document.getElementById("myTable").childNodes.length
```

All of these statements are shown primarily for completeness. Unless you have a good reason to support legacy browsers, the last technique (IE5+, W3C) should be used to access the `length` property.

Related Items: `area`, `select`, `option`, and `Array` objects

localName
namespaceURI
prefix

Value: String   Read-Only

Compatibility: WinIE8+, MacIE-, NN6+, Moz+, Safari+, Opera+, Chrome+

The three properties `localName`, `namespaceURI`, and `prefix` apply to any node in an XML document that associates a namespace URI with an XML tag. Although NN6 exposes all three properties for all element (and node) objects, the properties do not return the desired values. However, Mozilla-based browsers, including NN7+, remedy the situation. To understand better what values these three properties represent, consider the following XML content:

```
<x xmlns:bk='http://bigbooks.org/schema'>
    <bk:title>To Kill a Mockingbird</bk:title>
</x>
```

The element whose tag is `<bk:title>` is associated with the Namespace URI defined for the block, and the element's `namespaceURI` property would return the string `http:// bigbooks .org/schema`. The tag name consists of a prefix (before the colon) and the local name (after the colon). In the preceding example, the `prefix` property for the element defined by the `<bk:title>` tag would be `bk`, whereas the `localName` property would return `title`. The `localName` property of any node returns the same value as its `nodeName` property value, such as `#text` for a text node.

For more information about XML Namespaces, visit `http://www.w3.org/TR/REC-xml-names`.

Related Items: `scopeName`, `tagUrn` properties

nextSibling
previousSibling

Value: Object reference Read-Only

Compatibility: WinIE5+, MacIE5+, NN6+, Moz+, Safari+, Opera+, Chrome+

A *sibling node* is one that is at the same nested level as another node in the hierarchy of an HTML document. For example, the following `p` element has two child nodes (the `em` and `span` elements). Those two child nodes are siblings.

```
<p>MegaCorp is <em>the</em> source of the
   <span class="hot">hottest</span> gizmos.</p>
```

Sibling order is determined solely by the source-code order of the nodes. Therefore, in the previous example, the `em` node has no `previousSibling` property. Meanwhile, the `span` node has no `nextSibling` property (meaning that these properties return `null`). These properties provide another way to iterate through all nodes at the same level.

Example

The following function assigns the same class name to all child nodes of an element:

```
function setAllChildClasses(parentElem, className)
{
   var childElem = parentElem.firstChild;
      while (childElem.nextSibling)
      {
         childElem.className = className;
         childElem = childElem.nextSibling;
      }
}
```

This example is certainly not the only way to achieve the same results. Using a `for` loop to iterate through the `childNodes` collection of the parent element is an equally valid approach.

Related Items: `firstChild`, `lastChild`, `childNodes` properties; `hasChildNodes()`, `insertAdjacentElement()` methods

nodeName

Value: String Read-Only

Compatibility: WinIE5+, MacIE5+, NN6+, Moz+, Safari+, Opera+, Chrome+

For HTML and XML elements, the name of a node is the same as the tag name. The `nodeName` property is provided for the sake of consistency with the node architecture specified by the formal W3C DOM standard. The value, just like the `tagName` property, is an all-uppercase string of the tag name (even if the HTML source code is written with lowercase tags).

Some nodes, such as the text content of an element, do not have a tag. The `nodeName` property for such a node is a special value: `#text`. Another kind of node is an attribute of an element. For an attribute, the `nodeName` is the name of the attribute. See Chapter 25 for more about `Node` object properties.

Example

The following function demonstrates one (not very efficient) way to assign a new class name to every `p` element in an IE5+ document:

```
function setAllPClasses(className)
{
   for (var i = 0; i < document.all.length; i++)
   {
      if (document.all[i].nodeName == "P")
      {
         document.all[i].className = className;
      }
   }
}
```

A more efficient approach uses the `getElementsByTagName()` method to retrieve a collection of all `p` elements and then iterate through them directly.

Related Item: `tagName` property

nodeType

Value: Integer　　　　Read-Only

Compatibility: WinIE5+, MacIE5+, NN6+, Moz+, Safari+, Opera+, Chrome+

The W3C DOM specification identifies a series of constant values that denote categories of nodes. Every node has a value that identifies its type, all browsers currently support the `nodeType` property on all node types as objects, but this has not always been the case. Table 26-4 lists the `nodeType` values; all the values are considered part of the W3C DOM Level 2 specification.

The `nodeType` value is automatically assigned to a node, whether the node exists in the document's HTML source code or is generated on-the-fly via a script. For example, if you create a new element node through any of the ways available by script (for example, by assigning a string encased in HTML tags to the `innerHTML` property or by explicitly invoking the `document.createElement()` method), the new element assumes a `nodeType` of `1`.

Mozilla-based browsers and Safari go one step further in supporting the W3C DOM specification by implementing a set of `Node` object property constants for each of the `nodeType` values. Table 26-5 lists the entire set as defined in the DOM Level 2 specification. Substituting these constants for `nodeType` integers can improve the readability of a script. For example, instead of

```
if (myElem.nodeType == 1)
{
  // [statements to process if true]
}
```

***elementObject*.nodeType**

TABLE 26-4

nodeType Property Values

Value	Description
1	Element node
2	Attribute node
3	Text (#text) node
4	CDATA section node
5	Entity reference node
6	Entity node
7	Processing instruction node
8	Comment node
9	Document node
10	Document type node
11	Document fragment node
12	Notation node

TABLE 26-5

W3C DOM nodeType Constants

Reference	nodeType Value
Node.ELEMENT_NODE	1
Node.ATTRIBUTE_NODE	2
Node.TEXT_NODE	3
Node.CDATA_SECTION_NODE	4
Node.ENTITY_REFERENCE_NODE	5
Node.ENTITY_NODE	6
Node.PROCESSING_INSTRUCTION_NODE	7
Node.COMMENT_NODE	8
Node.DOCUMENT_NODE	9
Node.DOCUMENT_TYPE_NODE	10
Node.DOCUMENT_FRAGMENT_NODE	11
Node.NOTATION_NODE	12

it is much easier to see what's going on with

```
if (myElem.nodeType == Node.ELEMENT_NODE)
{
  // [statements to process if true]
}
```

Example

You can experiment with viewing `nodeType` property values in The Evaluator. The `p` element whose ID is `myP` is a good place to start. The `p` element itself is a `nodeType` of `1`:

```
document.getElementById("myP").nodeType
```

This element has three child nodes: a string of text (`nodeName #text`), an `em` element (`nodeName em`), and the rest of the text of the element content (`nodeName #text`). If you view the `nodeType` of either of the text portions, the value comes back as `3`:

```
document.getElementById("myP").childNodes[0].nodeType
```

Related Item: `nodeName` property

nodeValue

Value: Number, string, or null Read/Write

Compatibility: WinIE5+, MacIE5+, NN6+, Moz+, Safari+, Opera+, Chrome+

For a text node, the `nodeValue` property consists of the actual text for that node. Such a node cannot contain any further nested elements, so the `nodeValue` property offers another way of reading and modifying what Internet Explorer implements as an element's `innerText` property (but in the W3C DOM, you must reference the child text node of an element to get or set its node value).

Of the node types implemented in the W3C DOM–capable browsers, only the text and attribute types have readable values. The `nodeValue` property of an element type of node returns a `null` value. For an attribute node, the `nodeValue` property consists of the value assigned to that attribute. According to the W3C DOM standard, attribute values should be reflected as strings. WinIE5, however, returns values of type `Number` when the value is all numeric characters. Even if you assign a string version of a number to such a `nodeValue` property, it is converted to a `Number` type internally. Other browsers return `nodeValue` values as strings in all cases (and convert numeric assignments to strings).

Example

You can use the `nodeValue` property to carry out practical tasks. As an example, `nodeValue` can be used to increase the width of a `textarea` object by 10 percent. The `nodeValue` is converted to an integer before performing the math and reassignment:

```
function widenCols(textareaElem)
{
   var colWidth = parseInt(textareaElem.attributes["cols"].nodeValue, 10);
   textareaElem.attributes["cols"].nodeValue = (colWidth * 1.1);
}
```

As another example, you can replace the text of an element, assuming that the element contains no further nested elements:

```
function replaceText(elem, newText)
{
   if (elem.childNodes.length == 1 && elem.firstChild.nodeType == 3)
   {
      elem.firstChild.nodeValue = newText;
   }
}
```

The function builds in one final verification that the element contains just one child node and that it is a text type. An alternative version of the assignment statement of the second example uses the `innerText` property in IE with identical results:

```
elem.innerText = newText;
```

You could also use the `textContent` property in any browser except IE to achieve the same concise result:

```
elem.textContent = newText;
```

Related Items: `attributes`, `innerText`, `nodeType` properties

offsetHeight
offsetWidth

Value: Integer Read-Only

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

These properties, which ostensibly report the height and width of any element, have had a checkered history due to conflicts between interpretations of the CSS box model by Microsoft and the W3C. Both properties were invented by Microsoft for IE4. Although they are not currently part of any W3C standard (they are included in the Working Draft of the CSSOM View Module), other modern browsers, including Mozilla-based browsers, WebKit-based browsers, and Presto-based browsers, implement the properties because they're so valuable to scripters.

Assuming that you specify style sheet rules for the width or height of an inline (nonpositioned) element, the `offsetHeight` and `offsetWidth` properties act differently depending on whether the page puts the browser in standards-compatible mode (via the DOCTYPE). More specifically, when IE6+ is set to standards-compatible mode (by DOCTYPE switching, as described in Chapter 25), the properties measure the pixel dimensions of the element's content plus any padding or borders, excluding margins. This is also the default behavior for Mozilla and WebKit, which adhere to the W3C box model. In quirks mode, however, the default IE6+ behavior is to return a height and width of only the element's content, with no accounting for padding, borders, or margins. For versions of IE prior to IE6, this is the only behavior.

Note that for a normal block-level element whose height and width are not specified, the `offsetHeight` is determined by the actual height of the content after all text flows. But the `offsetWidth` always extends the full width of the containing element. Therefore, the `offsetWidth` property does not reveal the rendered width of text content that is narrower than the full parent element width. For example, a `p` element consisting of only a few words may report an `offsetWidth` of many hundreds of pixels because the paragraph's block extends the full width

of the `body` element that represents the containing parent of the `p` element. To find out the actual width of text within a full-width, block-level element, wrap the text within an inline element (such as a `span`), and inspect the `offsetWidth` property of the `span`.

Example

With IE4+, you can substitute the `offsetHeight` and `offsetWidth` properties for `clientHeight` and `clientWidth` in Listing 26-6. The reason is that the two elements in question have their widths hard-wired in style sheets. Thus, the `offsetWidth` property follows that lead rather than observing the default width of the parent (BODY) element.

With IE5+ and W3C browsers, you can use The Evaluator to inspect the `offsetHeight` and `offsetWidth` property values of various objects on the page. Enter the following statements in the top text box:

```
document.getElementById("myP").offsetWidth
document.getElementById("myEM").offsetWidth
document.getElementById("myP").offsetHeight
document.getElementById("myTable").offsetWidth
```

Related Items: `clientHeight`, `clientWidth` properties

offsetLeft
offsetTop

Value: Integer Read-Only

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

The `offsetLeft` and `offsetTop` properties can suffer from the same version vagaries that afflict `offsetHeight` and `offsetWidth` properties when borders, margins, and padding are associated with an element and DOCTYPE switching is a factor. However, the `offsetLeft` and `offsetTop` properties are valuable in providing pixel coordinates of an element within the positioning context of the parent element — even when the elements are not positioned explicitly.

Note

The `offsetLeft` and `offsetTop` properties for positioned elements in MacIE do not return the same values as the `style.left` and `style.top` properties of the same element. See Listing 40-5 on the CD-ROM for an example of how to correct these discrepancies without having to hard-wire the precise pixel differences in your code. ■

The element used as a coordinate context for these properties is whatever element the `offsetParent` property returns. This means that to determine the precise position of any element, you may have to add some code that iterates through the `offsetParent` hierarchy until that property returns `null`.

As with the `offsetHeight` and `offsetWidth` properties, the `offsetLeft` and `offsetTop` properties are not yet part of the W3C DOM specification; they are supported across most browsers because they are convenient for some scriptable Dynamic HTML (DHTML) tasks. Through these two properties, a script can read the pixel coordinates of any block-level or inline element. Measurements are made relative to the `body` element, but this may change in the future. See the discussion later in this chapter about the `offsetParent` property.

Example

The following IE script statements use all four offset dimensional properties to size and position a `div` element so that it completely covers a `span` element located within a `p` element. This can be for a fill-in-the-blank quiz that provides text entry fields elsewhere on the page. As the user gets an answer correct, the blocking `div` element is hidden to reveal the correct answer.

```
document.all.blocker.style.pixelLeft = document.all.span2.offsetLeft
document.all.blocker.style.pixelTop = document.all.span2.offsetTop
document.all.blockImg.height = document.all.span2.offsetHeight
document.all.blockImg.width = document.all.span2.offsetWidth
```

Because the `offsetParent` property for the `span` element is the `body` element, the positioned `div` element can use the same positioning context (it's the default context, anyway) for setting the `pixelLeft` and `pixelTop` style properties. (Remember that positioning properties belong to an element's `style` object.) The `offsetHeight` and `offsetWidth` properties can read the dimensions of the `span` element (the example has no borders, margins, or padding to worry about) and assign them to the dimensions of the image contained by the blocker `div` element.

This example is also a bit hazardous in some implementations. If the text of `span2` wraps to a new line, the new `offsetHeight` value has enough pixels to accommodate both lines. But the `blockImg` and `blocker div` elements are block-level elements that render as a simple rectangle. In other words, the `blocker` element doesn't turn into two separate strips to cover the pieces of `span2` that spread across two lines.

Related Items: `clientLeft`, `clientTop`, `offsetParent` properties

offsetParent

Value: Object reference Read-Only

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

The `offsetParent` property returns a reference to the object that acts as a positioning context for the current element. Values for the `offsetLeft` and `offsetTop` properties are measured relative to the top-left corner of the `offsetParent` object.

The returned object is usually, but not always, the next outermost block-level container. For most document elements, the `offsetParent` object is the `document.body` object (with exceptions for some elements in some browsers).

Table cells, for example, have different `offsetParent` elements in different browsers:

Browser	td offsetParent
WinIE4	`tr`
WinIE5+/NN7+/Moz-based/WebKit-based/Presto-based	`table`
MacIE	`table`
NN6	`body`

Fortunately, the property behaves predictably for positioned elements in most modern browsers. For example, a first-level positioned element's `offsetParent` element is the `body`; the `offsetParent`

of a nested positioned element (for example, one absolute-positioned div inside another) is the next outer container (in other words, the positioning context of the inner element).

Example

You can use the offsetParent property to help you locate the position of a nested element on the page. Listing 26-12 demonstrates how a script can walk up the hierarchy of offsetParent objects in IE for Windows to assemble the location of a nested element on a page. The goal of the exercise in Listing 26-12 is to position an image at the top-left corner of the second table cell. The entire table is centered on the page.

The onload event handler invokes the setImagePosition() function. The function first sets a Boolean flag that determines whether the calculations should be based on the client or offset sets of properties. WinIE4 and MacIE5 rely on client properties, whereas WinIE5+ works with the offset properties. The discrepancies even out, however, with the while loop. This loop traverses the offsetParent hierarchy starting with the offsetParent of the cell out to the document.body object. In IE5+ and other browsers, the while loop executes only three times (for the td, table, and body elements) because just the table element exists between the cell and the body; in IE4, the loop executes four times to account for the tr and table elements up the hierarchy. The loop terminates when the offsetParent is null. Finally, the cumulative values of left and top measures are applied to the positioning properties of the div object's style, and the image is made visible.

LISTING 26-12

Using the offsetParent Property

HTML: jsb26-12.html

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>offsetParent Property</title>
      <style type="text/css">
         #myImg
         {
            position:absolute; height:12px; width:12px;
         }
      </style>
      <script type="text/javascript" src="../jsb-global.js"></script>
      <script type="text/javascript" src="jsb26-12.js"></script>
   </head>
   <body onload="setImagePosition()">
      <h1>The offsetParent Property</h1>
      <hr />
      <p>After the document loads, the script positions a small image in the
         upper left corner of the second table cell.
      </p>
      <table border="1" align="center">
         <tr>
            <td>This is the first cell</td>
            <td id="myCell">This is the second cell.</td>
```

continued

LISTING 26-12 *(continued)*

```
         </tr>
      </table>
      <img id="myImg" alt="image" src="end.gif" height="12" width="12"
           style="visibility:hidden;" />
   </body>
</html>
```

JavaScript: jsb26-12.js

```
function setImagePosition()
{
   var x = 0;
   var y = 0;
   var offsetPointer = document.getElementById("myCell"); // cElement;
      while (offsetPointer)
      {
         x += offsetPointer.offsetLeft;
         y += offsetPointer.offsetTop;
         offsetPointer = offsetPointer.offsetParent;
      }
   // correct for MacIE body margin factors
   if (navigator.userAgent.indexOf("Mac") != -1 &&
       typeof document.body.leftMargin != "undefined")
   {
      x += document.body.leftMargin;
      y += document.body.topMargin;
   }
   document.getElementById("myImg").style.left = x + "px";
   document.getElementById("myImg").style.top = y + "px";
   document.getElementById("myImg").style.visibility = "visible";
}
```

Related Items: `offsetLeft`, `offsetTop`, `offsetHeight`, `offsetWidth` properties

outerHTML
outerText

Value: String Read/Write

Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari 1.3+, Opera+, Chrome+

One way that Internet Explorer, WebKit-based browsers, and Presto-based browsers expose an entire element to scripting is by way of the `outerHTML` and `outerText` properties. The primary distinction between these two properties is that `outerHTML` includes the element's start and end tags, whereas `outerText` includes only rendered text that belongs to the element (including text from any nested elements).

The `outerHTML` property contains not only the text content for an element as seen on the page, but also every bit of HTML tagging associated with that content. For example, consider the following bit of HTML source code:

```
<p id="paragraph1">"How <em>are</em> you?" he asked.</p>
```

The value of the `p` object's `outerHTML` property (`document.all.paragraph1. outerHTML`) is exactly the same as that of the source code.

The browser interprets any HTML tags in a string that you assign to an element's `outerHTML` property. This means that you can delete (set the property to an empty string) or replace an entire tag with this property. The document's object model adjusts itself to whatever adjustments you make to the HTML in this manner.

In contrast, the `outerText` property knows only about the text content of an element container. In the preceding example, the value of the paragraph's `outerText` property (`document.all.paragraph1.innerText`) is

```
"How are you?" he asked.
```

If this looks familiar, it's because in most cases the `innerText` and `outerText` properties of an existing element return the same strings.

Example

Listing 26-13 demonstrates how to use the `outerHTML` and `outerText` properties to access and modify web-page content dynamically. The page generated by Listing 26-13 (WinIE4+/WebKit-or Presto-based browsers) contains an `h1` element label and a paragraph of text. The purpose is to demonstrate how the `outerHTML` and `outerText` properties differ in their intent. Two text boxes contain the same combination of text and HTML tags that replaces the element that creates the paragraph's label.

If you apply the default content of the first text box to the `outerHTML` property of the `label1` object, the `h1` element is replaced by a `span` element whose `class` attribute acquires a different style sheet rule defined earlier in the document. Notice that the ID of the new `span` element is the same as that of the original `h1` element. This allows the script attached to the second button to address the object. But this second script replaces the element with the raw text (including tags). The element is gone, and any subsequent attempt to change the `outerHTML` or `outerText` properties of the `label1` object causes an error because there is no longer a `label1` object in the document.

Use this laboratory to experiment with some other content in both text boxes.

LISTING 26-13

Using outerHTML and outerText Properties

HTML: jsb26-13.html

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>outerHTML and outerText Properties</title>
      <style type="text/css">
         h1
         {
```

continued

LISTING 26-13 *(continued)*

```
            font-size:18pt; font-weight:bold;
            font-family:"Comic Sans MS", Arial, sans-serif;
         }
         .heading
         {
            font-size:20pt; font-weight:bold;
            font-family:"Arial Black", Arial, sans-serif;
         }
      </style>
      <script type="text/javascript" src="../jsb-global.js"></script>
      <script type="text/javascript" src="jsb26-13.js"></script>
   </head>
   <body>
      <form>
         <p id="myP1">
            <input type="text" name="HTMLInput"
                   value="&lt;span id='label1' class='heading'&gt;↩
                        Article the First&lt;/span&gt;"
                   size="55" />
            <input type="button" value="Change Heading HTML"
                   onclick="setGroupLabelAsHTML(this.form)" />
         </p>
         <p id="myP2">
            <input type="text" name="textInput"
                   value="&lt;span id='label1' class='heading'&gt;↩
                        Article the First&lt;/span&gt;"
                   size="55" />
            <input type="button" value="Change Heading Text"
                   onclick="setGroupLabelAsText(this.form)" />
         </p>
      </form>
      <h1 id="label1">ARTICLE I</h1>
      <p>Congress shall make no law respecting an establishment of religion, or
         prohibiting the free exercise thereof; or abridging the freedom of
         speech, or of the press; or the right of the people peaceably to
         assemble, and to petition the government for a redress of grievances.
      </p>
   </body>
</html>
```

JavaScript: jsb26-13.js

```
function setGroupLabelAsText(form)
{
   var content = form.textInput.value;
      if (content)
      {
         document.getElementById("label1").outerText = content;
      }
}
```

```
function setGroupLabelAsHTML(form)
{
   var content = form.HTMLInput.value;
      if (content)
      {
         document.getElementById("label1").outerHTML = content;
      }
}
```

Related Items: `innerHTML`, `innerText` properties; `replaceNode()` method

ownerDocument

Value: Document object reference Read-Only

Compatibility: WinIE6+, MacIE5+, NN6+, Moz+, Safari+, Opera+, Chrome+

The `ownerDocument` property belongs to any element or node in the W3C DOM. The property's value is a reference to the document node that ultimately contains the element or node. If a script encounters a reference to an element or node (perhaps it has been passed as a parameter to a function), the object's `ownerDocument` property provides a way to build references to other objects in the same document or to access properties and methods of the document objects. IE's proprietary version of this property is simply `document`.

Example

Use The Evaluator (see Chapter 4) to explore the `ownerDocument` property. Enter the following statement in the top text box:

```
document.body.childNodes[5].ownerDocument
```

The result is a reference to the `document` object. You can use that to inspect a property of the document, as shown in the following statement, which you should enter in the top text box:

```
document.body.childNodes[5].ownerDocument.URL
```

This returns the `document.URL` property for the document that owns the child node.

Related Item: `document` object

parentElement

Value: Element object reference or `null` Read-Only

Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari 1.2+, Opera+, Chrome+

The `parentElement` property returns a reference to the next outermost HTML element from the current element. This parent-child relationship of elements is often, but not always, the same as a parent-child node relationship (see the `parentNode` property later in this chapter). The difference is that the `parentElement` property deals only with HTML elements as reflected as document objects, whereas a node is not necessarily an HTML element (for example, an attribute or text chunk).

There is also a distinction between `parentElement` and `offsetParent` properties. The latter returns an element that may be many generations removed from a given element but is the immediate parent with regard to positioning context. For example, a `td` element's `parentElement` property is most likely its enclosing `tr` element, but a `td` element's `offsetParent` property is its `table` element (unless you're dealing with WinIE4).

A script can walk the element hierarchy outward from an element with the help of the `parentElement` property. The top of the parent chain is the `html` element. Its `parentElement` property returns `null`.

Example

You can experiment with the `parentElement` property in The Evaluator. The document contains a `p` element named `myP`. Type each of the following statements from the left column in the top expression evaluation text box and press Enter to see the results.

Expression	IE	WebKit/Presto
`document.getElementById ("myP").tagName`	`p`	`P`
`document.getElementById ("myP").parentElement`	`[object]`	`object HTMLHtmlElement`
`document.getElementById("myP"). parentElement.tagName`	`body`	`body`
`document.getElementById("myP"). parentElement.parentElement`	`[object]`	`object HTMLBodyElement`
`document.getElementById("myP"). parentElement.parentElement.tagName`	`html`	`html`
`document.getElementById("myP"). parentElement.parentElement. parentElement`	`null`	`null`

Related Items: `offsetParent`, `parentNode` properties

parentNode

Value: Node object reference or `null` Read-Only

Compatibility: WinIE5+, MacIE5+, NN6+, Moz+, Safari+, Opera+, Chrome+

`parentNode` is a W3C property returns a reference to the next outermost node that is reflected as an object belonging to the document. For a standard element object, the `parentNode` property is the same as IE/WebKit and Presto browser's `parentElement` because both objects happen to have a direct parent-child node relationship as well as a parent-child element relationship.

Other kinds of content, however, can be nodes, including text fragments within an element. A text fragment's `parentNode` property is the next outermost node or element that encompasses that fragment. A text node object in IE/Opera does not have a `parentElement` property.

Example

Use The Evaluator to examine the `parentNode` property values of both an element and a nonelement node. Begin with the following two statements, and watch the results of each:

```
document.getElementById("myP").parentNode.tagName
document.getElementById("myP").parentElement.tagName (IE/WebKit/Presto browsers)
```

Now examine the properties from the point of view of the first text fragment node of the `myP` paragraph element:

```
document.getElementById("myP").childNodes[0].nodeValue
document.getElementById("myP").childNodes[0].parentNode.tagName
document.getElementById("myP").childNodes[0].parentElement (WebKit only)
```

Notice in IE and Opera that the text node does not have a `parentElement` property.

Related Items: `childNodes`, `nodeName`, `nodeType`, `nodeValue`, `parentElement` properties

parentTextEdit

Value: Element object reference or `null` Read-Only

Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-, Opera-, Chrome-

Only a handful of objects in IE's object model are capable of creating text ranges (see the `TextRange` object in Chapter 33 on the CD-ROM). To find an object's next outermost container capable of generating a text range, use the `parentTextEdit` property. If an element is in the hierarchy, that element's object reference is returned. Otherwise (for example, `document.body.parentTextEdit`), the value is `null`. MacIE always returns a value of `null` because the browser doesn't support the `TextRange` object.

Example

Listing 26-14 contains an example that demonstrates how to use the `parentTextEdit` property to create a text range. The page resulting from Listing 26-14 contains a paragraph of Latin text and three radio buttons that select the size of a paragraph chunk: one character, one word, or one sentence. If you click anywhere within the large paragraph, the `onclick` event handler invokes the `selectChunk()` function. The function first examines which of the radio buttons is selected to determine how much of the paragraph to highlight (select) around the point at which the user clicks.

After the script employs the `parentTextEdit` property to test whether the clicked element has a valid parent capable of creating a text range, it calls on the property again to help create the text range. From there, `TextRange` object methods shrink the range to a single insertion point, move that point to the spot nearest the cursor location at click time, expand the selection to encompass the desired chunk, and select that bit of text.

Notice one workaround for the `TextRange` object's `expand()` method anomaly: If you specify a sentence, IE doesn't treat the beginning of a `p` element as the starting end of a sentence automatically. A camouflaged (white text color) period is appended to the end of the previous element to force the `TextRange` object to expand only to the beginning of the first sentence of the targeted `p` element.

LISTING 26-14

Using the parentTextEdit Property

HTML: jsb26-14.html

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>parentTextEdit Property</title>
      <style type="text/css">
         p
         {
            cursor:hand
         }
         #placeHolder
         {
            color:white;
         }
      </style>
      <script type="text/javascript" src="../jsb-global.js"></script>
      <script type="text/javascript" src="jsb26-14.js"></script>
   </head>
   <body bgcolor="white">
      <form>
         <p>Choose how much of the paragraph is to be selected when you click
            anywhere in it:
            <br />
            <input type="radio" name="chunk" value="character"
                   checked="checked" />Character
            <input type="radio" name="chunk"
                   value="word" />Word
            <input type="radio" name="chunk"
                   value="sentence" />Sentence
            <span id="placeHolder">.</span>
         </p>
      </form>
      <p onclick="selectChunk()">Lorem ipsum dolor sit amet, consectetaur
        elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.
        Ut enim adminim veniam, quis nostrud exercitation ullamco laboris nisi
        ut aliquip ex ea commodo consequat. Duis aute irure dolor in
        reprehenderit involuptate velit esse cillum dolore eu fugiat nulla
        pariatur. Excepteur sint occaecat cupidatat non proident, sunt in
        culpa qui officia deserunt mollit anim id est laborum.
      </p>
   </body>
</html>
```

JavaScript: jsb26-14.js

```
function selectChunk()
{
   var chunk, range;
```

```
    for (var i = 0; i < document.forms[0].chunk.length; i++)
    {
       if (document.forms[0].chunk[i].checked)
       {
          chunk = document.forms[0].chunk[i].value;
          break;
       }
    }
    var x = window.event.clientX;
    var y = window.event.clientY;
    if (window.event.srcElement.parentTextEdit)
    {
       range = window.event.srcElement.parentTextEdit.createTextRange();
       range.collapse();
       range.moveToPoint(x, y);
       range.expand(chunk);
       range.select();
    }
}
```

Related Items: `isTextEdit` property; `TextRange` object (Chapter 33, "Body Text Objects" on the CD-ROM)

prefix

(See `localName`)

previousSibling

(See `nextSibling`)

readyState

Value: String (integer for `OBJECT` object) Read-Only

Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-, Opera-, Chrome-

A script can query an element in IE to find out whether it has loaded all ancillary data (for example, external image files or other media files) before other statements act on that object or its data. The `readyState` property lets you know the loading status of an element.

Table 26-6 lists the possible values and their meanings.

For most HTML elements, this property always returns `complete`. Most of the other states are used by elements such as `img`, `embed`, and `object`, which load external data and even start other processes (such as ActiveX controls) to work.

One word of caution: Do not expect the `readyState` property to reveal whether an object exists in the document (for example, `uninitialized`). If the object does not exist, it cannot have a `readyState` property; the result is a script error for an undefined object. If you want to run a script only after every element and its data are fully loaded, trigger the function by way of the `onload` event handler for the `body` element or the `onreadystatechange` event handler for the object (and check that the `readyState` property is `complete`).

TABLE 26-6

readyState Property Values

HTML Value	OBJECT Value	Description
complete	4	Element and data are fully loaded.
interactive	3	Data may not be loaded fully, but user can interact with element.
loaded	2	Data is loaded, but object may be starting up.
loading	1	Data is loading.
uninitialized	0	Object has not started loading data yet.

Example

To witness a `readyState` property other than `complete` for standard HTML, you can try examining the property in a script that immediately follows an `<img>` tag:

```
...
<img id="myImg" src="someImage.gif">
<script type="text/javaScript">
   alert(document.getElementById("myImg").readyState);
</script>
...
```

Putting this fragment into a document that is accessible across a slow network helps. If the image is not in the browser's cache, you might get the `uninitialized` or `loading` result. The former means that the `img` object exists, but it has not started receiving the image data from the server. If you reload the page, chances are that the image will load instantaneously from the cache, and the `readyState` property will report `complete`.

Related Item: `onreadystatechange` event handler

recordNumber

Value: Integer or `null` Read-Only

Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-, Opera-, Chrome-

Virtually every object has a `recordNumber` property, but it applies only to elements used in Internet Explorer data binding to represent repeated data. For example, if you display 30 records from an external data store in a table, the `tr` element in the table is represented only once in the HTML. However, the browser repeats the table row (and its component cells) to accommodate all 30 rows of data. If you click a row, you can use the `recordNumber` property of the `tr` object to see which record was clicked. A common application of this facility is in data binding situations that allow for updating records. For example, script a table so that clicking an uneditable row of data displays that record's data in editable text boxes elsewhere on the page. If an object is not bound to a data source, or if it is a nonrepeating object bound to a data source, the `recordNumber` property is `null`.

Example

Listing 26-15 shows how to use the `recordNumber` property to navigate to a specific record in a sequence of data. The data source is a small, tab-delimited file consisting of 20 records of Academy Awards data. Thus, the table that displays a subset of the fields is bound to the data source object. Also bound to the data source object are three `span` objects embedded within a paragraph near the top of the page. As the user clicks a row of data, three fields from that clicked record are placed in the bound `span` objects.

The script part of this page is a mere single statement. When the user triggers the `onclick` event handler of the repeated `tr` object, the function receives as a parameter a reference to the `tr` object. The data store object maintains an internal copy of the data in a `recordset` object. One of the properties of this `recordset` object is the `AbsolutePosition` property, which is the integer value of the current record that the data object points to (it can point to only one row at a time, and the default row is the first row). The statement sets the `AbsolutePosition` property of the `recordset` object to the `recordNumber` property for the row that the user clicks. Because the three `span` elements are bound to the same data source, they are immediately updated to reflect the change to the data object's internal pointer to the current record. Notice, too, that the third `span` object is bound to one of the data source fields not shown in the table. You can reach any field of a record because the data source object holds the entire data source content. However, in actual practice, it would be disconcerting for the visitor to your web site to see something like the name of the film the best actor was in when that information is not reflected in the table below (some visitor's might even start muttering "bug").

LISTING 26-15

Using the Data Binding recordNumber Property

HTML: jsb26-15.html

```
<!DOCTYPE html>
<html>
  <head>
    <meta http-equiv="content-type" content="text/html;charset=utf-8">
    <title>Data Binding (recordNumber)</title>
    <style type="text/css">
       #instructions
       {
          fon-weight:bold;
       }
       .filmTitle
       {
          font-style:italic;
       }
       #columnHeaders
       {
          background-color:yellow; text-align:center;
       }
    </style>
    <script type="text/javascript" src="../jsb-global.js"></script>
    <script type="text/javascript" src="jsb26-15.js"></script>
```

continued

LISTING 26-15 *(continued)*

```
  </head>
  <body>
   <p><span id="instructions">Academy Awards 1978-1997</span> (Click on a table
        row to extract data from one record into the following paragraph.)
   </p>
   <p>The award for Best Actor of <span datasrc="#oscars" datafld="Year"></span>
       went to <span datasrc="#oscars" datafld="Best Actor"></span>
       for his outstanding achievement in the film
     <span class="filmTitle" datasrc="#oscars" datafld="Best Actor Film"></span>.
   </p>
   <table border="1" datasrc="#oscars" align="center">
       <thead id="columnHeaders">
          <tr>
             <td>Year</td>
             <td>Film</td>
             <td>Director</td>
             <td>Actress</td>
             <td>Actor</td>
          </tr>
       </thead>
       <tr id="repeatableRow" onclick="setRecNum(this)">
          <td><div id="col1" datafld="Year"></div></td>
          <td><div class="filmTitle" id="col2" datafld="Best Picture"></div></td>
          <td><div id="col3" datafld="Best Director"></div></td>
          <td><div id="col4" datafld="Best Actress"></div></td>
          <td><div id="col5" datafld="Best Actor"></div></td>
       </tr>
    </table>
    <object id="oscars" classid="clsid:333C7BC4-460F-11D0-BC04-0080C7055A83">
       <param name="DataURL" value="Academy Awards.txt" />
       <param name="UseHeader" value="True" />
       <param name="FieldDelim" value="&#09;" />
    </object>
   </body>
</html>
```

JavaScript: jsb26-15.js

```
// set recordset pointer to the record clicked on in the table.
function setRecNum(row)
{
   document.oscars.recordset.AbsolutePosition = row.recordNumber;
}
```

Related Items: `dataFld`, `dataSrc` properties; `table`, `tr` objects (Chapter 41)

runtimeStyle

Value: `style` object Read-Only

Compatibility: WinIE5+, MacIE5+, NN-, Moz-, Safari-, Opera-, Chrome-

You can determine the browser default settings for style sheet attributes with the help of the `runtimeStyle` property. The `style` object that this property returns contains all style attributes and the default settings at the time the page loads. This property does not reflect values assigned to elements by style sheets in the document or by scripts. The default values returned by this property differ from the values returned by the `currentStyle` property. The latter includes data about values that are not assigned explicitly by style sheets yet are influenced by the default behavior of the browser's rendering engine. In contrast, the `runtimeStyle` property shows unassigned style values as empty or zero.

Example

To change a style property setting, access it via the element's `style` object. Use The Evaluator (see Chapter 4) to compare the properties of the `runtimeStyle` and `style` objects of an element. For example, an unmodified copy of The Evaluator contains an `em` element whose ID is `"myEM"`. Enter both

```
document.getElementById("myEM").style.color
```

and

```
document.getElementById("myEM").runtimeStyle.color
```

in the top text box in turn. Initially, both values are empty. Now assign a color to the `style` property via the top text box:

```
document.getElementById("myEM").style.color = "red"
```

If you type the two earlier statements in the top box, you can see that the `style` object reflects the change, whereas the `runtimeStyle` object holds onto its original (empty) value.

Related Items: `currentStyle` property; `style` object (Chapter 38)

scopeName
tagUrn

Value: String Read-Only

Compatibility: WinIE5+, MacIE-, NN-, Moz-, Safari-, Opera-, Chrome-

The `scopeName` property is associated primarily with XML code that is embedded within a document. When you include XML, you can specify one or more XML Namespaces that define the owner of a custom tag name, thus aiming toward preventing conflicts of identical custom tags from different sources in a document.

The XML Namespace is assigned as an attribute of the `<html>` tag that surrounds the entire document:

```
<html xmlns:fred='http://www.someURL.com'>
```

After that, the Namespace value precedes all custom tags linked to that Namespace:

```
<fred:FIRST_Name id="fredFirstName"/>
```

To find out the Namespace owner of an element, you can read the `scopeName` property of that element. For the preceding example, the `scopeName` returns `fred`. For regular HTML elements, the returned value is always `HTML`. The `tagURN` property sits alongside `scopeName` and stores the URI for the namespace.

The `scopeName` property is available only in Win32 and UNIX flavors of IE5+. The comparable properties for `scopeName` and `tagURN` in the W3C DOM are `prefix` and `namespaceURI`.

Example

If you have a sample document that contains XML and a namespace spec, you can use `document.write()` or `alert()` methods to view the value of the `scopeName` property. The syntax is

```
document.getElementById("elementID").scopeName
```

Related Item: `tagUrn` property

scrollHeight
scrollWidth

Value: Integer Read-Only

Compatibility: WinIE4+, MacIE4+, NN7+, Moz1.0.1+, Safari+, Opera+, Chrome+

The `scrollHeight` and `scrollWidth` properties contain the pixel measures of an object, regardless of how much of the object is visible on the page. Therefore, if the browser window displays a vertical scroll bar, and the body extends below the bottom of the viewable space in the window, the `scrollHeight` takes into account the entire height of the body as though you were to scroll downward and see the entire element. For most elements that don't have their own scroll bars, the `scrollHeight` and `scrollWidth` properties have the same values as the `clientHeight` and `clientWidth` properties.

Example

Use The Evaluator (see Chapter 4) to experiment with these two properties of the `textarea` object, which displays the output of evaluations and property listings. To begin, enter the following in the bottom one-line text box to list the properties of the `body` object:

```
document.body
```

This displays a long list of properties for the `body` object. Now enter the following property expression in the top one-line text box to see the `scrollHeight` property of the output `textarea` when it holds the dozens of lines of property listings:

```
document.getElementById("myTextArea").scrollHeight
```

The result, some number probably in the hundreds, is now displayed in the output `textarea`. This means that you can scroll the content of the `output` element vertically to reveal that number of pixels. Click the Evaluate button once more. The result, 13 or 14, is a measure of the `scrollHeight` property of the `textarea` that had only the previous result in it. The scrollable height of that content was only 13 or 14 pixels, the height of the font in the `textarea`. The `scrollWidth` property of the output `textarea` is fixed by the width assigned to the element's `cols` attribute (as calculated by the browser to determine how wide to make the text area on the page).

Related Items: `clientHeight` , `clientWidth` properties; `window.scroll()` method

scrollLeft
scrollTop

Value: Integer Read-Only

Compatibility: WinIE4+, MacIE4+, NN7+, Moz1.0.1+, Safari+, Opera+, Chrome+

If an element is scrollable (in other words, it has its own scroll bars), you can find out how far the element is scrolled in the horizontal and vertical direction via the `scrollLeft` and `scrollTop` properties. These values are pixels. For nonscrollable elements, these values are always zero — even if they are contained by elements that are scrollable. For example, if you scroll a browser window (or frame in a multiframe environment) vertically, the `scrollTop` property of the `body` object is whatever the pixel distance is between the top of the object (now out of view) and the first visible row of pixels of the element. But the `scrollTop` value of a table that is in the document remains zero.

Netscape browsers prior to version 7 (Mozilla) treat scrolling of a `body` element from the point of view of the window. If you want to find out the scrolled offset of the current page in these browsers, use `window.scrollX` and `window.scrollY`.

Scripts that involve tracking mouse events in IE need to take into account the `scrollLeft` and `scrollTop` properties of the body to compensate for scrolling of the page. See the `Event` object in Chapter 32.

Example

Use The Evaluator (see Chapter 4) to experiment with these two properties of the `textarea` object, which displays the output of evaluations and property listings. To begin, enter the following in the bottom one-line text box to list the properties of the `body` object:

```
document.body
```

This displays a long list of properties for the `body` object. Use the `textarea`'s scroll bar to page down a couple of times. Now enter the following property expression in the top one-line text box to see the `scrollTop` property of the output `textarea` after you scroll:

```
document.getElementById("myTextArea").scrollTop
```

The result, some number, is now displayed in the output `textarea`. This means that the content of the `output` element was scrolled vertically. Click the Evaluate button once more. The result, `0`, is a measure of the `scrollTop` property of the `textarea` that had only the previous result in it. There wasn't enough content in the `textarea` to scroll, so the content was not scrolled at all. The `scrollTop` property, therefore, is zero. The `scrollLeft` property of the output is always zero because the `textarea` element is set to wrap any text that overflows the width of the element. No horizontal scroll bar appears in this case, and the `scrollLeft` property never changes.

Related Items: `clientLeft`, `clientTop` properties; `window.scroll()` method

sourceIndex

Value: Integer Read-Only

Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari+, Opera+, Chrome+

The `sourceIndex` property returns the numeric index (zero-based) of the object within the entire document, which is the group of all elements in the document.

Example

Although the operation of this property is straightforward, the sequence of elements exposed by the `document.all` property may not be. To that end, you can use The Evaluator (see Chapter 4) to experiment in IE4+ with the values that the `sourceIndex` property returns to see how the index values of the `document.all` collection follow the source code.

To begin, reload The Evaluator. Enter the following statement in the top text box to set a preinitialized global variable:

```
a = 0
```

When you evaluate this expression, a zero should appear in the Results box. Next, enter the following statement in the top text box:

```
document.all[a].tagName + " [" + a++ + "]"
```

There are a lot of plus signs in this statement, so be sure you enter it correctly. As you successively evaluate this statement (repeatedly click the Evaluate button), the global variable (`a`) is incremented, enabling you to walk through the elements in source-code order. The `sourceIndex` value for each HTML tag appears in square brackets in the Results box. You generally begin with the following sequence:

```
html [0]
head [1]
title [2]
```

You can continue until there are no more elements, at which point an error message appears because the value of `a` exceeds the number of elements in the `document.all` array. Compare your findings against the HTML source code view of The Evaluator.

Related Item: `item()` method

style

Value: `style` object reference Read/Write

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

The `style` property is the gateway to an element's style sheet settings. The property's value is a `style` object whose properties enable you to read and write the style sheet settings for the element. Although scripts do not usually manipulate the `style` object as a whole, it is quite common in a DHTML page for scripts to get or set multiple properties of the `style` object to effect animation, visibility, and all appearance parameters of the element. Note that style properties returned through this object are only those that are explicitly set by the element's `style` attribute or by script.

You can find significant differences in the breadth of properties of the `style` object in different versions of IE and NN. See Chapter 38 for more details on the `style` object.

Example

Most of the action with the `style` property has to do with the `style` object's properties, so you can use The Evaluator here simply to explore the lists of `style` object properties available on as many

DHTML-compatible browsers as you have running. To begin, enter the following statement in the bottom, one-line text box to inspect the `style` property for the `document.body` object:

```
document.body.style
```

Now inspect the `style` property of the table element that is part of the original version of The Evaluator. Enter the following statement in the bottom text box:

```
document.getElementById("myTable").style
```

In both cases, the values assigned to the `style` object's properties are quite limited by default.

Related Items: `currentStyle`, `runtimeStyle` properties; `style` object (Chapter 38)

tabIndex

Value: Integer Read/Write

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

The `tabIndex` property controls where in the tabbing sequence the current object receives focus. This property obviously applies only to elements that can receive focus. IE5+ permits giving focus to more elements than most other browsers, but for all browsers compatible with this property, the primary elements for which you may want to control focus (namely, form input elements) are covered.

In general, browsers treat form elements as focusable elements by default. Nonform elements usually don't receive focus unless you specifically set their `tabIndex` properties (or `tabindex` tag attributes). If you set the `tabIndex` property of one form element to `1`, that element is first in the tabbing order. Meanwhile, the rest fall into source-code tabbing order on successive presses of the Tab key. If you set two elements to, say, `1`, the tabbing proceeds in source-code order for those two elements and then on to the rest of the elements in source-code order starting with the top of the page.

In Internet Explorer and Moz1.8+, you can remove an element from tabbing order entirely by setting its `tabIndex` property to `-1`. Users can still click those elements to make changes to form element settings, but tabbing bypasses the element.

Example

Listing 26-16 contains a sample script that demonstrates how to control the tab order of a form via the `tabIndex` property. This example demonstrates not only the way you can modify the tabbing behavior of a form on-the-fly, but also how to force form elements out of the tabbing sequence entirely. In this page, the top form (named `lab`) contains four elements. Scripts invoked by buttons in the bottom form control the tabbing sequence. Notice that the `tabindex` attributes of all bottom form elements are set to `-1`, which means that these control buttons are not part of the tabbing sequence.

When you load the page, the default tabbing order for the `lab` form control elements (default setting of zero) takes charge. If you start pressing the Tab key, the precise results at first depend on the browser you use. In IE, the Address field is first selected; next, the Tab sequence gives focus to the window (or frame, if this page were in a frameset); finally, the tabbing reaches the `lab` form. Continue pressing the Tab key, and watch how the browser assigns focus to each of the element types. In WebKit- and Presto-based browsers, as well as recent versions of Mozilla, the tabbing begins in the `lab` form. In NN6, however, you must click anywhere on the content to get the Tab key to start working on form controls.

The sample script inverts the tabbing sequence with the help of a `for` loop that initializes two variables that work in opposite directions as the looping progresses. This gives the last element the lowest

tabIndex value. The skip2() function simply sets the tabIndex property of the second text box to -1, removing it from the tabbing entirely (IE used to be the only browser that supported this, but all modern browsers support it). Notice, however, that you can click in the field and still enter text. (See the disabled property earlier in this chapter to see how to prevent field editing.) NN6 does not provide a tabIndex property setting that forces the browser to skip a form control; you should disable the control instead.

LISTING 26-16

Controlling the tabIndex Property

HTML: jsb26-16.html

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>tabIndex Property</title>
      <script type="text/javascript" src="../jsb-global.js"></script>
      <script type="text/javascript" src="jsb26-16.js"></script>
   </head>
   <body>
      <h1>tabIndex Property Lab</h1>
      <hr />
      <form name="lab">
         Text box no. 1: <input type="text" name="text1" />
         <br />
         Text box no. 2: <input type="text" name="text2" />
         <br />
         <input type="button" value="A Button" />
         <br />
         <input type="checkbox" />And a checkbox
      </form>
      <hr />
      <form name="control">
         <input type="button" value="Invert Tabbing Order" tabindex="-1"
                onclick="invert()" />
         <br />
         <input type="button" value="Skip Text box no. 2"
                tabindex="-1" onclick="skip2()" />
         <br />
         <input type="button" value="Reset to Normal Order" tabindex="-1"
                onclick="resetTab()" />
      </form>
   </body>
</html>
```

JavaScript: jsb26-16.js

```
function invert()
{
   var form = document.lab;
   for (var i = 0, j = form.elements.length; i < form.elements.length; i++, j--)
```

```
   {
      form.elements[i].tabIndex = j;
   }
}

function skip2()
{
      document.lab.text2.tabIndex = -1;
}

function resetTab()
{
   var form = document.lab;
   for (var i = 0; i < form.elements.length; i++)
   {
      form.elements[i].tabIndex = 0;
   }
}
```

The final function, `resetTab()`, sets the `tabIndex` property value to zero for all `lab` form elements; this restores the default order.

Related Items: `blur()`, `focus()` methods

tagName

Value: String Read-Only

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

The `tagName` property returns a string of the HTML or XML tag name belonging to the object. All `tagName` values are returned in all-uppercase characters, even if the source code is written in all-lowercase characters or a mixture. This consistency makes it easier to perform string comparisons. For example, you can create a generic function that contains a `switch` statement to execute actions for some tags and not others. The skeleton of such a function looks like the following:

```
function  processObj(objRef)
{
   switch (objRef.tagName)
   {
      case "TR":
         [statements to deal with table row object]
         break;
      case "TD":
         [statements to deal with table cell object]
         break;
      case "COLGROUP":
         [statements to deal with column group object]
         break;
      default:
         [statements to deal with all other object types]
   }
}
```

Example

You can also see the `tagName` property in action in the example associated with the `sourceIndex` property discussed earlier in the chapter. In that example, the `tagName` property is read from a sequence of objects in source-code order.

Related Items: `nodeName` property; `getElementsByTagName()` method

tagUrn

(See `scopeName`)

textContent

Value: String Read/Write

Compatibility: WinIE-, MacIE-, NN-, Moz1.7+, Safari+, Opera+, Chrome+

This property stores the text string of a node, including any combined text nodes within an element. This means that the content of a node is reflected in the `textContent` property as a single string of text even if it has other nested elements, such as `em`. If you replace the content of a node with a string of text by setting the `textContent` property, all previous node content is replaced, including nested elements. You can think of the `textContent` property as the W3C DOM equivalent of IE's `innerText` property.

Related Item: `innerText` property

title

Value: String Read/Write

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

The W3C standard states that you should use the `title` property (and `title` attribute) in an advisory role. Keep in mind that devices for the disabled use the `title` property and `title` attribute, so we recommend you always give them values. Most browsers interpret this role as text assigned to tooltips that pop up momentarily while the cursor rests atop an element. The advantage of having this property available for writing is that your scripts can modify an element's tooltip text in response to other user interaction on the page. A tooltip can provide brief help about the behavior of icons or links on the page. It can also convey a summary of key facts from the destination of a link, thus enabling a visitor to see vital information without having to navigate to the other page.

As with setting the status bar, we don't recommend using tooltips for conveying mission-critical information to the user. Not all users are patient enough to let the pointer pause for the tooltip to appear. On the other hand, a user may be more likely to notice a tooltip when it appears rather than a status-bar message (even though the latter appears instantaneously).

Example

Listing 26-17 provides a glimpse at how you can use the `title` property to establish tooltips for a page. A simple paragraph element has its `title` attribute set to `"First Time!"`, which is what the tooltip displays if you roll the pointer atop the paragraph and pause after the page loads. But an `onmouseover` event handler for that element increments a global variable counter in the script, and the `title` property of the paragraph object is modified with each mouseover action. The `count` value is made part of a string assigned to the `title` property. Notice that there is not a live connection between the `title` property and the variable; instead, the new value explicitly sets the `title` property.

LISTING 26-17

Controlling the title Property

HTML: jsb26-17.html

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>title Property</title>
      <script type="text/javascript" src="../jsb-global.js"></script>
      <script type="text/javascript" src="jsb26-17.js"></script>
   </head>
   <body>
      <h1>title Property Lab</h1>
      <hr />
      <p id="myP" title="First Time!" onmouseover="incrementCount(this)">Roll
         the mouse over this paragraph a few times.
         <br />
         Then pause atop it to view the tooltip.
      </p>
   </body>
</html>
JavaScript: jsb26-17.js
// global counting variable
var count = 0;

function setToolTip(elem)
{
   elem.title = "You have previously rolled atop this paragraph " + count + "
       time(s).";
}

function incrementCount(elem)
{
   count++;
   setToolTip(elem);
}
```

Related Item: `window.status` property

uniqueID

Value: String Read-Only

Compatibility: WinIE5+, MacIE-, NN-, Moz-, Safari-, Opera-, Chrome-

You can let the WinIE5+ browser generate an identifier (`id` property) for a dynamically generated element on the page with the aid of the `uniqueID` property. You should use this feature with care, because the ID it generates at any given time may differ from the ID generated the next time the element is created in the page. Therefore, you should use the `uniqueID` property when your scripts

require an unknown element to have an id property, but the algorithms are not expecting any specific identifier.

To guarantee that an element gets only one ID assigned to it while the object exists in memory, assign the value via the uniqueID property of that same object — not some other object. After you retrieve the uniqueID property of an object, the property's value stays the same no matter how often you access the property again. In general, you assign the value returned by the uniqueID property to the object's id property for other kinds of processing. (For example, the parameter of a getElementById() method requires the value assigned to the id property of an object.)

Example

Listing 26-18 demonstrates the recommended syntax for obtaining and applying a browser-generated identifier for an object. After you enter some text in the text box and click the button, the addRow() function appends a row to the table. The left column displays the identifier generated via the table row object's uniqueID property. IE5+ generates identifiers in the format "ms__idn", where *n* is an integer starting with zero for the current browser session. Because the addRow() function assigns uniqueID values to the row and the cells in each row, the integer for each row is three greater than the previous one. There is no guarantee that future generations of the browser will follow this format, so do not rely on the format or sequence in your scripts.

LISTING 26-18

Using the uniqueID Property

HTML: jsb26-18.html

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Inserting an WinIE5+ Table Row</title>
      <script type="text/javascript" src="../jsb-global.js"></script>
      <script type="text/javascript" src="jsb26-18.js"></script>
   </head>
   <body>
      <table id="myTable" border="1">
         <tr>
            <th>Row ID</th>
            <th>Data</th>
         </tr>
         <tr id="firstDataRow">
            <td>firstDataRow</td>
            <td>Fred</td>
         </tr>
         <tr id="secondDataRow">
            <td>secondDataRow</td>
            <td>Jane</td>
         </tr>
      </table>
      <hr />
```

```
      <form>
         Enter text to be added to the table:
         <br />
         <input type="text" name="input" size="25" />
         <br />
         <input type='button' value='Insert Row'
                onclick='addRow(this.form.input.value)' />
      </form>
   </body>
</html>
```

JavaScript: jsb26-18.js

```
function addRow(item1)
{
   if (item1)
   {
      // assign long reference to shorter var name
      var theTable = document.getElementById("myTable");
      // append new row to the end of the table
      var newRow = theTable.insertRow(theTable.rows.length);
      // give the row its own ID
      newRow.id = newRow.uniqueID;

      // declare cell variable and data variable
      var newCell;
      var newText;

      // an inserted row has no cells, so insert the cells
      newCell = newRow.insertCell(0);
      // give this cell its own id
      newCell.id = newCell.uniqueID;
      // display the row's id as the cell text
      newText = document.createTextNode(newRow.id);
      newCell.appendChild(newText);
      newCell.bgColor = "yellow"
      // re-use cell var for second cell insertion
      newCell = newRow.insertCell(1);
      newCell.id = newCell.uniqueID;
      newText = document.createTextNode(item1);
      newCell.appendChild(newText);
   }
}
```

Related Items: `id` property; `getElementById()` method

unselectable

Value: String constant (`"on"` or `"off"`) Read/Write

Compatibility: WinIE5.5+, MacIE-, NN-, Moz-, Safari-, Opera+, Chrome-

This property controls the selectability of an element — that is, whether the element's content can be selected by the user. You might use this property to prevent a sensitive piece of data from being selected and copied.

Methods

addBehavior("*URL*")

Returns: Integer ID

Compatibility: WinIE5+, MacIE-, NN-, Moz-, Safari-, Opera-, Chrome-

The `addBehavior()` method imports an external Internet Explorer behavior and attaches it to the current object, thereby extending the properties and/or methods of that object. (See Chapter 51 on the CD-ROM for details on IE behaviors.) The sole parameter of the `addBehavior()` method is a URL pointer to the behavior component's code. This component may be in an external file (with an `.htc` extension), in which case the parameter can be a relative or absolute URL. IE also includes a library of built-in (default) behaviors, whose URLs are in the following format:

```
#default#behaviorName
```

Here, `behaviorName` is one of the default behaviors. If the behavior is imported into the document via the `object` tag, the `addBehavior()` method parameter is the ID of that element in the following format:

```
#objectID
```

When you add a behavior, the loading of the external code occurs asynchronously. This means that even though the method returns a value instantly, the behavior is not necessarily ready to work. Only when the behavior is fully loaded can it respond to events or allow access to its properties and methods. Behaviors loaded from external files observe domain security rules.

Example

Listing 26-19a shows what a behavior file looks like. It is the file used to demonstrate the `addBehavior()` method in Listing 26-19b. The behavior component and the HTML page that loads it must come from the same server and domain; they also must load via the same protocol (for example, `http://`, `https://`, and `file://` are mutually exclusive, mismatched protocols).

LISTING 26-19a

The makeHot.htc Behavior Component

```
<PUBLIC:ATTACH EVENT="onmousedown" ONEVENT="makeHot()" />
<PUBLIC:ATTACH EVENT="onmouseup" ONEVENT="makeNormal()" />
<PUBLIC:PROPERTY NAME="hotColor" />
<PUBLIC:METHOD NAME="setHotColor" />
<script type="text/jscript">
   var oldColor
   var hotColor = "red"

   function setHotColor(color)
   {
```

```
      hotColor = color
   }

   function makeHot()
   {
      if (event.srcElement == element)
      {
         oldColor = style.color
         runtimeStyle.color = hotColor
      }
   }

   function makeNormal()
   {
      if (event.srcElement == element)
      {
         runtimeStyle.color = oldColor
      }
   }
</script>
```

The object to which the component is attached is a simple paragraph object, shown in Listing 26-19b. When the page loads, the behavior is not attached, so clicking the paragraph text has no effect.

When you turn on the behavior by invoking the `turnOn()` function, the `addBehavior()` method attaches the code of the `makeHot.htc` component to the `myP` object. At this point, the `myP` object has one more property, one more method, and two more event handlers that are written to be made public by the component's code. If you want the behavior to apply to more than one paragraph in the document, you have to invoke the `addBehavior()` method for each paragraph object.

After the behavior file is instructed to start loading, the `setInitialColor()` function is called to set the new color property of the paragraph to the user's choice from the `select` list. But this can happen only if the component is fully loaded. Therefore, the function checks the `readyState` property of `myP` for completeness before invoking the component's function. If IE is still loading the component, the function is invoked again in 500 milliseconds.

As long as the behavior is loaded, you can change the color used to turn the paragraph hot. The function first ensures that the component is loaded by checking that the object has the new color property. If it does, the method of the component is invoked (as a demonstration of how to expose and invoke a component method). You can also simply set the property value.

LISTING 26-19b

Using addBehavior() and removeBehavior()

HTML: jsb26-19.html

```
<!DOCTYPE html>
<html>
   <head>
```

continued

LISTING 26-19b *(continued)*

```
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>addBehavior() and removeBehavior() Methods</title>
      <script type="text/javascript" src="../jsb-global.js"></script>
      <script type="text/javascript" src="jsb26-19.js"></script>
   </head>
   <body>
      <h1>addBehavior() and removeBehavior() Method Lab</h1>
      <hr />
      <p id="myP">This is a sample paragraph. After turning on the behavior, it
         will turn your selected color when you mouse down anywhere in this
         paragraph.
      </p>
      <form>
         <input type="button" value="Switch On Behavior" onclick="turnOn()" />
         Choose a 'hot' color:
         <select name="colorChoice" onchange="setColor(this, this.value)">
            <option value="red">red</option>
            <option value="blue">blue</option>
            <option value="cyan">cyan</option>
         </select>
         <br />
         <input type="button" value="Switch Off Behavior"
                onclick="turnOff()" />
         <p>
            <input type="button" value="Count the URNs"
                   onclick="showBehaviorCount()" />
         </p>
      </form>
   </body>
</html>
```

JavaScript: jsb26-19.js

```
var myPBehaviorID;

function turnOn()
{
   myPBehaviorID = document.getElementById("myP").addBehavior("makeHot.htc");
   setInitialColor();
}

function setInitialColor()
{
   if (document.getElementById("myP").readyState == "complete")
   {
      var select = document.forms[0].colorChoice;
      var color = select.options[select.selectedIndex].value;
      document.getElementById("myP").setHotColor(color);
   }
   else
```

```
   {
      setTimeout("setInitialColor()", 500);
   }
}

function turnOff()
{
   document.getElementById("myP").removeBehavior(myPBehaviorID);
}

function setColor(select, color)
{
   if (document.getElementById("myP").hotColor)
   {
      document.getElementById("myP").setHotColor(color);
   }
   else
   {
      alert("This feature is not available. Turn on the Behavior first.");
      select.selectedIndex = 0;
   }
}

function showBehaviorCount()
{
   var num = document.getElementById("myP").behaviorUrns.length;
   var msg = "The myP element has " + num + " behavior(s). ";
   alert(msg);
}
```

To turn off the behavior, the `removeBehavior()` method is invoked. Notice that the `removeBehavior()` method is associated with the `myP` object, and the parameter is the ID of the behavior added earlier. If you associate multiple behaviors with an object, you can remove one without disturbing the others, because each has its own unique ID.

Related Items: `readyState` property; `removeBehavior()` method; IE behaviors (Chapter 51 on the CD-ROM)

addEventListener("eventType", listenerFunc, useCapture) removeEventListener("eventType", listenerFunc, useCapture)

Returns: Nothing

Compatibility: WinIE-, MacIE-, NN6+, Moz+, Safari+, Opera+, Chrome+

The W3C DOM's event mechanism accommodates both event bubbling and trickling (see Chapter 32). Although the new mechanism supports the long-standing notion of binding an event to an element by way of HTML attributes (for example, the old `onclick` event handler), it encourages binding events by registering an event listener with an element. (In browsers that support the W3C event model, other ways of binding events — such as event handler attributes — are internally converted to registered events.)

To tell the DOM that an element should listen for a particular kind of event, use the `addEventListener()` method on the element object. The method requires three parameters. The first is a string version of the event type for which the element should listen. Event type strings do not include the well-used `on` prefix of event handlers; instead, the names consist only of the event and are usually in all lowercase (except for some special systemwide events preceded by DOM). Table 26-7 shows all the events recognized by the W3C DOM specification (including some new DOM ones that are not yet implemented in browsers).

TABLE 26-7

W3C DOM Event Listener Types

abort	error
blur	focus
change	load
click	mousedown
DOMActivate	mousemove
DOMAttrModified	mouseout
DOMCharacterDataModified	mouseover
DOMFocusIn	mouseup
DOMFocusOut	reset
DOMNodeInserted	resize
DOMNodeInsertedIntoDocument	scroll
DOMNodeRemoved	select
DOMNodeRemovedFromDocument	submit
DOMSubtreeModified	unload

Note that the event types specified in the DOM Level 2 are more limited than the wide range of events defined in IE4+. Also, the W3C temporarily tabled the issue of keyboard events until DOM Level 3. Fortunately, most W3C-compatible browsers implement keyboard events in a fashion that likely will appear as part of the W3C DOM Level 3.

The second parameter of the `addEventListener()` method is a reference to the JavaScript function to be invoked. This is the same form used to assign a function to an event property of an object (for example, `objReference.onclick = someFunction`), and it should *not* be a quoted string. This approach also means that you cannot specify parameters in the function call. Therefore, functions that need to reference forms or form control elements must build their own references (with the help of the event object's property that says which object is the event's target).

By default, the W3C DOM event model has events bubble upward through the element container hierarchy starting with the target object of the event (for example, the button being clicked). However, if you specify `true` for the third parameter of the `addEventListener()` method, event capture is

enabled for this particular event type whenever the current object is the event target. This means that any other event type targeted at the current object bubbles upward unless it, too, has an event listener associated with the object and the third parameter is set to `true`.

Using the `addEventListener()` method requires that the object to which it is attached already exists. Therefore, you most likely will use the method inside an initialization function triggered by the `onload` event handler for the page. (The `document` object can use `addEventListener()` for the load event immediately, because the `document` object exists early in the loading process.)

A script can also eliminate an event listener that was previously added by script. The `removeEventListener()` method takes the same parameters as `addEventListener()`, which means that you can turn off one listener without disturbing others. In fact, because you can add two listeners for the same event and listener function (one set to capture and one not — a rare occurrence indeed), the three parameters of the `removeEventListener()` enable you to specify precisely which listener to remove from an object.

Unlike the event capture mechanism of NN4, the W3C DOM event model does not have a global capture mechanism for an event type regardless of target. And with respect to Internet Explorer, the `addEventListener()` method is closely analogous to the IE5+ `attachEvent()` method. Also, event capture in IE5+ is enabled via the separate `setCapture()` method. Both the W3C and IE event models use their own syntaxes to bind objects to event handling functions, so the actual functions may be capable of serving both models with browser version branching required only for event binding. See Chapter 32 for more about event handling with these two event models.

Example

Listing 26-20 provides a compact workbench to explore and experiment with the basic W3C DOM event model. When the page loads, no event listeners are registered with the browser (except the control buttons, of course). But you can add an event listener for a `click` event in bubble and/or capture mode to the `body` element or the `p` element that surrounds the `span` holding the line of text. If you add an event listener and click the text, you see a readout of the element processing the event and information indicating whether the event phase is bubbling (3) or capture (1). With all event listeners engaged, notice the sequence of events being processed. Remove listeners one at a time to see the effect on event processing.

LISTING 26-20

W3C Event Lab

HTML: jsb26-20.html

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>W3C Event Model Lab</title>
      <style type="text/css">
         td
         {
            text-align:center
         }
      </style>
```

continued

***elementObject*.removeEventListener()**

LISTING 26-20 *(continued)*

```
      <script type="text/javascript" src="../jsb-global.js"></script>
      <script type="text/javascript" src="jsb26-20.js"></script>
   </head>
   <body id="myBODY">
      <h1>W3C Event Model Lab</h1>
      <hr />
      <p id="myP"><span id="mySPAN">This paragraph (a SPAN element nested inside
         a P element) can be set to listen for "click" events.</span>
      </p>
      <hr />
      <form name="controls" id="controls">
         <p>Examine click event characteristics: 
           <input type="button" value="Clear" onclick="clearTextArea()" />
           <br />
           <textarea name="output" cols="80" rows="6" wrap="virtual"></textarea>
         </p>
         <table cellpadding="5" border="1">
            <caption style="font-weight:bold">Control Panel</caption>
            <tr style="background-color:#ffff99">
               <td rowspan="2">"Bubble"-type click listener:</td>
               <td>
                  <input type="button" value="Add to BODY"
                         onclick="addBubbleListener('myBODY')" />
               </td>
               <td>
                  <input type="button" value="Remove from BODY"
                         onclick="removeBubbleListener('myBODY')" />
               </td>
            </tr>
            <tr style="background-color:#ffff99">
               <td>
                  <input type="button" value="Add to P"
                         onclick="addBubbleListener('myP')" />
               </td>
               <td>
                  <input type="button" value="Remove from P"
                         onclick="removeBubbleListener('myP')" />
               </td>
            </tr>
            <tr style="background-color:#ff9999">
               <td rowspan="2">"Capture"-type click listener:</td>
               <td>
                  <input type="button" value="Add to BODY"
                         onclick="addCaptureListener('myBODY')" />
               </td>
               <td>
                  <input type="button" value="Remove from BODY"
                         onclick="removeCaptureListener('myBODY')" />
               </td>
            </tr>
```

```
            <tr style="background-color:#ff9999">
               <td>
                  <input type="button" value="Add to P"
                         onclick="addCaptureListener('myP')" />
               </td>
               <td>
                  <input type="button" value="Remove from P"
                         onclick="removeCaptureListener('myP')" />
               </td>
            </tr>
         </table>
      </form>
   </body>
</html>
```

JavaScript: jsb26-20.js

```
// add event listeners
function addBubbleListener(elemID)
{
   document.getElementById(elemID).addEventListener("click",
     reportEvent, false);
}
function addCaptureListener(elemID)
{
   document.getElementById(elemID).addEventListener("click", reportEvent, true);
}

// remove event listeners
function removeBubbleListener(elemID)
{
   document.getElementById(elemID).removeEventListener("click",
     reportEvent, false);
}
function removeCaptureListener(elemID)
{
   document.getElementById(elemID).removeEventListener("click",
     reportEvent, true);
}

// display details about any event heard
function reportEvent(evt)
{
   var elem = (evt.target.nodeType == 3) ? evt.target.parentNode : evt.target;
   if (elem.id == "mySPAN")
   {
      var msg = "Event processed at " + evt.currentTarget.tagName +
                " element (event phase = " + evt.eventPhase + ").\n";
      document.controls.output.value += msg;
   }
}
```

continued

LISTING 26-20 *(continued)*

```
// clear the details textarea
function clearTextArea()
{
   document.controls.output.value = "";
}
```

Related Items: `attachEvent()`, `detachEvent()`, `dispatchEvent()`, `fireEvent()`, `removeEventListener()` methods

appendChild(*elementObject*)

Returns: Node object reference

Compatibility: WinIE5+, MacIE5+, NN6+, Moz+, Safari+, Opera+, Chrome+

The `appendChild()` method inserts an element or text node (defined by other code that comes before it) as the new, last child of the current element. Aside from the more obvious application of adding a new child element to the end of a sequence of child nodes, the `appendChild()` method is also practical for building element objects and their content before appending, replacing, or inserting the element into an existing document. The `document.createElement()` method generates a reference to an element of whatever tag name you assign as that method's parameter.

The `appendChild()` method returns a reference to the appended node object. This reference differs from the object that is passed as the method's parameter because the returned value represents the object as part of the document rather than as a freestanding object in memory.

Example

Listing 26-21 contains an example that shows how to use the `appendChild()` method in concert with `removeChild()` and `replaceChild()` to modify child elements in a document. Because many W3C DOM browsers treat source-code carriage returns as text nodes (and, thus, child nodes of their parent), the HTML for the affected list elements in Listing 26-21 is shown without carriage returns between elements.

The `append()` function creates a new `li` element and then uses the `appendChild()` method to attach the text box text as the displayed text for the item. The nested expression, `document.createTextNode(newData)`, evaluates to a legitimate node that is appended to the new `li` item. All of this occurs before the new `li` item is added to the document. In the final statement of the function, `appendChild()` is invoked from the vantage point of the `ul` element — thus adding the `li` element as a child node of the `ul` element.

Invoking the `replaceChild()` method in the `rplace()` function uses some of the same code. The main difference is that the `replaceChild()` method requires a second parameter: a reference to the child element to be replaced. This demonstration replaces the final child node of the `ul` list, so the function takes advantage of the `lastChild` property of all elements to get a reference to that final nested child. That reference becomes the second parameter to `replaceChild()`.

LISTING 26-21

Various Child Methods

HTML: jsb26-21.html

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>appendChild(), removeChild(), and replaceChild() Methods</title>
      <script type="text/javascript" src="../jsb-global.js"></script>
      <script type="text/javascript" src="jsb26-21.js"></script>
   </head>
   <body>
      <h1>Child Methods</h1>
      <hr />
      Here is a list of items:
      <ul id="myUL"><li>First Item</li><li>Second Item</li></ul>
      <form>
         Enter some text to add/replace in the list:
         <input type="text" id="newListData" size="30" />
         <br />
         <input type="button" value="Append to List"
                onclick="append()" />
         <input type="button" value="Replace Final Item"
                onclick="rplace()" />
         <input type="button" value="Truncate List to 2 Items"
                onclick="truncateList()" />
      </form>
   </body>
</html>
```

JavaScript: jsb26-21.js

```
function append()
{
   var newData = document.getElementById("newListData").value;
   if (newData)
   {
      var newItem = document.createElement("LI");
      newItem.appendChild(document.createTextNode(newData));
      document.getElementById("myUL").appendChild(newItem);
   }
}

// Opera throws an error when use keyword replace, so....
function rplace()
{
   var newData = document.getElementById("newListData").value;
   if (newData)
   {
```

continued

LISTING 26-21 *(continued)*

```
      var newItem = document.createElement("LI");
      var lastChild = document.getElementById("myUL").lastChild;
      newItem.appendChild(document.createTextNode(newData));
      document.getElementById("myUL").replaceChild(newItem, lastChild);
   }
}

function truncateList()
{
   var oneChild;
   var mainObj = document.getElementById("myUL");
   while (mainObj.childNodes.length > 2)
   {
      oneChild = mainObj.lastChild;
      mainObj.removeChild(oneChild);
   }
}
```

The final part of the demonstration uses the `removeChild()` method to peel away all children of the `ul` element until just two items are left standing. Again, the `lastChild` property comes in handy as the `truncateList()` function keeps removing the last child until only two remain.

Related Items: `removeChild()`, `replaceChild()` methods; nodes and children (Chapter 25)

applyElement(*elementObject*[, *type*])

Returns: Nothing

Compatibility: WinIE5+, MacIE-, NN-, Moz-, Safari-, Opera-, Chrome-

The `applyElement()` method enables you to insert a new element as the parent or child of the current object. An important feature of this method is that the new object is wrapped around the current object (if the new element is to become the parent) or the current object's content (if the new element is to become a child). When the new element becomes a child, all previous children are nested further by one generation to become immediate children of the new element. You can imagine how the resulting action of this method affects the containment hierarchy of the current element, so you must be careful how you use the `applyElement()` method.

One parameter, a reference to the object to be applied, is required. This object may be generated from constructions such as `document.createElement()` or from one of the child or node methods that returns an object. The second parameter is optional, and it must be one of the following values:

Parameter Value	Description
`outside`	New element becomes the parent of the current object.
`inside`	New element becomes the immediate child of the current object.

If you omit the second parameter, the default value (`outside`) is assumed. Listing 26-22 shows how the `applyElement()` method is used both with and without default values.

Example

To help you visualize the impact of the `applyElement()` method with its different parameter settings, Listing 26-22 enables you to apply a new element (an `em` element) to a `span` element inside a paragraph. At any time, you can view the HTML of the entire `p` element to see where the `em` element is applied, as well as its impact on the element containment hierarchy for the paragraph.

After you load the page, inspect the HTML for the paragraph before doing anything else. Notice the `span` element and its nested `code` element, both of which surround the one-word content. If you apply the `em` element inside the `span` element (click the middle button), the `span` element's first (and only) child element becomes the `em` element; the `code` element is now a child of the new `em` element.

LISTING 26-22

Using the applyElement() Method

HTML: jsb26-22.html

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>applyElement() Method</title>
      <style type="text/css">
         #mySpan
         {
            font-size:x-large;
         }
      </style>
      <script type="text/javascript" src="../jsb-global.js"></script>
      <script type="text/javascript" src="jsb26-22.js"></script>
   </head>
   <body>
      <h1>applyElement() Method</h1>
      <hr />
      <p id="myP">A simple paragraph with a <span id="mySpan"><code>
      special</code></span> word in it. How special? Click an apply button,
          then click the show button.
      </p>
      <form>
         <input type="button" value="Apply &lt;EM&gt; Outside"
                onclick="applyOutside()" />
         <input type="button" value="Apply &lt;EM&gt; Inside"
                onclick="applyInside()" />
         <input type="button" value="Show &lt;P&gt; HTML..."
                onclick="showHTML()" />
         <br />
         <input type="button" value="Restore Paragraph"
                onclick="location.reload()" />
```

continued

LISTING 26-22 *(continued)*

```
      </form>
   </body>
</html>
```

JavaScript: jsb26-22.js

```
function applyOutside()
{
   var newItem = document.createElement("EM");
   newItem.id = newItem.uniqueID;
   // no 2nd argument--use default: outside
   document.getElementById("mySpan").applyElement(newItem);
}

function applyInside()
{
   var newItem = document.createElement("EM");
   newItem.id = newItem.uniqueID;
   document.getElementById("mySpan").applyElement(newItem, "inside");
}

function showHTML()
{
   alert(document.getElementById("myP").outerHTML);
}
```

The visible results of applying the `em` element inside and outside the `span` element in this case are the same. But you can see from the HTML results that each element impacts the element hierarchy quite differently.

Related Items: `insertBefore()`, `appendChild()`, `insertAdjacentElement()` methods

attachEvent("*eventName*", *functionRef*)
detachEvent("*eventName*", *functionRef*)

Returns: Boolean

Compatibility: WinIE5+, MacIE-, NN-, Moz-, Safari-, Opera+, Chrome-

The `attachEvent()` method originated as a means to bind events for IE behaviors (see Chapter 51 on the CD-ROM). But the method has gained acceptance as an IE alternative to the W3C `addEventListener()` event binding method. To illustrate the method's usage, we want you to first consider the following example of the typical property assignment approach to binding an event handler:

```
myObject.onmousedown = setHilite;
```

The version with `attachEvent()` is as follows:

```
myObject.attachEvent("onmousedown", setHilite);
```

Both parameters are required. The first parameter is a string version (case insensitive) of the event name. The second is a reference to the function to be invoked when the event fires for this object. A *function reference* is an unquoted, case-sensitive identifier for the function without any parentheses (which also means that you cannot pass parameters in this function call).

There is a subtle benefit to using `attachEvent()` over the event property binding approach. When you use `attachEvent()`, the method returns a Boolean value of `true` if the event binding succeeds. IE triggers a script error if the function reference fails, so don't rely on a returned value of `false` to catch these kinds of errors. Also, there is no validation that the object recognizes the event name.

If you have used `attachEvent()` to bind an event handler to an object's event, you can disconnect that binding with the `detachEvent()` method. The parameters are the same as for `attachEvent()`. The `detachEvent()` method cannot unbind events whose associations are established via tag attributes or event property settings.

The W3C DOM event model provides functionality similar to these IE-only methods: `addEventListener()` and `removeEventListener()`.

Example

Use The Evaluator (see Chapter 4) to create an anonymous function that is called in response to an `onmousedown` event of the first paragraph on the page. Begin by assigning the anonymous function to global variable `a` (already initialized in The Evaluator) in the top text box:

```
a = new Function("alert('Function created at " + (new Date()) + "')")
```

The quote marks and parentheses can get jumbled easily, so enter this expression carefully. When you enter the expression successfully, the Results box shows the function's text. Now assign this function to the `onmousedown` event of the `myP` element by entering the following statement in the top text box:

```
document.getElementById("myP").attachEvent("onmousedown", a)
```

The Results box displays `true` when successful. If you mouse down on the first paragraph after the evaluator panel, an alert box displays the date and time when the anonymous function was created (when the `new Date()` expression was evaluated).

Now disconnect the event relationship from the object by entering the following statement in the top text box:

```
document.getElementById("myP").detachEvent("onmousedown", a)
```

Related Items: `addEventListener()`, `detachEvent()`, `dispatchEvent()`, `fireEvent()`, `removeEventListener()` methods; event binding (Chapter 32)

blur()
focus()

Returns: Nothing

Compatibility: WinIE3+, MacIE3+, NN2+, Moz+, Safari+, Opera+, Chrome+

The `blur()` method removes focus from an element, whereas the `focus()` method gives focus to an element. Even though the `blur()` and `focus()` methods have been around since the earliest scriptable browsers, not every focusable object has enjoyed these methods since the beginning.

***elementObject*.blur()**

Browsers before IE4 and NN6 limited these methods primarily to the `window` object and form control elements.

Windows

For window objects, the `blur()` method (NN3+, IE4+, Opera) pushes the referenced window to the back of all other open windows. If other browser suite windows (such as email or newsreader windows) are open, the window receiving the `blur()` method is placed behind these windows as well.

In Safari and Firefox, the `blur()` method pushes the referenced window only behind the window that opened it. In Chrome, the `window.blur()` method does not work, as demonstrated if you click the Put Me in Back button in Listing 26-23a. For that matter, the `window.focus()` method does not work in Chrome either, as demonstrated if you click the Bring Main to Front button in Listing 26-23a. The window.focus() method does not work in IE8 either.

Caution

When it works, the `window.blur()` method does not consistently adjust the stacking order of the current window in the different browsers. But a script in a window can invoke the `focus()` method of another window to bring that other window to the front (provided that a scriptable linkage, such as the `window.opener` property, exists between the two windows). ■

The minute you create another window for a user in your web-site environment, you must pay attention to window layer management. With browser windows so easily activated by the slightest mouse click, a user can lose a smaller window behind a larger one in a snap. Most inexperienced users don't think to check the Windows taskbar or browser menu bar (if the browser is so equipped) to see whether a smaller window is still open and then activate it. If that subwindow is important to your site design, you should present a button or other device in each window that enables users to switch among windows safely. The `window.focus()` method brings the referenced window to the front of all the windows.

Rather than supply a separate button on your page to bring a hidden window forward, you should build your window-opening functions in such a way that if the window is already open, the function automatically brings that window forward (as shown in Listing 26-23a). This removes the burden of window management from your visitors.

The key to success with this method is making sure that your references to the desired windows are correct. Therefore, be prepared to use the `window.opener` property to refer to the main window if a subwindow needs to bring the main window back into focus.

Form control elements

The `blur()` and `focus()` methods apply primarily to text-oriented form controls: text input, `select`, and `textarea` elements.

Just as a camera lens blurs when it goes out of focus, a text object blurs when it loses focus — when someone clicks or tabs out of the field. Under script control, `blur()` deselects whatever may be selected in the field, and the text insertion pointer leaves the field. The pointer does not proceed to the next field in tabbing order, as it does if you perform a blur by tabbing out of the field manually.

For a text object, having focus means that the text insertion pointer is flashing in that text object's field. Giving a field focus is like opening it up for human editing.

Setting the focus of a text box or `textarea` does not by itself enable you to place the cursor at any specified location in the field. The cursor usually appears at the beginning of the text. To prepare a field for entry to remove the existing text, use both the `focus()` and `select()` methods in series.

There is a caveat about using `focus()` and `select()` together to preselect the content of a text box for immediate editing: Many versions of Internet Explorer fail to achieve the desired results due to an internal timing problem. You can work around this problem (and remain compatible with other browsers) by initiating the focus and selection actions through a `setTimeout()` method.

A common design requirement is to position the insertion pointer at the end of a text box or `textarea` so that a user can begin appending text to existing content immediately. This is possible in IE4+ with the help of the `TextRange` object. The following script fragment moves the text insertion pointer to the end of a `textarea` element whose ID is `myTextarea`:

```
var range = document.getElementById("myTextarea").createTextRange();
range.move("textedit");
range.select();
```

You should be very careful in combining `blur()` or `focus()` methods with `onblur` and `onfocus` event handlers — especially if the event handlers display alert boxes. Many combinations of these events and methods can cause an infinite loop in which it is impossible to dismiss the alert dialog box completely. On the other hand, there is a useful combination for older browsers that don't offer a `disabled` property for text boxes. The following text box event handler can prevent users from entering text in a text box:

```
onfocus = "this.blur()";
```

Some operating systems and browsers enable you to give focus to elements such as buttons (including radio and checkbox buttons) and hypertext links (encompassing both `a` and `area` elements). Typically, once such an element has focus, you can accomplish the equivalent of a mouse click on the element by pressing the spacebar. This is helpful for accessibility to those who have difficulty using a mouse.

An unfortunate side effect of button focus in Win32 environments is that the focus highlight (a dotted rectangle) remains around the button after a user clicks it and until another object gets focus. You can eliminate this artifact for browsers and objects that implement the `onmouseup` event handler by including the following event handler in your buttons:

```
onmouseup = "this.blur()";
```

IE5.5+ recognizes the often undesirable effect of that dotted rectangle and lets scripts set the `hideFocus` property of an element to `true` to keep that rectangle hidden while giving the element focus. It is a trade-off for the user, however, because there is no visual feedback about which element has focus.

Other elements

For other kinds of elements that support the `focus()` method, you can bring an element into view in lieu of the `scrollIntoView()` method. Link (`a`) and `area` elements in Windows versions of IE

***elementObject*.blur()**

display the dotted rectangle around them after a user brings focus to them. To eliminate that artifact, use the same

```
onmouseup = "this.blur()";
```

event handler (or IE5.5+ `hideFocus` property) just described for form controls.

Example

Listing 26-23a contains an example of using the `focus()` and `blur()` methods to tinker with changing the focus of windows. This example creates a two-window environment; from each window, you can bring the other window to the front. The main window uses the object returned by `window.open()` to assemble the reference to the new window. In the subwindow (whose content is created entirely on-the-fly by JavaScript), `self.opener` is summoned to refer to the original window, whereas `self.blur()` operates on the subwindow itself. Blurring one window and focusing on another window yields the same result of sending the window to the back of the pile.

LISTING 26-23a

The window.focus() and window.blur() Methods

HTML: jsb26-23.html

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Window Focus() and Blur()</title>
      <script type="text/javascript" src="../jsb-global.js"></script>
      <script type="text/javascript" src="jsb26-23.js"></script>
   </head>
   <body>
      <h1>Window focus() and blur() Methods</h1>
      <hr />
      <form>
         <input type="button" name="newOne" value="Show New Window"
                onclick="makeNewWindow()" />
      </form>
   </body>
</html>
```

JavaScript: jsb26-23.js

```
// declare global variable name
var newWindow = null;

function makeNewWindow()
{
   // check if window already exists
   if (!newWindow || newWindow.closed)
   {
      // store new window object in global variable
      newWindow = window.open("","","width=250,height=250");
      // pause briefly to let IE3 window finish opening
```

```
      setTimeout("fillWindow()",100);
   }
   else
   {
      // window already exists, so bring it forward
      newWindow.focus();
   }
}

// assemble new content and write to subwindow
function fillWindow()
{
   var newContent = "<html><head><title>Another Sub Window<\/title><\/head>";
   newContent += "<body bgColor='salmon'>";
   newContent += "<h1>A Salmon-Colored Subwindow.<\/h1>";
   newContent += "<form><input type='button' value='Bring Main to Front'
      onclick='self.opener.focus()'>";
   newContent += "<input type='button' value='Put Me in Back'
      onclick='self.blur()'>";
   newContent += "<\/form><\/body><\/html>";
   // write HTML to new window document
   newWindow.document.write(newContent);
   newWindow.document.close();
}
```

A key ingredient to the success of the `makeNewWindow()` function in Listing 26-23a is the first conditional expression. Because `newWind` is initialized as a `null` value when the page loads, that is its value the first time through the function. But after you open the subwindow the first time, `newWind` is assigned a value (the subwindow object) that remains intact even if the user closes the window. Thus, the value doesn't revert to `null` by itself. To catch the possibility that the user has closed the window, the conditional expression also sees whether the window is closed. If it is, a new subwindow is generated, and that new window's reference value is reassigned to the `newWind` variable. On the other hand, if the window reference exists and the window is not closed, the `focus()` method brings that subwindow to the front.

In Listing 26-23a you can see where the subwindow's content is created entirely on-the-fly by JavaScript. One of the nice little tools in IE and Firefox is the ability to view generated source. It's a little ugly, in that all the code is on a single line. We've formatted the generated source for you in Listing 26-23b so that you can see just what was created entirely on-the-fly in Listing 26.23a.

LISTING 26-23b

The Source Code Generated On-the-Fly in Listing 26-23a

```
<html>
   <head>
      <title>Another Sub Window</title>
   </head>
   <body bgcolor="salmon">
```

continued

LISTING 26-23b *(continued)*

```
      <h1>A Salmon-Colored Subwindow.</h1>
      <form>
         <input value="Bring Main to Front" onclick="self.opener.focus()"
               type="button">
         <input value="Put Me in Back" onclick="self.blur()" type="button">
      </form>
   </body>
</html>
```

You can see the `focus()` method for a text object in action in the description of the `select()` method for text objects in Chapter 36.

Related Items: `window.open()`, `document.formObject.textObject.select()` methods

clearAttributes()

Returns: Nothing

Compatibility: WinIE5+, MacIE-, NN-, Moz-, Safari-, Opera-, Chrome-

The `clearAttributes()` method removes all attributes from an element except the `name` and `id` values. Thus, styles and event handlers are removed, as are custom attributes assigned in either the HTML source code or later by script. You should know that the `clearAttributes()` method does not alter the length of the element's `attributes` collection, because the collection always contains all possible attributes for an element. (See the `attributes` property for elements earlier in this chapter.)

This method is handy if you wish to construct an entirely new set of attributes for an element and prefer to start out with a blank slate. Be aware, however, that unless your scripts immediately assign new attributes to the element, the appearance of the element reverts to its completely unadorned form until you assign new attributes. This means that even positioned elements find their way back to their source-code order until you assign a new positioning style. If you simply want to change the value of one or more attributes of an element, it is faster to use the `setAttribute()` method or adjust the corresponding properties.

To accomplish a result in NN6+/Moz that simulates that of IE5+'s `clearAttributes()`, you must iterate through all attributes of an element and remove those attributes (via the `removeAttribute()` method) whose names are other than `id` and `name`.

Example

Use The Evaluator (see Chapter 4) to examine the attributes of an element before and after you apply `clearAttributes()`. To begin, display the HTML for the table element on the page by entering the following statement in the top text box:

```
myTable.outerHTML
```

Notice the attributes associated with the `<table>` tag. Look at the rendered table to see how attributes such as `border` and `width` affect the display of the table. Now enter the following statement in the top text box to remove all removable attributes from this element:

```
myTable.clearAttributes()
```

First, look at the table. The border is gone, and the table is rendered only as wide as is necessary to display the content with no cell padding. Finally, view the results of the `clearAttributes()` method in the `outerHTML` of the table again:

```
myTable.outerHTML
```

The source-code file has not changed, but the object model in the browser's memory reflects the changes you made.

Related Items: `attributes` property; `getAttribute()`, `setAttribute()`, `removeAttribute()`, `mergeAttributes()`, and `setAttributeNode()` methods

click()

Returns: Nothing

Compatibility: WinIE4+, MacIE4+, NN2+, Moz+, Safari+, Opera+, Chrome+

The `click()` method lets a script perform nearly the same action as clicking an element. Before NN4 and IE4, the `click()` method invoked on a button did not trigger the `onclick` event handler for the object. This has significant impact if you expect the `onclick` event handler of a button to function even if a script performs the click. For earlier browser versions, you have to invoke the event handler statements directly. Also, just because a script is clicking a button, not all buttons in all platforms change their appearance in response. For example, NN4 on the Mac does not change the state of a checkbox clicked remotely.

If you want to script the action of clicking a button, you can safely invoke the resulting event handler function directly. And if the element is a radio button or checkbox, handle the change of state directly (for example, set the `checked` property of a checkbox) rather than expect the browser to take care of it for you.

Example

Use The Evaluator (see Chapter 4) to experiment with the `click()` method. The page includes various types of buttons at the bottom. You can click the checkbox, for example, by entering the following statement in the top text box:

```
document.myForm2.myCheckbox.click()
```

If you use a recent browser version, you most likely can see the checkbox change states between checked and unchecked each time you execute the statement.

Related Item: `onclick` event handler

cloneNode(*deepBoolean*)

Returns: Node object reference

Compatibility: WinIE5+, MacIE5+, NN6+, Moz+, Safari+, Opera+, Chrome+

The `cloneNode()` method makes an exact copy of the current node object. This copy does not have a parent node or other relationship with any element after the copy exists (of course, the original node remains in place). The clone also does not become part of the document's object model (the node tree) unless you explicitly insert or append the node somewhere on the page. The copy includes all element attributes, including the `id` attribute. Because the value returned by the `cloneNode()` method is

a genuine `Node` object, you can operate on it with any `Node` object methods while it is still in the nondocument object state.

The Boolean parameter of the `cloneNode()` method controls whether the copy of the node includes all child nodes (`true`) or just the node itself (`false`). For example, if you clone a paragraph element by itself, the clone consists only of the raw element (equivalent of the tag pair, including attributes in the start tag) and none of its content. But including child nodes makes sure that all content within that paragraph element is part of the copy. This parameter is optional in IE5 (defaulting to `false`), but it is required in other W3C-compatible browsers.

Example

Use The Evaluator (see Chapter 4) to clone, rename, and append an element found in The Evaluator's source code. Begin by cloning the paragraph element named `myP` along with all of its content. Enter the following statement in the top text box:

```
a = document.getElementById("myP").cloneNode(true)
```

The variable `a` now holds the clone of the original node, so you can change its `id` attribute at this point by entering the following statement:

```
a.setAttribute("id", "Dolly")
```

If you want to see the properties of the cloned node, enter `a` in the bottom text box. The precise listing of properties you see depends on the browser you're using; in either case, you should be able to locate the `id` property, whose value is now `Dolly`.

As a final step, append this newly named node to the end of the `body` element by entering the following statement in the top text box:

```
document.body.appendChild(a)
```

You can now scroll down to the bottom of the page and see a duplicate of the content. But because the two nodes have different `id` attributes, they cannot confuse scripts that need to address one or the other.

Related Items: `Node` object (Chapter 25); `appendChild()`, `removeChild()`, `removeNode()`, `replaceChild()`, and `replaceNode()` methods

compareDocumentPosition(nodeRef)

Returns: Integer

Compatibility: WinIE-, MacIE-, NN6+, Moz1.4+, Safari+, Opera+, Chrome+

This method determines the tree position of one node with respect to another node. More specifically, the `nodeRef` object provided as a parameter (Node B) is compared with the object on which the method is called (Node A). The result is returned from the method as an integer value that can contain one or more of the comparison masks listed in Table 26-8.

The integer value returned by the `compareDocumentPosition()` method is actually a bitmask, which explains why the values in Table 26-8 are powers of 2. This allows the method to return multiple comparison values simply by adding them together. For example, a return value of `20` indicates that Node B is contained by Node A (`16`) and also that Node B follows Node A (4).

TABLE 26-8

Comparison Return Flags

Integer Value	Constant	Description
0		Node B and Node A are one and the same.
1	DOCUMENT_POSITION_DISCONNECTED	No connection exists between the nodes.
2	DOCUMENT_POSITION_PRECEDING	Node B precedes Node A.
4	DOCUMENT_POSITION_FOLLOWING	Node B follows Node A.
8	DOCUMENT_POSITION_CONTAINS	Node B contains Node A (and therefore precedes it).
16	DOCUMENT_POSITION_CONTAINED_BY	Node B is contained by Node A (and therefore follows it).
32	DOCUMENT_POSITION_IMPLEMENTATION_ SPECIFIC	The comparison is determined by the browser.

Related Item: `contains()` method

componentFromPoint(*x*,*y*)

Returns: String

Compatibility: WinIE5+, MacIE-, NN-, Moz-, Safari-, Opera-, Chrome-

The `componentFromPoint()` method assists in some event-related tasks. You can use it for a kind of collision detection (in other words, to determine whether an event occurs inside or outside a particular element). If the element has scroll bars, the method can provide additional information about the event, such as which component of the scroll bar the user activates.

A key aspect of this method is that you invoke it on any element that you want to use as the point of reference. For example, if you want to find out whether a `mouseup` event occurs in an element whose ID is `myTable`, invoke the method as follows:

```
var result = document.getElementById("myTable").componentFromPoint(↩
   event.clientX, event.clientY);
```

Parameters passed to the method are *x* and *y* coordinates. These coordinates do not have to come from an event, but the most likely scenario links this method with an event of some kind. Mouse events (other than `onclick`) work best.

The value returned by the method is a string that provides details about where the coordinate point is with respect to the current element. If the coordinate point is inside the element's rectangle, the returned value is an empty string. Conversely, if the point is completely outside the element, the returned value is the string `"outside"`. For scroll-bar pieces, the list of possible returned values is quite lengthy (as shown in Table 26-9).

TABLE 26-9

Returned Values for componentFromPoint()

Returned String	Element Component at Coordinate Point
(empty)	Inside the element content area
`outside`	Outside the element content area
`handleBottom`	Resize handle at bottom
`handleBottomLeft`	Resize handle at bottom left
`handleBottomRight`	Resize handle at bottom right
`handleLeft`	Resize handle at left
`handleRight`	Resize handle at right
`handleTop`	Resize handle at top
`handleTopLeft`	Resize handle at top left
`handleTopRight`	Resize handle at top right
`scrollbarDown`	Scroll-bar down arrow
`scrollbarHThumb`	Scroll-bar thumb on horizontal bar
`scrollbarLeft`	Scroll-bar left arrow
`scrollbarPageDown`	Scroll-bar page-down region
`scrollbarPageLeft`	Scroll-bar page-left region
`scrollbarPageRight`	Scroll-bar page-right region
`scrollbarPageUp`	Scroll-bar page-up region
`scrollbarRight`	Scroll-bar right arrow
`scrollbarUp`	Scroll-bar up arrow
`scrollbarVThumb`	Scroll-bar thumb on vertical bar

You do not have to use this method for most collision or event detection, however. The `event` object's `srcElement` property returns a reference to whatever object receives the event.

Example

Listing 26-24 demonstrates how the `componentFromPoint()` method can be used to determine exactly where a mouse event occurred. As presented, the method is associated with a `textarea` object that is specifically sized to display both vertical and horizontal scroll bars. As you click various areas of the `textarea` and the rest of the page, the status bar displays information about the location of the event with the help of the `componentFromPoint()` method.

The script uses a combination of the `event.srcElement` property and the `componentFromPoint()` method to help you distinguish how you can use each one for different types of event processing. The `srcElement` property is used initially as a filter to decide whether the status bar will reveal further processing about the `textarea` element's event details.

The `onmousedown` event handler in the `body` element triggers all event processing. IE events bubble up the hierarchy (and no events are canceled in this page), so all `mousedown` events eventually reach the `body` element. Then the `whereInWorld()` function can compare each `mousedown` event from any element against the text area's geography.

LISTING 26-24

Using the componentFromPoint() Method

HTML: jsb26-24.html

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>componentFromPoint() Method</title>
      <script type="text/javascript" src="../jsb-global.js"></script>
      <script type="text/javascript" src="jsb26-24.js"></script>
   </head>
   <body onmousedown="whereInWorld()">
      <h1>componentFromPoint() Method</h1>
      <hr />
      <p>Tracking the mouseDown event relative to the textarea object. View
         results in status bar.
      </p>
      <form>
         <textarea id="myTextarea" wrap="off" cols="12" rows="4">
            This is Line 1
            This is Line 2
            This is Line 3
            This is Line 4
            This is Line 5
            This is Line 6
         </textarea>
      </form>
   </body>
</html>
```

JavaScript: jsb26-24.js

```
function whereInWorld(elem)
{
  var x = event.clientX;
  var y = event.clientY;
  var component = document.getElementById("myTextarea").componentFromPoint(x,y);
  if (window.event.srcElement == document.getElementById("myTextarea"))
  {
```

continued

LISTING 26-24 *(continued)*

```
      if (component == "")
      {
         status = "mouseDown event occurred inside the element";
      }
      else
      {
         status = "mouseDown occurred on the element\'s " + component;
      }
   }
   else
   {
      status = "mouseDown occurred " + component + " of the element";
   }
}
```

Related Item: `event` object

contains(*elementObjectReference*)

Returns: Boolean

Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari+-, Opera+, Chrome+

The `contains()` method reports whether the current object contains another known object within its HTML containment hierarchy. Note that this is not geographical collision detection of overlapping elements, but the determination of whether one element is nested somewhere within another.

The scope of the `contains()` method extends as deeply within the current object's hierarchy as is necessary to locate the object. In essence, the `contains()` method examines all of the elements that are part of an element's `all` array. Therefore, you can use this method as a shortcut replacement for a `for` loop that examines each nested element of a container for the existence of a specific element.

The parameter to the `contains()` method is a reference to an object. If you have only the element's ID as a string to go by, you can use the `document.getElementById()` method to generate a valid reference to the nested element.

Note

An element always contains itself. ■

Example

Using The Evaluator (Chapter 4), see how the `contains()` method responds to the object combinations in each of the following statements as you enter them in the top text box:

```
document.body.contains(document.getElementById("myP"))
document.getElementById("myP").contains(document.getElementById("myEM"))
document.getElementById("myEM").contains(document.getElementById("myEM"))
document.getElementById("myEM").contains(document.getElementById("myP"))
```

Feel free to test other object combinations within this page.

Related Items: `item()`, `document.getElementById()` methods

`createControlRange("*param*")`

Returns: Integer ID

Compatibility: WinIE5+, MacIE-, NN-, Moz-, Safari-, Opera-, Chrome-

The `createControlRange()` method is used to create a control range for a selection of text. Although the method is implemented for several elements, it is intended solely for the `selection` object and, therefore, should be used only on that object.

Related Item: `selection` object

`detachEvent()`

(See `attachEvent()`)

`dispatchEvent(*eventObject*)`

Returns: Boolean

Compatibility: WinIE-, MacIE-, NN6+, Moz+, Safari+, Opera+, Chrome+

The `dispatchEvent()` method allows a script to fire an event aimed at any object capable of supporting that event. This is the W3C event model way of generalizing mechanisms that earlier browsers sometimes mimic with object methods such as `click()` and `focus()`.

The process of generating one of these events is similar to the way a script generates a new node and inserts that node somewhere into the DOM. For events, however, the object that is created is an `Event` object, which is generated via the `document.createEvent()` method. An event generated in this manner is simply a specification about an event. Use properties of an event object to supply specifics about the event, such as its coordinates or mouse button. Then dispatch the event to a target object by invoking that target object's `dispatchEvent()` method and passing the newly created `Event` object as the sole parameter.

Interpreting the meaning of the Boolean value that the `dispatchEvent()` method returns is not straightforward. The browser follows the dispatched event through whatever event propagation is in effect for that object and event type (either bubbling or capture). If any of the event listener functions triggered by this dispatched event invokes the `preventDefault()` method, the `dispatchEvent()` method returns `false` to indicate that the event did not trigger the native action of the object; otherwise, the method returns `true`. Notice that this returned value indicates nothing about propagation type or how many event listeners run as a result of dispatching this event.

Caution

Although the `dispatchEvent()` method was implemented in NN6, the browser does not yet provide a way to generate new events from scratch. And if you attempt to redirect an existing event to another object via the `dispatchEvent()` method, the browser is prone to crashing. In other words, Mozilla-based browsers are much better candidates for scripts that use `dispatchEvent()`. ■

Example

Listing 26-25 demonstrates how to fire events programmatically using the W3C DOM `dispatchEvent()` method. Notice the syntax in the `doDispatch()` function for creating and initializing a new mouse event, supported most reliably in Mozilla-based browsers. The behavior is identical to that of Listing 26-26 later in this chapter, which demonstrates the IE5.5+ equivalent: `fireEvent()`.

LISTING 26-25

Using the dispatchEvent() Method

HTML: jsb26-25.html

```
<!DOCTYPE html>
<html>
  <head>
    <meta http-equiv="content-type" content="text/html;charset=utf-8">
    <title>Using the dispatchEvent() Method</title>
    <style type="text/css">
       #mySPAN
       {
         font-style:italic;
       }
       #ctrlPanel
       {
          font-weight:bold;
       }
    </style>
    <script type="text/javascript" src="../jsb-global.js"></script>
    <script type="text/javascript" src="jsb26-25.js"></script>
  </head>
  <body id="myBODY" onload="init()">
   <h1>dispatchEvent() Method</h1>
   <hr />
   <p id="myP">This is a paragraph <span id="mySPAN">(with a nested SPAN)</span>
            that receives click events.
   </p>
   <hr />
   <p>
      <span id="ctrlPanel">Control Panel</span>
   </p>
   <form name="controls">
      <p>
         <input type="checkbox" name="bubbleOn"
                onclick="event.stopPropagation()" />Cancel event bubbling.
      </p>
      <p>
         <input type="button" value="Fire Click Event on BODY"
                onclick="doDispatch('myBODY', event)" />
      </p>
      <p>
```

```
         <input type="button" value="Fire Click Event on myP"
                onclick="doDispatch('myP', event)" />
      </p>
      <p>
         <input type="button" value="Fire Click Event on mySPAN"
                onclick="doDispatch('mySPAN', event)" />
      </p>
    </form>
  </body>
</html>
```

JavaScript: jsb26-25.js

```
// assemble a couple event object properties
function getEventProps(evt)
{
   var msg = "";
   var elem = evt.target;
   msg += "event.target.nodeName: " + elem.nodeName + "\n";
   msg += "event.target.parentNode: " + elem.parentNode.id + "\n";
   msg += "event button: " + evt.button;
   return msg;
}

// onClick event handlers for body, myP, and mySPAN
function bodyClick(evt)
{
   var msg = "Click event processed in BODY\n\n";
   msg += getEventProps(evt);
   alert(msg);
   checkCancelBubble(evt);
}
function pClick(evt)
{
   var msg = "Click event processed in P\n\n";
   msg += getEventProps(evt);
   alert(msg);
   checkCancelBubble(evt);
}
function spanClick(evt)
{
   var msg = "Click event processed in SPAN\n\n";
   msg += getEventProps(evt);
   alert(msg);
   checkCancelBubble(evt);
}

// cancel event bubbling if checkbox is checked
function checkCancelBubble(evt)
{
   if (document.controls.bubbleOn.checked)
   {
      evt.stopPropagation();
```

continued

LISTING 26-25 *(continued)*

```
   }
}

// assign onClick event handlers to three elements
function init()
{
   document.body.onclick = bodyClick;
   document.getElementById("myP").onclick = pClick;
   document.getElementById("mySPAN").onclick = spanClick;
}

// invoke dispatchEvent() on object whose ID is passed as parameter
function doDispatch(objID, evt)
{
   // create empty mouse event
   var newEvt = document.createEvent("MouseEvents");
   // initialize as click with button ID 3
   newEvt.initMouseEvent("click", true, true, window, 0, 0, 0, 0, 0, false,
        false, false, false, 3, null);
   // send event to element passed as param
   document.getElementById(objID).dispatchEvent(newEvt);
   // don't let button clicks bubble
   evt.stopPropagation();
}
```

Related Item: `fireEvent()` method

doScroll("*scrollAction*")

Returns: Nothing

Compatibility: WinIE5+, MacIE-, NN-, Moz-, Safari-, Opera-, Chrome-

The `doScroll()` method is used to control the scrolling of an element by triggering its scroll bars. Although a subtle distinction, `doScroll()` doesn't move the scroll bars to a specific position; instead, it simulates a scroll-bar click. The end result is an `onscroll` event being fired, which is what you would expect from a simulated scroll.

The string parameter to `doScroll()` can be one of the following values to indicate what kind of scrolling is to take place: `scrollbarUp`, `scrollbarDown`, `scrollbarLeft`, `scrollbarRight`, `scrollbarPageUp`, `scrollbarPageDown`, `scrollbarPageLeft`, `scrollbarPageRight`, `scrollbarHThumb`, or `scrollbarVThumb`.

dragDrop()

Returns: Boolean

Compatibility: WinIE5.5+, MacIE-, NN-, Moz-, Safari-, Opera-, Chrome-

The `dragDrop()` method initiates a mouse drag-and-drop sequence by triggering an `ondragstart` event. The return value is a Boolean that indicates when the user releases the mouse button (`true`).

fireEvent("*eventType*"[, *eventObjectRef*])

Returns: Boolean

Compatibility: WinIE5.5+, MacIE-, NN-, Moz-, Safari-, Opera-, Chrome-

Although some objects have methods that emulate physical events (for example, the `click()` and `focus()` methods), WinIE5.5+ generalizes the mechanism by letting a script direct any valid event to any object. The `fireEvent()` method is the vehicle.

One required parameter is the event type, formatted as a string. IE event types are coded just like the property names for event handlers (for example, `onclick`, `onmouseover`, and so on).

A second, optional parameter is a reference to an existing `event` object. This object can be an event that some user or system action triggers (meaning that the `fireEvent()` method is in a function invoked by an event handler). The existing event can also be an object created by the IE5.5+ `document.createEventObject()` method. In either case, the purpose of providing an existing `event` object is to set the properties of the `event` object that the `fireEvent()` method creates. The event type is defined by the method's first parameter, but if you have other properties to set (for example, coordinates or a keyboard key code), those properties are picked up from the existing object. Here is an example of a sequence that creates a new `mousedown` event, stuffs some values into its properties, and then fires the event at an element on the page:

```
var newEvent = document.createEventObject();
newEvent.clientX = 100;
newEvent.clientY = 30;
newEvent.cancelBubble = false;
newEvent.button = 1;
document.getElementById("myElement").fireEvent("onmousedown", newEvent);
```

Events generated by the `fireEvent()` method are just like regular IE `window.event` objects, and they have several important `event` object properties that the browser presets. It is important that `cancelBubble` is set to `false` and `returnValue` is set to `true` — just like a regular user- or system-induced event. This means that if you want to prevent event bubbling and/or prevent the default action of the event's source element, the event handler functions must set these `event` object properties just like normal event handling in IE.

The `fireEvent()` method returns a Boolean value that the `returnValue` property of the event determines. If the `returnValue` property is set to `false` during event handling, the `fireEvent()` method returns `false`. Under normal processing, the method returns `true`.

The W3C DOM (Level 2) event model includes the `dispatchEvent()` method to accommodate script-generated events (and `Event` object methods to create event objects), which is roughly the W3C equivalent of the `fireEvent()` method.

Example

Listing 26-26 contains script code that shows how to fire events programmatically using the `fireEvent()` method. Three buttons in the example page enable you to direct a click event to each of the three elements that have event handlers defined for them. The events fired this way are artificial, generated via the `createEventObject()` method. For demonstration purposes, the `button` property of these scripted events is set to 3. This property value is assigned to the `event`

object that eventually gets directed to an element. With event bubbling left on, the events sent via `fireEvent()` behave just like the physical clicks on the elements. Similarly, if you disable event bubbling, the first event handler to process the event cancels bubbling, and no further processing of that event occurs. Notice that event bubbling is canceled within the event handlers that process the event. To prevent the clicks of the checkbox and action buttons from triggering the `body` element's `onclick` event handlers, event bubbling is turned off for the buttons right away.

LISTING 26-26

Using the fireEvent() Method

HTML: jsb26-26.html

```
<!DOCTYPE html>
<html>
  <head>
    <meta http-equiv="content-type" content="text/html;charset=utf-8">
    <title>Using the fireEvent() Method</title>
    <style type="text/css">
       #mySPAN
       {
          font-style:italic;
       }
       #ctrlPanel
       {
          font-weight:bold;
       }
    </style>
    <script type="text/javascript" src="../jsb-global.js"></script>
    <script type="text/javascript" src="jsb26-26.js"></script>
  </head>
  <body id="myBODY" onload="init()">
   <h1>fireEvent() Method</h1>
   <hr />
   <p id="myP">This is a paragraph <span id="mySPAN">(with a nested SPAN)</span>
            that receives click events.
   </p>
   <hr />
   <p>
      <span id="ctrlPanel">Control Panel</span>
   </p>
   <form name="controls">
      <p>
         <input type="checkbox" name="bubbleOn"
                onclick="event.cancelBubble=true" />Cancel event bubbling.
      </p>
      <p>
         <input type="button" value="Fire Click Event on BODY"
                onclick="doFire('myBODY')" />
      </p>
      <p>
         <input type="button" value="Fire Click Event on myP"
```

```
               onclick="doFire('myP')" />
      </p>
      <p>
         <input type="button" value="Fire Click Event on mySPAN"
               onclick="doFire('mySPAN')" />
      </p>
   </form>
  </body>
</html>
```

JavaScript: jsb26-26.js

```
// assemble a couple event object properties
function getEventProps()
{
   var msg = "";
   var elem = event.srcElement;
   msg += "event.srcElement.tagName: " + elem.tagName + "\n";
   msg += "event.srcElement.id: " + elem.id + "\n";
   msg += "event button: " + event.button;
   return msg;
}

// onClick event handlers for body, myP, and mySPAN
function bodyClick()
{
   var msg = "Click event processed in BODY\n\n";
   msg += getEventProps();
   alert(msg);
   checkCancelBubble();
}
function pClick()
{
   var msg = "Click event processed in P\n\n";
   msg += getEventProps();
   alert(msg);
   checkCancelBubble();
}
function spanClick()
{
   var msg = "Click event processed in SPAN\n\n";
   msg += getEventProps();
   alert(msg);
   checkCancelBubble();
}

// cancel event bubbling if checkbox is checked
function checkCancelBubble()
{
   event.cancelBubble = document.controls.bubbleOn.checked;
}

// assign onClick event handlers to three elements
```

continued

LISTING 26-26 *(continued)*

```
function init()
{
   document.getElementById("myBODY").onclick = bodyClick;
   document.getElementById("myP").onclick = pClick;
   document.getElementById("mySPAN").onclick = spanClick;
}

// invoke fireEvent() on object whose ID is passed as parameter
function doFire(objID)
{
   var newEvt = document.createEventObject();
   newEvt.button = 3;
   document.all(objID).fireEvent("onclick", newEvt);
   // don't let button clicks bubble
   event.cancelBubble = true;
}
```

Related Item: `dispatchEvent()` method

`focus()`

(See `blur()`)

`getAdjacentText("`*`position`*`")`

Returns: String

Compatibility: WinIE5+, MacIE-, NN-, Moz-, Safari-, Opera-, Chrome-

The `getAdjacentText()` method enables you to extract copies of plain-text components of an element object (in other words, without any HTML tag information). The sole parameter is one of four case-insensitive string constant values that indicate from where, in relation to the current object, the text should be extracted. The values are:

Parameter Value	Description
`beforeBegin`	Text immediately in front of the element's tag, back to the preceding tag
`afterBegin`	Text that begins inside the element tag, up to the next tag (whether it be a nested element or the element's end tag)
`beforeEnd`	Text immediately in front of the element's end tag, back to the preceding tag (whether it be a nested element or the element's start tag)
`afterEnd`	Text immediately following the element's end tag, forward until the next tag

If the current object has no nested elements, both the `afterBegin` and `beforeEnd` versions return the same as the object's `innerText` property. When the current object is encased immediately within another element (for example, a `td` element inside a `tr` element), there is no text before the element's beginning or after the element's end, so these values are returned as empty strings.

The strings returned from this method are roughly equivalent to values of text fragment nodes in the W3C DOM, but IE5+ treats these data pieces as string data types rather than as text node types. W3C DOM equivalents for the four versions are

```
document.getElementById("objName").previousSibling.nodeValue
document.getElementById("objName").firstChild.nodeValue
document.getElementById("objName").lastChild.nodeValue
document.getElementById("objName").nextSibling.nodeValue
```

Example

Use The Evaluator (see Chapter 4) to examine all four adjacent text possibilities for the `myP` and nested `myEM` elements in that document. Enter each of the following statements in the top text box, and view the results:

```
document.getElementById("myP").getAdjacentText("beforeBegin")
document.getElementById("myP").getAdjacentText("afterBegin")
document.getElementById("myP").getAdjacentText("beforeEnd")
document.getElementById("myP").getAdjacentText("afterEnd")
```

The first and last statements return empty strings because the `myP` element has no text fragments surrounding it. The `afterBegin` version returns the text fragment of the `myP` element up to, but not including, the EM element nested inside. The `beforeEnd` string picks up after the end of the nested EM element and returns all text to the end of `myP`.

Now see what happens with the nested `myEM` element:

```
document.getElementById("myEM").getAdjacentText("beforeBegin")
document.getElementById("myEM").getAdjacentText("afterBegin")
document.getElementById("myEM").getAdjacentText("beforeEnd")
document.getElementById("myEM").getAdjacentText("afterEnd")
```

Because this element has no nested elements, the `afterBegin` and `beforeEnd` strings are identical — the same value as the `innerText` property of the element.

Related Items: `childNodes`, `data`, `firstChild`, `lastChild`, `nextSibling`, `nodeValue`, and `previousSibling` properties

getAttribute("*attributeName*"[, *caseSensitivity*])

Returns: (See text)

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

The `getAttribute()` method returns the value assigned to a specific attribute of the current object. You can use this method as an alternative to retrieving properties of an object, particularly when your

script presents you the attribute name as a string (in contrast to a fully formed reference to an object and its property). Thus, the following example statements yield the same data:

```
var mult = document.getElementById("mySelect").multiple;
var mult = document.getElementById("mySelect").getAttribute("multiple");
```

Returned value types from `getAttribute()` are either strings (including attribute values assigned as unquoted numeric values) or Booleans (for example, the `multiple` property of a `select` element object).

Note

The W3C DOM Level 2 standard recommends `getAttribute()` and `setAttribute()` for reading and writing element object attribute values, rather than reading and writing those values by way of their corresponding properties. Although using these methods is certainly advisable for XML elements, the same DOM standard sends conflicting signals by defining all kinds of properties for HTML element objects. Browsers, of course, will support access via properties well into the future, so don't feel obligated to change your ways just yet. ■

All browsers that support the `getAttribute()` method require one parameter, which is a string of the attribute name. By default, this parameter is not case sensitive. Note that this has impact on custom attributes that you might assign to HTML or XML elements in your documents. Attribute names are automatically converted to lowercase when they are turned into properties of the object. Therefore, you must avoid reusing attribute names, even if you use different case letters in the source-code assignments.

IE includes an optional extension to the method in the form of a second parameter that enables you to be more specific about the case sensitivity of the first parameter. The default value of the second parameter is `false`, which means that the first parameter is not case sensitive. A value of `true` makes the first parameter case sensitive. This matters only if you use `setAttribute()` to add a parameter to an existing object and if the IE version of that method insists on case sensitivity. The default behavior of `setAttribute()` respects the case of the attribute name. See also the discussion of the `setAttribute()` method later in this chapter with regard to `setAttribute()`'s influence on the IE `attributes` property.

Example

Use The Evaluator (see Chapter 4) to experiment with the `getAttribute()` method for the elements in the page. You can enter the following sample statements in the top text box to view attribute values:

```
document.getElementById("myTable").getAttribute("cellpadding")
document.getElementById("myTable").getAttribute("border")
```

Related Items: `attributes` property; `document.createAttribute()`, `setAttribute()` methods

getAttributeNode("*attributeName*")

Returns: Attribute node object

Compatibility: WinIE6+, MacIE-, NN6+, Moz+, Safari+, Opera+, Chrome+

In the W3C DOM, an attribute is an object that inherits all the properties of a `Node` object (see Chapter 25). As its name implies, an attribute object represents a name-value pair of an attribute that

is explicitly defined inside an element's tag. The ability to treat attributes as node objects is far more important when working with XML than HTML, but it is helpful to understand attribute nodes within the context of the W3C DOM object-oriented view of a document. It is important that attribute nodes specifically are not recognized as nodes of a document hierarchy. Therefore, an attribute node is not a child node of the element that defines the attribute.

The nodeness of attributes comes into play when addressing the contents of an object's `attributes` property. The W3C `attributes` property builds on the DOM's formal structure by returning an object known (internally) as a *named node map*. Like an array, the named node map has a `length` property (facilitating `for` loop iteration through the map), plus several methods that allow for inserting, removing, reading, or writing attribute name-value pairs within the node map.

An attribute object inherits all the properties of the `Node` object. Table 26-10 lists the properties of an attribute object.

All of this is a long way to explain the W3C DOM `getAttributeNode()` method, which returns a W3C DOM attribute object. The sole parameter of the method is a case-insensitive string version of the attribute's name. Then you can use any of the properties shown in Table 26-10 to get or set attribute values. Of course, HTML attributes are generally exposed as properties of HTML elements, so it is usually easier to read or write the object's properties directly.

TABLE 26-10

Attribute Object Properties of W3C DOM–Compatible Browsers

`attributes`
`childNodes`
`firstChild`
`lastChild`
`name`
`nextSibling`
`nodeName`
`nodeType`
`nodeValue`
`ownerDocument`
`parentNode`
`previousSibling`
`specified`
`value`

Example

Use The Evaluator (see Chapter 4) to explore the `getAttributeNode()` method. The Results `textarea` element provides several attributes to check out. Because the method returns an object, enter the following statements in the bottom text box so you can view the properties of the attribute node object returned by the method:

```
document.getElementById("myTextArea").getAttributeNode("cols")
document.getElementById("myTextArea").getAttributeNode("rows")
document.getElementById("myTextArea").getAttributeNode("wrap")
document.getElementById("myTextArea").getAttributeNode("style")
```

All (except the last) statements display a list of properties for each attribute node object. The last statement, however, returns nothing because the `style` attribute is not specified for the element.

You can also use The Evaluator to explore the properties of the attribute object returned by the `getAttributeNode()` method. Enter the following statement in the top text box:

```
document.getElementById("myTextArea").getAttributeNode("cols").value
```

Related Items: `attributes` property; `getAttribute()`, `removeAttributeNode()`, `setAttributeNode()` methods

getAttributeNodeNS("*namespaceURI*", "*localName*")

Returns: Attribute node object

Compatibility: WinIE-, MacIE-, NN6+, Moz+, Safari+-, Opera+, Chrome_

This method returns a W3C DOM attribute object. The first parameter of the method is a URI string matching a URI assigned to a label in the document. The second parameter is the local name portion of the attribute you are getting.

Related Items: `attributes`, `namespaceURI`, `localName` properties; `getAttributeNode()`, `setAttributeNodeNS()` methods

getAttributeNS("*namespaceURI*", "*localName*")

Returns: (See text)

Compatibility: WinIE-, MacIE-, NN6+, Moz+, Safari+-, Opera+, Chrome+

This method returns the value assigned to a specific attribute of the current object when that attribute's name is defined by way of an XML namespace definition within the document. The first parameter of the method is a URI string matching a URI assigned to a namespace label in a tag defined earlier in the document. The second parameter is the local name portion of the attribute whose value you are getting.

Returned value types from `getAttributeNS()` are either strings (including attribute values assigned as unquoted numeric values) or Booleans (for example, the `multiple` property of a `select` element object). In the W3C DOM, Netscape, Safari, and Opera, return values are always strings.

Related Items: `attributes`, `namespaceURI`, `localName` properties; `getAttribute()`, `getAttributeNodeNS()`, `setAttributeNodeNS()` methods

getBoundingClientRect()
getClientRects()

Returns: `getBoundingClientRect()` returns a `TextRectangle` object; `getClientRects()` returns an array of `TextRectangle` objects

Compatibility: WinIE5+, MacIE-, NN-, Moz+-, Safari+-, Opera+, Chrome+

Original with IE5+, most modern browsers assign to every content-holding element a rectangle that describes the space that the element occupies on the page. This rectangle is called a *bounding rectangle,* and it is expressed as a TextRectangle object (even when the content is an image or some other kind of object). A `TextRectangle` object has four properties (`top`, `left`, `bottom`, and `right`) that are the pixel coordinates that define the rectangle. You can get the same information from older browsers that do not support the `TextRectangle` object with `offsetTop`, `offsetLeft`, `offsetHeight`, and `offsetWidth`. The `getBoundingClientRect()` method returns a `TextRectangle` object that describes the bounding rectangle of the current object. You can access an individual measure of an object's bounding rectangle, as in the following example:

```
var parTop = document.getElementById("myP").getBoundingClientRect().top;
```

For elements that consist of text, such as paragraphs, the dimensions of individual `TextRectangles` for each line of text in the element influence the dimensions of the bounding rectangle. For example, if a paragraph contains two lines, and the second line extends only halfway across the width of the first line, the width of the second line's `TextRectangle` object is only as wide as the actual text in the second line. But because the first line extends close to the right margin, the width of the encompassing bounding rectangle is governed by that wider, first line `TextRectangle`. Therefore, an element's bounding rectangle is as wide as its widest line and as tall as the sum of the height of all `TextRectangle` objects in the paragraph.

The method `getClientRects()` enables you to obtain a collection of line-by-line `TextRectangle` objects for an element. It returns an array of all `TextRectangle` objects that fall within the current object the moment the method is invoked. Each `TextRectangle` object has its own `top`, `left`, `bottom`, and `right` coordinate properties. You can then, for example, loop through all objects in this array to calculate the pixel width of each line. If you want to find out the aggregate height and/or maximum width of the entire collection, you can use the `getBoundingClientRect()` method as a shortcut.

The length of the collection returned by `getClientRects()` is where you'll see differences between the browsers. WebKit-based browsers and earlier releases of Mozilla-based browsers have only one `TextRectangle` object in the collection returned by `getClientRects()` for each block element found in the argument passed to it. IE and the most recent releases of Mozilla-based browsers will do the same only if a width is specified for a block element. Presto-based browsers always return a collection of one regardless of how many elements, block or not, are involved. All that we've said so far just pertains to an argument that is not identified by a block element (for example, the argument passed to `getClientRects()` in Listing 26-27a is a `span` element). If the argument is a block element (for example a `div`), then the only browser that returns a collection of `TextRectangle` objects for each line is IE. Obviously, this can be very frustrating for scripters. Keep this in mind when looking at Listings 26-27a and 26-27b in the various browsers.

Example

Listing 26-27a employs only the `getClientRects()` method. (We'll look at both methods in Listing 26-27b in a moment.) The element passed to the `getClientRects()` method is a `span` and it contains two paragraphs and an ordered list. The area contained by it is colored blue. There is a button that generates an alert about the length of the collection that is returned by the method.

LISTING 26-27a

Using getClientRects()

HTML: jsb26-27a.html

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>getClientRects() Method</title>
      <style type="text/css">
         #main
         {
            color:blue;
         }
      </style>
      <script type="text/javascript" src="../jsb-global.js"></script>
      <script type="text/javascript" src="jsb26-27a.js"></script>
   </head>
   <body>
      <h1>getClientRects() Method</h1>
      <hr />
      <p>The area we're interested in is colored blue.</p>
      <form>
         <button onclick="howMany();">How many TextRectangle objects
                                   are in the collection?</button>
      </form>
      <span id="main">
         <p>Lorem ipsum dolor sit amet, consectetaur adipisicing
            elit, sed do eiusmod tempor incididunt ut labore et dolore magna
            aliqua. Ut enim adminim veniam, quis nostrud exercitation
            ullamco:
         </p>
         <ul>
            <li>laboris</li>
            <li>nisi</li>
            <li>aliquip ex ea commodo</li>
         </ul>
         <p>Duis aute irure dolor in reprehenderit involuptate velit esse cillum
            dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat
            non proident, sunt in culpa qui officia deseruntmollit anim id est
            laborum. Et harumd und lookum like Greek to me, dereud facilis est
            er expedit distinct.
         </p>
      </span>
   </body>
</html>
```

JavaScript: jsb26-27a.js

```
function howMany()
{
   var clientRects = document.getElementById("main").getClientRects();
   alert("The length of the rectangle returned by getClientRects() is ="
         + clientRects.length + ".");
}
```

Listing 26-27b employs both the `getBoundingClientRect()` and `getClientRects()` methods in a demonstration of how they differ. While the listing works in all major browsers, due to the differences you've already since in the collection returned by the `getClientRects()` method, you'll only really see anything meaningful in IE, and you'll only really want to use it in IE anyway. Keep in mind, though, that you don't have to use the two methods together; there are many things you can do regardless of the browser with just the `getBoundingClientRect()` method.

As with Listing 26-27a, a set of elements is grouped within a `span` element named `main`. The group consists of two paragraphs and an unordered list, all of whose text is colored blue.

Two controls enable you to set the position of an underlying highlight rectangle to any line of your choice. A checkbox enables you to set whether the highlight rectangle should be only as wide as the line or the full width of the bounding rectangle for the entire `span` element.

All the code is located in the `hilite()` function. The `select` and checkbox elements invoke this function. Early in the function, the `getClientRects()` method is invoked for the `main` element to capture a snapshot of all `TextRectangle`s for the entire element. This array comes in handy when the script needs to get the coordinates of a rectangle for a single line, as chosen in the `select` element.

Whenever the user chooses a number from the `select` list, and the value is less than the total number of `TextRectangle` objects in `clientRects`, the function begins calculating the size and location of the underlying yellow highlighter. When the Full Width checkbox is checked, the left and right coordinates are obtained from the `getBoundingClientRect()` method because the entire `span` element's rectangle is the space you're interested in; otherwise, you pull the `left` and `right` properties from the chosen rectangle in the `clientRects` array.

Next comes the assignment of location and dimension values to the `hiliter` object's `style` property. The top and bottom are always pegged to whatever line is selected, so the `clientRects` array is polled for the chosen entry's `top` and `bottom` properties. The previously calculated `left` value is assigned to the `hiliter` object's `pixelLeft` property, whereas the width is calculated by subtracting the `left` from the `right` coordinates. Notice that the `top` and `left` coordinates also take into account any vertical or horizontal scrolling of the entire body of the document. If you resize the window smaller, line wrapping throws off the original line count. However, an invocation of `hilite()` from the `onresize` event handler applies the currently chosen line number to whatever content falls in that line after resizing.

Because the `z-index` style property of the `hiliter` element is set to `-1`, the element always appears beneath the primary content on the page. If the user selects a line number beyond the current number of lines in the main element, the `hiliter` element is hidden.

LISTING 26-27b

Using getBoundingClientRect()

HTML: jsb26-27b.html

```
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>getClientRects() and getBoundClientRect() Methods</title>
      <style type="text/css">
         #main
         {
            color:blue;
         }
      </style>
      <script type="text/javascript" src="../jsb-global.js"></script>
      <script type="text/javascript" src="jsb26-27b.js"></script>
   </head>
   <body id="myBody" onresize="hilite()">
      <h1>getClientRects() and getBoundClientRect() Methods</h1>
      <hr />
      <form>
         Choose a line to highlight:
         <select id="choice" onchange="hilite()">
            <option value="1">1</option>
            <option value="2">2</option>
            <option value="3">3</option>
            <option value="4">4</option>
            <option value="5">5</option>
            <option value="6">6</option>
            <option value="7">7</option>
            <option value="8">8</option>
            <option value="9">9</option>
            <option value="10">10</option>
            <option value="11">11</option>
            <option value="12">12</option>
            <option value="13">13</option>
            <option value="14">14</option>
            <option value="15">15</option>
         </select>
         <br />
         <input name="fullWidth" type="checkbox" onclick="hilite()" />
         Full Width (bounding rectangle)
      </form>
      <span id="main">
         <p>Lorem ipsum dolor sit amet, consectetaur adipisicing
            elit, sed do eiusmod tempor incididunt ut labore et dolore magna
            aliqua. Ut enim adminim veniam, quis nostrud exercitation
            ullamco:
         </p>
```

```
        <ul>
           <li>laboris</li>
           <li>nisi</li>
           <li>aliquip ex ea commodo</li>
        </ul>
        <p>Duis aute irure dolor in reprehenderit involuptate velit esse cillum
           dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat
           non proident, sunt in culpa qui officia deseruntmollit anim id est
           laborum Et harumd und lookum like Greek to me, dereud facilis est
           er expedit distinct.
        </p>
     </span>
     <div id="hiliter"
     style="position:absolute; background-color:yellow;↩
            z-index:-1; visibility:hidden">
     </div>
  </body>
</html>
```

JavaScript: jsb26-27b.js

```
function hilite()
{
   var hTop, hLeft, hRight, hBottom, hWidth;
   var selectList = document.getElementById("choice");
   var n = parseInt(selectList.options[selectList.selectedIndex].value) - 1;
   var clientRects = document.getElementById("main").getClientRects();
   var mainElem = document.getElementById("main");

      if (n >= 0 && n < clientRects.length)
      {
         if (document.forms[0].fullWidth.checked)
         {
            hLeft = mainElem.getBoundingClientRect().left;
            hRight = mainElem.getBoundingClientRect().right;
         }
         else
         {
            hLeft = clientRects[n].left;
            hRight = clientRects[n].right;
         }
         document.getElementById("hiliter").style.pixelTop =
              clientRects[n].top + document.getElementById("myBody").scrollTop;
         document.getElementById("hiliter").style.pixelBottom
              = clientRects[n].bottom;
         document.getElementById("hiliter").style.pixelLeft = hLeft
              + document.getElementById("myBody").scrollLeft;
         document.getElementById("hiliter").style.pixelWidth = hRight - hLeft;
         document.getElementById("hiliter").style.visibility = "visible";
      }
      else if (n > 0)
      {
```

continued

LISTING 26-27b *(continued)*

```
            alert("The content does not have that many lines.");
            document.getElementById("hiliter").style.visibility = "hidden";
        }
  }
```

Related Item: `TextRectangle` object (Chapter 33, on the CD-ROM)

getElementsByTagName("*tagName*")

Returns: Array of element objects

Compatibility: WinIE5+, MacIE5+, NN6+, Moz+, Safari+, Opera+, Chrome+

The `getElementsByTagName()` method returns an array of all elements contained by the current object whose tags match the tag name supplied as the sole parameter to the method. The tag name parameter must be in the form of a string and is case insensitive. The group of elements returned in the array includes only those elements that are within the containment scope of the current object. Therefore, if you have two table objects in a document, and you invoke the `getElementsByTagName("td")` method on one of them, the list of returned table cell elements is confined to those cells within the current table object. The current element is not included in the returned array.

For MacIE5, WinIE6+, and all other supporting browsers, the method accepts a wildcard character (`"*"`) for matching descendent elements regardless of tag name. The resulting array of elements is nearly identical to what IE4+ returns via the `document.all` collection.

Example

Use The Evaluator (see Chapter 4) to experiment with the `getElementsByTagName()` method. Enter the following statements one at a time in the top text box, and study the results:

```
document.body.getElementsByTagName("div")
document.body.getElementsByTagName("div").length
document.getElementById("myTable").getElementsByTagName("td").length
```

Because the `getElementsByTagName()` method returns an array of objects, you can use one of those returned values as a valid element reference:

```
document.getElementsByTagName("form")[0].getElementsByTagName("input").length
```

Related Items: `getElementByTagNameNS()`, `getElementById()`, `tags()` methods

getElementsByTagNameNS("*namespaceURI*", "*localName*")

Returns: Array of element objects

Compatibility: WinIE-, MacIE-, NN6+, Moz+, Safari+, Opera+, Chrome+

This method returns an array of all elements contained by the current object (within an XML document) as specified in the two parameters. The first parameter of the method is a URI string matching a URI assigned to a label in the document. The second parameter is the local name portion of the attribute whose value you are getting.

Returned value types from `getAttributeNS()` are either strings (including attribute values assigned as unquoted numeric values) or Booleans (for example, the `multiple` property of a `select` element object).

Related Items: `attributes`, `namespaceURI`, `localName` properties; `getElementsByTagNameNS()`, `getElementById()`, `tags()` methods

getExpression("*attributeName*")

Returns: String

Compatibility: WinIE5+, MacIE-, NN-, Moz-, Safari-, Opera-, Chrome-

The `getExpression()` method returns the text of the expression that was assigned to an element's attribute via the `setExpression()` method. The returned value is not the value of the expression, but the expression itself. If you want to find out the current value of the expression (assuming that the variables used are within the scope of your script), you can use the `eval()` function on the call to `getExpression()`. This action converts the string to a JavaScript expression and returns the evaluated result.

One parameter, a string version of the attribute name, is required.

Example

See Listing 26-32 for the `setExpression()` method. This listing demonstrates the kinds of values returned by `getExpression()`.

Related Items: `document.recalc()`, `removeExpression()`, `setExpression()` methods

getFeature("*feature*", "*version*")

Returns: Object

Compatibility: WinIE-, MacIE-, NN-, Moz1.7.2+, Safari-, Opera+, Chrome-

According to the W3C DOM specification, the `getFeature()` method accepts a scripting feature and version, and returns an object that implements the APIs for the feature. Examples of possible feature parameters to this method are `Core` and `Events`, which correspond to DOM modules.

As recently as Mozilla 1.8.1 (Firefox 2.0), the `getFeature()` method returns an object but the object exposes no API features to the script.

Related Item: `implementation.hasFeature()` method

getUserData("*key*")

Returns: Object

Compatibility: WinIE-, MacIE-, NN6-, Moz1.7.2+, Safari+-, Opera-, Chrome+

The `getUserData()` method enables you to access custom user data that has been associated with a node. A given node can have multiple objects of user data, in which case each one is identified

through a text key. This key is the parameter that you pass into `getUserData()` to obtain a user data object. As of Mozilla 1.8.1 (Firefox 2.0), the method is only partially implemented and, therefore, still not useful.

hasAttribute("*attributeName*")

Returns: Boolean

Compatibility: WinIE+, MacIE-, NN6+, Moz+, Safari+, Opera+, Chrome+

The `hasAttribute()` method returns `true` if the current object has an attribute whose name matches the sole parameter; it returns `false` otherwise.

Related Items: `hasAttributeNS()`, `hasAttributes()` methods

hasAttributeNS("*namespaceURI*", "*localName*")

Returns: Boolean

Compatibility: WinIE-, MacIE-, NN6+, Moz+, Safari+, Opera+, Chrome+

The `hasAttributeNS()` method returns `true` if the current object has an attribute as identified by the two parameters; it returns `false` otherwise. The first parameter of the method is a URI string matching a URI assigned to a label in the document. The second parameter is the local name portion of the attribute whose value you are getting.

Related Items: `attributes`, `namespaceURI`, `localName` properties; `hasAttribute()`, `hasAttributes()` methods

hasAttributes()

Returns: Boolean

Compatibility: WinIE+, MacIE-, NN6+, Moz+, Safari+, Opera+, Chrome+

The `hasAttributes()` method returns `true` if the current object has any attributes explicitly assigned within the tag; it returns `false` otherwise.

Related Items: `hasAttribute()`, `hasAttributeNS()` methods

hasChildNodes()

Returns: Boolean

Compatibility: WinIE5+, MacIE5+, NN6+, Moz+, Safari+, Opera+, Chrome+

The `hasChildNodes()` method returns `true` if the current object has child nodes nested within; it returns `false` otherwise. A child node is not necessarily the same as a child element, so the following two expressions return `true` when the current object has at least one child node:

```
document.getElementById("myObject").hasChildNodes()
document.getElementById("myObject").childNodes.length > 0
```

You cannot use the second expression interchangeably with the following statement (which uses the `children` property):

```
document.getElementById("myObject").children.length > 0
```

You generally use the `hasChildNodes()` method in a conditional expression to make sure such nodes exist before performing operations on them:

```
if (document.getElementById("myObject").hasChildNodes()
{
    // [statements that apply to child nodes]
}
```

Example

Use The Evaluator (see Chapter 4) to experiment with the `hasChildNodes()` method. If you enter the following statement in the top text box

```
document.getElementById("myP").hasChildNodes()
```

the returned value is `true`. You can find out how many nodes there are by getting the `length` of the `childNodes` array:

```
document.getElementById("myP").childNodes.length
```

This expression reveals a total of three nodes: the two text nodes and the `em` element between them. Check out whether the first text node has any children:

```
document.getElementById("myP").childNodes[0].hasChildNodes()
```

The response is `false` because text fragments do not have any nested nodes. But check out the `em` element, which is the second child node of the `myP` element:

```
document.getElementById("myP").childNodes[1].hasChildNodes()
```

The answer is `true` because the `em` element has a text fragment node nested within it. Sure enough, the statement

```
document.getElementById("myP").childNodes[1].childNodes.length
```

yields a node count of `1`. You can also go directly to the `em` element in your references:

```
document.getElementById("myEM").hasChildNodes()
document.getElementById("myEM").childNodes.length
```

If you want to see the properties of the text fragment node inside the `em` element, enter the following in the bottom text box:

```
document.getElementById("myEM").childNodes[0]
```

You can see that the `data` and `nodeValue` properties for the text fragment return the text `"all"`.

Related Items: `childNodes` property; `appendChild()`, `removeChild()`, `replaceChild()` methods.

`insertAdjacentElement("`*`location`*`", `*`elementObject`*`)`

Returns: Object

Compatibility: WinIE5+, MacIE-, NN-, Moz-, Safari+, Opera+, Chrome+

The `insertAdjacentElement()` method inserts an element object (coming from a variety of sources) into a specific position relative to the current object. Both parameters are required. The first must be one of four possible case-insensitive locations for the insertion, shown in the following table:

Location	Description
`beforeBegin`	Before the current element's start tag
`afterBegin`	After the start tag but before any nested content
`beforeEnd`	Before the end tag but after all other nested content
`afterEnd`	After the end tag

These locations are relative to the current object. The element type of the current object (a block-level or inline element) has great bearing on how the inserted element is rendered. For example, suppose that you create a `b` element (using `document.createElement()`) and assign some inner text to it. You then use `insertAdjacentElement()` in an effort to insert this `b` element before some text in a `p` element. Because a `p` element is a block-level element, the location `beforeBegin` places the new `b` element before the start tag of the `p` element. This means, however, that the bold text appears in a text line above the start of the `p` element because a `<p>` tag begins a new block at the left margin of its container (unless instructed otherwise by style sheets). The resulting HTML looks like the following:

```
<b>The new element.</b><p>The original paragraph element.</p>
```

To make the new `b` element a part of the `p` element — but in front of the existing `p` element's content — use the `afterBegin` location. The resulting HTML looks like the following:

```
<p><b>The new element.</b>The original paragraph element.</p>
```

To complete the demonstration of the four location types, the following is the result of the `beforeEnd` location:

```
<p>The original paragraph element. <b>The new element.</b></p>
```

And this is the result of the `afterEnd` location:

```
<p>The original paragraph element.</p><b>The new element.</b>
```

The object to be inserted is a reference to an element object. The object reference can come from any expression that evaluates to an element object or, more likely, from the result of the `document.createElement()` method. Bear in mind that the object generated by `document.createElement()` initially has no content, and all attribute values are set to default values. Moreover, the object is passed to `insertAdjacentElement()` by reference, which means that there is only one instance of that object. If you attempt to insert that object in two places with two statements, the object is moved from the first location to the second. If you need to copy an existing object so that the original is not moved or otherwise disturbed by this method, use the `cloneNode()` method to specify the `true` parameter to capture all nested content of the node.

Example

Use The Evaluator (see Chapter 4) in WinIE5+ to experiment with the `insertAdjacentElement()` method. The goal of the experiment is to insert a new `h1` element above the `myP` element.

All actions require you to enter a sequence of statements in the top text box. Begin by storing a new element in the global variable `a`:

```
a = document.createElement("h1")
```

Give the new object some text:

```
a.innerText = "New Header"
```

Now insert this element before the start of the `myP` object:

```
myP.insertAdjacentElement("beforeBegin", a)
```

Notice that you have not assigned an `id` property value to the new element. But because the element was inserted by reference, you can modify the inserted object by changing the object stored in the `a` variable:

```
a.style.color = "red"
```

The inserted element is also part of the document hierarchy, so you can access it through hierarchy references such as `myP.previousSibling`.

The parent element of the newly inserted element is the `body`. Thus, you can inspect the current state of the HTML for the rendered page by entering the following statement in the top text box:

```
document.body.innerHTML
```

If you scroll down past the first form, you can find the `<h1>` element that you added along with the `style` attribute.

Related Items: `document.createElement()`, `applyElement()` methods

insertAdjacentHTML("*location*", "*HTMLtext*")
insertAdjacentText("*location*", "*text*")

Returns: Nothing

Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari+, Opera+, Chrome+

These two methods insert HTML or straight text at a location relative to the current element. They are intended for use after a page loads, rather than inserting content while the page loads (in which case you can use `document.write()` wherever you need evaluated content to appear on the page).

The first parameter must be one of four possible case-insensitive locations for the insertion, shown in the following table:

Location	Description
`beforeBegin`	Before the current element's start tag
`afterBegin`	After the start tag but before any nested content
`beforeEnd`	Before the end tag but after all other nested content
`afterEnd`	After the end tag

These locations yield the same results as described in the `insertAdjacentElement()` function discussed earlier in this chapter.

Whether you use `insertAdjacentHTML()` or `insertAdjacentText()` depends on the nature of your content and what you want the browser to do with it. If the content contains HTML tags that you want the browser to interpret and render as though it were part of the page source code, use the `insertAdjacentHTML()` method. All tags become objects in the document's object model. But if you want only to display some text (including HTML tags in their raw form), use `insertAdjacentText()`. The rendering engine does not interpret any tags included in the string passed as the second parameter. Instead, these tags are displayed as characters on the page. This distinction is identical to the one between the `innerHTML` and `innerText` properties.

The difference between `insertAdjacentHTML()` and `insertAdjacentElement()` is the nature of the content that you insert. The former enables you to accumulate the HTML as a string, whereas the latter requires the creation of an element object. Also, the two methods in this section work with IE4+ (including Mac versions), whereas `insertAdjacentElement()` requires the newer object model of WinIE5+.

If the HTML you pass as the second parameter of `insertAdjacentHTML()` contains `<script>` tags, you must set the `defer` attribute in the opening tag. This prevents script statements from executing as you insert them.

Example

Use The Evaluator (see Chapter 4) to experiment with these two methods. The example here demonstrates the result of employing both methods in an attempt to add some HTML to the beginning of the `myP` element.

Begin by assigning a string of HTML code to the global variable `a`:

```
a = "<b id='myB'>Important News!</b>"
```

Because this HTML is to go on the same line as the start of the `myP` paragraph, use the `afterBegin` parameter for the insert method:

```
myP.insertAdjacentHTML("afterBegin", a)
```

Notice that there is no space after the exclamation mark of the inserted HTML. But to prove that the inserted HTML is genuinely part of the document's object model, now you can insert the text of a space after the `b` element whose ID is `myB`:

```
myB.insertAdjacentText("afterEnd", " ")
```

Each time you evaluate the preceding statement (by repeatedly clicking the Evaluate button or pressing Enter with the cursor in the top text box), another space is added.

You should also see what happens when the string to be inserted with `insertAdjacentText()` contains HTML tags. Reload The Evaluator, and enter the following two statements in the top text box, evaluating each one in turn:

```
a = "<b id='myB'>Important News!</b>"
myP.insertAdjacentText("afterBegin", a)
```

The HTML is not interpreted but is displayed as plain text. There is no object named `myB` after executing this latest insert method.

Related Items: `innerText`, `innerHTML`, `outerText`, `outerHTML` properties; `insertAdjacentElement()`, `replaceAdjacentText()` methods

insertBefore(*newChildNodeObject, referenceChildNode*)

Returns: `Node` object

Compatibility: WinIE5+, MacIE5+, NN6+, Moz+, Safari+, Opera+, Chrome+

The `insertBefore()` method is the W3C DOM syntax for inserting a new child node into an existing element. Node references for both parameters must be valid `Node` objects (including those that `document.createElement()` generates).

The behavior of this method might seem counterintuitive at times. If you include the second parameter (a reference to an existing child node of the current element — optional in IE), the new child node is inserted before that existing one. But if you omit the second parameter (or its value is `null`), the new child node is inserted as the last child of the current element — in which case the method acts the same as the `appendChild()` method. The true power of this method is summoned when you specify that second parameter; from the point of view of a parent element, you can drop a new child into any spot among its existing children. If an inserted node already exists in the document tree, it will be removed from its previous position.

Bear in mind that the `insertBefore()` method works from a parent element. Internet Explorer provides additional methods, such as `insertAdjacentElement()`, to operate from the perspective of what will become a child element.

Example

Listing 26-28 demonstrates how the `insertBefore()` method can insert child elements (`li`) inside a parent (`ol`) at different locations, depending on the second parameter. A text box enables you to enter your choice of text and/or HTML for insertion at various locations within the `ol` element. If you don't specify a position, the second parameter of `insertBefore()` is passed as `null` — meaning that the new child node is added to the end of the existing children. But choose a spot from the select

list where you want to insert the new item. The value of each `select` list option is an index of one of the first three child nodes of the `ol` element.

LISTING 26-28

Using the insertBefore() Method

HTML: jsb26-28.html

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>insertBefore() Method</title>
      <script type="text/javascript" src="../jsb-global.js"></script>
      <script type="text/javascript" src="jsb26-28.js"></script>
   </head>
   <body>
      <h1>insertBefore() Method</h1>
      <hr />
      <form onsubmit="return false">
         <p>Enter text or HTML for a new list item:
            <input type="text"
                   name="newText" size="40" value="" />
         </p>
         <p>Before which existing item?
            <select name="itemIndex">
               <option value="null">None specified</option>
               <option value="0">1</option>
               <option value="1">2</option>
               <option value="2">3</option>
            </select>
         </p>
         <input type="button" value="Insert Item"
                onclick="doInsert(this.form)" />
      </form>
      <ol id="myUL">
         <li>Originally the First Item</li>
         <li>Originally the Second Item</li>
         <li>Originally the Third Item</li>
      </ol>
   </body>
</html>
```

JavaScript: jsb26-28.js

```
function doInsert(form)
{
   if (form.newText)
   {
      var newChild = document.createElement("LI");
      newChild.innerHTML = form.newText.value;
```

```
        var choice = form.itemIndex.options[form.itemIndex.selectedIndex].value;
        var insertPoint = (isNaN(choice)) ?
           null : document.getElementById("myUL").childNodes[choice];
        document.getElementById("myUL").insertBefore(newChild, insertPoint);
    }
}
```

Related Items: `appendChild()`, `replaceChild()`, `removeChild()`, `insertAdjacentElement()` methods

`isDefaultNamespace("`*`namespaceURI`*`")`

Returns: Boolean

Compatibility: WinIE-, MacIE-, NN6-, Moz1.7.2+, Safari+, Opera+, Chrome+

This method checks whether the specified namespace matches the default namespace of the current node.

`isEqualNode(`*`nodeRef`*`)`
`isSameNode(`*`nodeRef`*`)`

Returns: Integer ID

Compatibility: WinIE-, MacIE-, NN-, Moz1.7.2+, Safari+, Opera+, Chrome+

When it comes to nodes, there is a distinct difference between a node being equal to another node and a node being the same as another node. Equality has a very specific meaning with respect to nodes: Two nodes are considered equal if they have the same values for the `attributes`, `childNodes`, `localname`, `namespaceURI`, `nodeName`, `nodeType`, `nodeValue`, and `prefix` properties. Together, these properties essentially reflect the content of a node. What they don't reflect is the relative position of a node within a document, which means that nodes can be equal and reside in different locations in the node tree. Two nodes are considered the same if ... well, they are the same identical node. The `isEqualNode()` method checks for node equality, whereas `isSameNode()` checks whether two nodes are the same. Both methods expect a node reference as their only parameter. Currently, Opera does not support the `isEqualNode()` method.

`isSupported("`*`feature`*`", "`*`version`*`")`

Returns: Boolean

Compatibility: WinIE-, MacIE-, NN6+, Moz+, Safari+, Opera+, Chrome+

The `isSupported()` method returns `true` if the current node supports required portions of the specified W3C DOM module and version; it returns `false` otherwise. The first parameter accepts any of the following case-sensitive DOM module name strings: `Core`, `XML`, `HTML`, `Views`, `StyleSheets`, `CSS`, `CSS2`, `Events`, `UIEvents`, `MouseEvents`, `MutationEvents`, `HTMLEvents`, `Range`, and `Traversal`. The second parameter accepts a string representation of the major and minor DOM module version, such as `"2.0"` for DOM Level 2.

Example

Use The Evaluator (see Chapter 4) to experiment with the `isSupported()` method. If you have multiple versions of NN6 or later and Mozilla, try the following (and others) to see how the support for various modules has evolved:

```
document.body.isSupported("CSS", "2.0")
document.body.isSupported("CSS2", "2.0")
document.body.isSupported("Traversal", "2.0")
```

If you have access to Safari, Opera, or Chrome, try the same methods there to see the differences in modules supported compared with Mozilla-based browsers.

item(*index* | "*index*" [, *subIndex*])

Returns: Object

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

The `item()` method works with most objects that are themselves collections of other objects. In W3C DOM terminology, these kinds of objects are known as *named node lists* (for objects such as nodes and attributes) or *HTML collections* (for objects such as elements of a form). You may call the `item()` method with a single numeric parameter that is the index value of the desired object within the collection. If you know the index number of the item, you can use JavaScript array syntax instead. The following two statements return the same object reference:

```
document.getElementById("myTable").childNodes.item(2)
document.getElementById("myTable").childNodes[2]
```

The method also supports a string of the ID of an object within the collection. (Integer values are required for the `attributes`, `rules`, and `TextRectangle` objects, however.) Additionally, if the collection has more than one object with the same ID (never a good idea except when necessary), a second numeric parameter enables you to select which identically named group you want (using zero-based index values within that subgroup). This obviously does not apply to collections, such as attributes and rules, which have no ID associated with them.

The method returns a reference to the object specified by the parameters.

Example

Use The Evaluator (see Chapter 4) to experiment with the `item()` method. Type the following statements in the top text box, and view the results for each.

W3C and IE5:

```
document.getElementById("myP").childNodes.length
document.getElementById("myP").childNodes.item(0).data
document.getElementById("myP").childNodes.item(1).nodeName
```

W3C, IE4, and IE5:

```
document.forms[1].elements.item(0).type
```

The first approach is helpful when your script is working with a string version of an object's name. If your script already knows the object reference, the second approach is more efficient and compact.

Related Items: All object element properties that return collections (arrays) of other objects

lookupNamespaceURI("*prefix*")
lookupPrefix("*namespaceURI*")

Returns: Namespace or prefix string (see description)

Compatibility: WinIE-, MacIE-, NN-, Moz1.7.2+, Safari+, Opera+, Chrome+

These two methods use one piece of information to look up the other. The `lookupNamespaceURI()` method accepts a prefix as its only parameter and returns a URI string for the node if the prefix matches a previously defined namespace. Operating in the reverse, the `lookupPrefix()` method accepts a namespace URI string and returns a prefix string for the node if the namespace parameter matches a previously defined namespace.

mergeAttributes("*sourceObject*")

Returns: Nothing

Compatibility: WinIE5+, MacIE-, NN-, Moz-, Safari-, Opera-, Chrome-

The `mergeAttributes()` method is a convenient way to propagate attributes in newly created elements without painstakingly adding attributes one at a time. When you have an object whose attributes can function as a prototype for other elements, those attributes (except for the `id` attribute) can be applied to a newly created element instantaneously. The default action of this method is not to duplicate the `id` or `name` attributes of the element. However, IE5.5+ introduced an extra Boolean parameter, `preserveIDs`, that enables you to duplicate these two attributes by setting the parameter to `false` (`true` is the default).

Example

Listing 26-29 demonstrates the usage of `mergeAttributes()` in the process of replicating the same form input field while assigning a unique ID to each new field. So that you can see the results as you go, we display the HTML for each input field in the field.

The `doMerge()` function begins by generating two new elements: a `p` element and an `input` element. Because these newly created elements have no properties associated with them, a unique ID is assigned to the `input` element through the `uniqueID` property. Attributes from the field in the source code (`field1`) are merged into the new `input` element. Thus, all attributes except `name` and `id` are copied to the new element. The `input` element is inserted into the `p` element, and the `p` element is appended to the document's form element. Finally, the `outerHTML` of the new element is displayed in its field. Notice that except for the `name` and `id` attributes, all others are copied. This includes style sheet attributes and event handlers. To prove that the event handler works in the new elements, you can add a space to any one of them and press Tab to trigger the `onchange` event handler that changes the content to all-uppercase characters.

LISTING 26-29

Using the mergeAttributes() Method

HTML: jsb2-29.html

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>mergeAttributes() Method</title>
      <script type="text/javascript" src="../jsb-global.js"></script>
      <script type="text/javascript" src="jsb26-29.js"></script>
   </head>
   <body
   onload="document.expandable.field1.value = ↩
           document.expandable.field1.outerHTML">
      <h1>mergeAttributes() Method</h1>
      <hr />
      <form name="expandable" onsubmit="return false">
         <p>
            <input type="button" value="Append Field 'Clone'"
                   onclick="doMerge(this.form)" />
         </p>
         <p>
            <input type="text" name="field1" id="FIELD1" size="120" value=""
                   style="font-size:9pt" onchange="upperMe(this)" />
         </p>
      </form>
   </body>
</html>
```

JavaScript: jsb26-29.js

```
function doMerge(form)
{
   var newPElem = document.createElement("p");
   var newInputElem = document.createElement("input");
   newInputElem.id = newInputElem.uniqueID;
   newInputElem.mergeAttributes(form.field1);
   newPElem.appendChild(newInputElem);
   form.appendChild(newPElem);
   newInputElem.value = newInputElem.outerHTML;
}

// called by onChange event handler of fields
function upperMe(field)
{
   field.value = field.value.toUpperCase();
}
```

Related Items: `clearAttributes()`, `cloneNode()`, `removeAttributes()` methods

normalize()

Returns: Nothing

Compatibility: WinIE6+, MacIE5+, NN7+, Moz+, Safari1.2+, Opera+, Chrome+

In the course of appending, inserting, removing, and replacing child nodes of an element, it is conceivable that two text nodes can end up adjacent to each other. Although this typically has no effect on the rendering of the content, some XML-centric applications that rely heavily on the document node hierarchy to interpret content properly may not like having two text nodes sitting next to each other. The proper form of a node hierarchy is for a single text node to be bounded by other node types. The `normalize()` method sweeps through the child nodes of the current node object and combines adjacent text nodes into a single text node. The effect obviously impacts the number of child nodes of an element, but it also cleanses the nested node hierarchy.

Example

Use The Evaluator (see Chapter 4) to experiment with the `normalize()` method. The following sequence adds a text node adjacent to one in the `myP` element. A subsequent invocation of the `normalize()` method removes the division between the adjacent text nodes.

Begin by confirming the number of child nodes of the `myP` element:

```
document.getElementById("myP").childNodes.length
```

Three nodes initially inhabit the element. Next, create a text node, and append it as the last child of the `myP` element:

```
a = document.createTextNode("This means you!")
document.getElementById("myP").appendChild(a)
```

With the new text now rendered on the page, the number of child nodes increases to four:

```
document.getElementById("myP").childNodes.length
```

You can see that the last child node of `myP` is the text node you just created:

```
document.getElementById("myP").lastChild.nodeValue
```

But by invoking `normalize()` on `myP`, all adjacent text nodes are accumulated into single nodes:

```
document.getElementById("myP").normalize()
```

You can see that the `myP` element is back to three child nodes, and the last child is a combination of the two previously distinct, but adjacent, text nodes:

```
document.getElementById("myP").childNodes.length
document.getElementById("myP").lastChild.nodeValue
```

Related Items: `document.createTextNode()`, `appendChild()`, `insertBefore()`, `removeChild()`, `replaceChild()` methods

releaseCapture()
setCapture(containerBoolean)

Returns: Nothing

Compatibility: WinIE5+, MacIE-, NN-, Moz-, Safari-, Opera-, Chrome-

You can instruct a single object on a page to capture all mouse events (`onmousedown`, `onmouseup`, `onmousemove`, `onmouseout`, `onmouseover`, `onclick`, and `ondblclick`) via the WinIE-specific `setCapture()` method. A primary scenario for mouse event capture is when some content appears on the page that you wish to leave as the center of user focus — items such as pull-down menus, context menus, or simulated modal window areas. When such items appear onscreen, you want the effect of blocking all mouse events except those that apply to the menu or currently visible pseudowindow. When the region disappears, mouse events can be released so that individual elements (such as buttons and links elsewhere on the page) respond to mouse events.

Event capture does not block the events. Instead, the events are redirected to the object set to capture all mouse events. Events bubble up from that point unless explicitly canceled (see Chapter 32). For example, consider a document that has a `<body>` tag containing an `onclick` event handler that governs the entire document at all times. If you turn on event capture for a `div` somewhere in the document, the click event first goes to the `div`. That `div` might have an `onclick` event handler that looks to process click events when they occur in some of its child elements. If the event handler for the `div` does not also cancel the bubbling of that click event, the `body` element's `onclick` event handler eventually receives and processes the event, even though the `div` initially captured the event.

Deciding which object should capture events is an important design issue to confront. With event capture engaged, all mouse events (no matter where they occur) get funneled to the object set to capture the events. Therefore, if you design an application whose entire interface consists of clicking and dragging positionable elements, you can set one of those elements (or even the `document` object) to perform the capturing. For pop-up regions, however, it is generally more logical and convenient for your coding to assign the capture mechanism to the primary container of the pop-up content (usually, a positioned `div`).

The `setCapture()` method has one optional Boolean parameter. The parameter controls whether mouse events on child elements within the capturing object are under control of the event capture mechanism. The default value (`true`) means that all mouse events targeted at elements within the current object go to the current object rather than to the original target — the most likely way you will use `setCapture()` for things such as pop-up and context menus. But if you specify `false` as the parameter, mouse events occurring in child elements of the capturing container receive their events directly. From there, regular event bubbling upward from the target ensues (see Chapter 32).

You may encounter odd behavior when the region you set up to capture mouse events contains form elements such as text input fields and `select` lists. Because these elements require mouse events to gain focus for interaction, the event capture mechanism inhibits access to these items. To work around this behavior, you can examine the click event's `srcElement` property to see whether the click was on one of these elements and script the focus of that element (or instruct the user to press the Tab key until the element gets focus manually).

After an object is set to capture events, your other code must define which events actually do something and decide whether events should bubble up beyond the capturing element. You need to worry about bubbling only if your design includes mouse event handlers in elements higher up the element containment hierarchy. You may not want those event handlers to fire while event capture is on; in this case, you need to cancel the bubbling of those events in the capturing object.

If your application design requires that the pop-up area be hidden and event handling be returned to normal (such as after the user makes a pop-up menu selection), use the `releaseCapture()` method

in conjunction with hiding the container. Because event capture can be engaged for only one element at a time, you can release capture by invoking the `releaseCapture()` method from the container or from the `document` object.

Event capture is automatically disengaged when the user performs any of the following actions:

- Gives focus to any other window
- Displays any system modal dialog window (for example, alert window)
- Scrolls the page
- Opens a browser context menu (by right-clicking)
- Tabs to give focus to the Address field in the browser window

Therefore, you may want to set the `document` object's `onlosecapture` event handler to hide any container that your script displays in concert with event capture.

Also be aware that even though mouse events may be captured to prevent mouse access to the rest of the page, keyboard events are not captured. Thus, using the event capture mechanism to simulate modal windows is not foolproof: A user can tab to any form element or link in the page and press the spacebar or Enter key to activate that element.

Event capture, as defined in the W3C DOM, operates differently from WinIE event capture. In the W3C DOM, you can instruct the browser to substitute event capture of any kind of event for the normal event bubbling behavior. For example, you can attach an event listener to the `body` element in such a way that it sees all click events aimed at elements contained by the `body` element before the events reach their target elements. (See Chapter 25 and Chapter 32, for more on the W3C DOM event model and how to integrate it into cross-browser applications.)

Example

Listing 26-30 demonstrates the usage of `setCapture()` and `releaseCapture()` in a quick-and-dirty context menu for WinIE5+. The job of the context menu is to present a list of numbering styles for the ordered list of items on the page. Whenever the user brings up the context menu atop the `ol` element, the custom context menu appears. Event capture is turned on in the process to prevent mouse actions elsewhere on the page from interrupting the context menu choice. Even a click of the link set up as the title of the list is inhibited while the context menu is visible. A click anywhere outside the context menu hides the menu. Clicking a choice in the menu changes the `listStyleType` property of the `ol` object and hides the menu. Whenever the context menu is hidden, event capture is turned off so that clicking the page (such as the link) works as normal.

For this design, `onclick`, `onmouseover`, and `onmouseout` event handlers are assigned to the `div` element that contains the context menu. To trigger the display of the context menu, the `ol` element has an `oncontextmenu` event handler. This handler invokes the `showContextMenu()` function. In this function, event capture is assigned to the context menu `div` object. The `div` is also positioned at the location of the click before it is set to be visible. To prevent the system's regular context menu from also appearing, the `event` object's `returnValue` property is set to `false`.

Now that all mouse events on the page go through the `contextMenu div` object, let's examine what happens with different kinds of events triggered by user action. As the user rolls the mouse, a flood of `mouseover` and `mouseout` events fires. The event handlers assigned to the `div` manage these events. But notice that the two event handlers, `highlight()` and `unhighlight()`, perform action only when the `srcElement` property of the event is one of the menu items in the `div`. Because the page has no other `onmouseover` or `onmouseout` event handlers defined for elements up the containment hierarchy, you do not have to cancel event bubbling for these events.

When a user clicks the mouse button, different things happen, depending on whether event capture is enabled. Without event capture, the `click` event bubbles up from wherever it occurred to the `onclick` event handler in the `body` element. (An alert dialog box displays to let you know when the event reaches the `body`.) But with event capture turned on (the context menu is showing), the `handleClick()` event handler takes over to apply the desired choice whenever the click is atop one of the context menu items. For all `click` events handled by this function, the context menu is hidden, and the `click` event is canceled from bubbling up any higher (no alert dialog box appears). This takes place whether the user makes a choice in the context menu or clicks anywhere else on the page. In the latter case, all you need is for the context menu to go away as the real context menu does. For added insurance, the `onlosecapture` event handler hides the context menu when a user performs any of the actions just listed that cancel capture.

LISTING 26-30

Using setCapture() and releaseCapture()

HTML: jsb26-30.html

```
<!DOCTYPE html>
<html>
  <head>
    <meta http-equiv="content-type" content="text/html;charset=utf-8">
    <style type="text/css">
       #contextMenu
       {
          position:absolute; background-color:#cfcfcf;
          border-style:solid; border-width:1px;
          border-color:#EFEFEF #505050 #505050 #EFEFEF;
          padding:3px 10px; font-size:8pt; font-family:Arial, Helvetica;
          line-height:150%; visibility:hidden;
       }
       .menuItem
       {
          color:black;
       }
       .menuItemOn
       {
          color:white
       }
       ol
       {
          list-style-position:inside; font-weight:bold; cursor:hand;
       }
       li
       {
          font-weight:normal;
       }
    </style>
    <script type="text/javascript" src="../jsb-global.js"></script>
    <script type="text/javascript" src="jsb26-30.js"></script>
  </head>
```

```
  <body onclick="alert('You reached the document object.')">
    <ol id="shapesList" oncontextmenu="showContextMenu()">
      <li style="list-style: none"><a
       href="javascript:alert('A%20sample%20link.')">Three-DimensionalShapes</a>
      </li>
      <li value="1">Circular Cylinder</li>
      <li>Cube</li>
      <li>Rectangular Prism</li>
      <li>Regular Right Pyramid</li>
      <li>Right Circular Cone</li>
      <li>Sphere</li>
    </ol>
    <div id="contextMenu" onlosecapture="hideMenu()" onclick="handleClick()"
         onmouseover="highlight()" onmouseout="unhighlight()">
       <span id="menuItem1" class="menuItem"
             listtype="upper-alpha">A,B,C,...</span>
       <br />
       <span id="menuItem2" class="menuItem"
             listtype="lower-alpha">a,b,c,...</span>
       <br />
       <span id="menuItem3" class="menuItem"
             listtype="upper-roman">I,II,III,...</span>
       <br />
       <span id="menuItem4" class="menuItem"
             listtype="lower-roman">i,ii,iii,...</span>
       <br />
       <span id="menuItem5" class="menuItem"
             listtype="decimal">1,2,3,...</span>
    </div>
  </body>
</html>
```

JavaScript: jsb26-30.js

```
function showContextMenu()
{
   contextMenu.setCapture();
   contextMenu.style.pixelTop = event.clientY + document.body.scrollTop;
   contextMenu.style.pixelLeft = event.clientX + document.body.scrollLeft;
   contextMenu.style.visibility = "visible";
   event.returnValue = false;
}

function revert()
{
   document.releaseCapture();
   hideMenu();
}

function hideMenu()
{
   contextMenu.style.visibility = "hidden";
}
```

continued

LISTING 26-30 *(continued)*

```
function handleClick()
{
   var elem = window.event.srcElement;
   if (elem.id.indexOf("menuItem") == 0)
   {
      document.getElementById("shapesList").style.listStyleType = elem.listtype;
   }
   revert();
   event.cancelBubble = true;
}

function highlight()
{
   var elem = event.srcElement;
   if (elem.className == "menuItem")
   {
      elem.className = "menuItemOn";
   }
}

function unhighlight()
{
   var elem = event.srcElement
   if (elem.className == "menuItemOn")
   {
      elem.className = "menuItem";
   }
}
```

Related Items: `addEventListener()`, `dispatchEvent()`, `fireEvent()`, `removeEventListener()` methods; `onlosecapture` event; `Event` object (Chapter 32)

removeAttribute("*attributeName*"[, *caseSensitivity*])

Returns: Boolean (IE), nothing (NN/DOM)

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

If you create an attribute with the `setAttribute()` method, you can eliminate that attribute from the element object via the `removeAttribute()` method. The required parameter is the name of the attribute. Internet Explorer permits you to set and remove attributes such that the attribute names are case sensitive. The default behavior of `removeAttribute()` in IE (the second parameter is a Boolean value) is `false`. Therefore, if you supply a value of `true` for the case-sensitivity parameter in `setAttribute()`, you should set the parameter to `true` in `removeAttribute()` to ensure a proper balance between created and removed attributes.

The W3C DOM (NN/Moz-/WebKit-based) version of the `removeAttribute()` method has a single parameter (a case-insensitive attribute name) and returns no value. The returned value in IE is `true`

if the removal succeeds and `false` if it doesn't succeed (or if the attribute is one that you set in some other manner).

Example

Use The Evaluator (see Chapter 4) to experiment with the `removeAttribute()` method for the elements in the page. See the examples for the `setAttribute()` method later in this chapter, and enter the corresponding `removeAttribute()` statements in the top text box. Interlace statements using `getAttribute()` to verify the presence or absence of each attribute.

Related Items: `attributes` property; `document.createAttribute()`, `getAttribute()`, `setAttribute()` methods

removeAttributeNode(*attributeNode*)
setAttributeNode(*attributeNode*)

Returns: `Attribute` object

Compatibility: WinIE6+, MacIE-, NN6+, Moz+, Safari+, Opera+, Chrome+

As discussed in the coverage of the `getAttributeNode()` method earlier in this chapter, the W3C DOM treats a name–value attribute pair as an attribute object. An attribute object is a distinct node within a named node map — a collection of attribute objects belonging to an element. Understanding named node maps and attribute objects is more useful in an XML environment, where attributes can not only contain valuable data, but also are not exposed to the DOM as properties you can access via script. Instead of accessing an object's properties, you work with the actual attributes.

If you want to insert an attribute in the formal W3C methodology, you can use `document.createAttribute()` to generate a new attribute object. Subsequent script statements assign values to the `nodeName` and `nodeValue` properties to give the attribute its traditional name–value pair. You can then insert that new attribute object into the attribute list of an object via the `setAttributeNode()` method. The sole parameter is an attribute object, and the return value is a reference to the newly inserted attribute object.

To remove an attribute node from an element using this syntax, employ the `removeAttributeNode()` method. Again, the sole parameter is an attribute object. If your script knows only the attribute's name, you can use `getAttributeNode()` to obtain a valid reference to the attribute object. The `removeAttributeNode()` method returns a reference to the removed attribute object. That object remains in the browser's memory, but it is not part of the document hierarchy. By capturing this removed attribute object in a variable, you have the flexibility to modify and assign it to another object elsewhere in the document.

In practice, you may rarely, if ever, need to address attributes as nodes. Other methods — notably `getAttribute()`, `removeAttribute()`, and `setAttribute()` — do the job when your scripts have only the name (as a string) of an attribute belonging to an element.

Example

Use The Evaluator (see Chapter 4) to experiment with the `setAttributeNode()` and `removeAttributeNode()` methods for the `p` element in the page. The task is to create and add a `style` attribute to the `p` element. Begin by creating a new attribute and storing it temporarily in the global variable `a`:

```
a = document.createAttribute("style")
```

Assign a value to the `attribute` object:

```
a.nodeValue = "color:red"
```

Now insert the new attribute into the `p` element:

```
document.getElementById("myP").setAttributeNode(a)
```

The paragraph changes color in response to the newly added attribute.

Due to the NN6 bug that won't allow the method to return a reference to the newly inserted attribute node, you can artificially obtain such a reference:

```
b = document.getElementById("myP").getAttributeNode("style")
```

Finally, use the reference to the newly added attribute to remove it from the element:

```
document.getElementById("myP").removeAttributeNode(b)
```

Upon the removal of the attribute, the paragraph resumes its initial color. See the example for the `setAttribute()` method later in this chapter to discover how you can perform this same kind of operation with `setAttribute()`.

Related Items: `attributes` property; `document.createAttribute()`, `getAttribute()`, `getAttributeNode()`, `setAttribute()` methods

removeAttributeNS("*namespaceURI*", "*localName*")

Returns: Nothing

Compatibility: WinIE-, MacIE-, NN6+, Moz+, Safari+, Opera+, Chrome+

This method removes the attribute specified in the two parameters. The first parameter of the method is a URI string matching a URI assigned to a label in the document. The second parameter is the local name portion of the attribute whose value you are removing.

Related Items: `attributes`, `namespaceURI`, `localName` properties; `removeAttribute()`, `getAttributeNS()`, `setAttributeNS()` methods

removeBehavior(*ID*)

Returns: Boolean

Compatibility: WinIE5+, MacIE-, NN-, Moz-, Safari-, Opera-, Chrome-

The `removeBehavior()` method detaches a behavior from an object. It assumes that the behavior was added to the object via the `addBehavior()` method. The return value of the `addBehavior()` method is a unique identifier for that particular behavior. This identifier is the required parameter for the `removeBehavior()` method. Thus, you can add two behaviors to an object and remove just one of them if you so desire. If the removal succeeds, the `removeBehavior()` method returns `true`; otherwise, it returns `false`.

Example

See Listing 26-19a and Listing 26-19b earlier in this chapter for examples of how to use `addBehavior()` and `removeBehavior()`.

Related Item: `addBehavior()` method

`removeChild(nodeObject)`

Returns: `Node` object reference

Compatibility: WinIE5+, MacIE5+, NN6+, Moz+, Safari+, Opera+, Chrome+

The `removeChild()` method erases a child element from the current element. Content associated with the child element is no longer visible on the page, and the object is no longer part of the document object hierarchy.

As destructive as that sounds, the specifications for the deleted object are not necessarily lost to the ether. The `removeChild()` method returns a reference to the removed node. By assigning this value to a variable, you can hold on to that object specification for insertion later in the session. You are free to use this value as a parameter to such methods as `appendChild()`, `replaceChild()`, `swapNode()`, and `insertBefore()`.

Remember that `removeChild()` is invoked from the point of view of a parent element. If you simply want to remove an element, you can do so more directly (in WinIE5+ only) with the `removeNode()` method. The IE `removeNode()` method also allows a node to remove itself, which isn't possible via the `removeChild()` method.

Example

You can see an example of `removeChild()` as part of Listing 26-21 earlier in this chapter.

Related Items: `appendChild()`, `replaceChild()`, `removeNode()` methods

`removeEventListener()`

(See `addEventListener()`)

`removeExpression("propertyName")`

Returns: Boolean

Compatibility: WinIE5+, MacIE-, NN-, Moz-, Safari-, Opera-, Chrome-

If you assign an expression to an object property (including an object's `style` object) via the `setExpression()` method, you can remove it under script control with the `removeExpression()` method. The sole parameter is the name of the property in string form. Property names are case sensitive.

The method returns `true` if the removal succeeds; otherwise, `false` is returned. Be aware that removing an expression does not alter the value that is currently assigned to the property. In other words, you can use `setExpression()` to set a property's value and then remove the expression so that no further changes are made when the document recalculates expressions. If this is your goal, however, you are probably better served by simply setting the property directly via scripting.

Example

You can experiment with all three expression methods in The Evaluator (Chapter 4). The following sequence adds an expression to a style sheet property of the `myP` element on the page and then removes it.

To begin, enter the number 24 in the bottom one-line text box in The Evaluator (but don't press Enter or click the List Properties button). This is the value used in the expression to govern the `fontSize` property of the `myP` object. Next, assign an expression to the `myP` object's `style` object by entering the following statement in the top text box:

```
myP.style.setExpression("fontSize","document.forms[0].↩
inspector.value","JScript")
```

Now you can enter different font sizes in the bottom text box and have the values immediately applied to the `fontSize` property. (Keyboard events in the text box automatically trigger the recalculation.) The default unit is `px`, but you can also append other units (such as `pt`) to the value in the text box to see how different measurement units influence the same numeric value.

Before proceeding to the next step, enter a value other than 16 (the default `fontSize` value). Finally, enter the following statement in the top text box to disconnect the expression from the property:

```
myP.style.removeExpression("fontSize")
```

Notice that although you can no longer adjust the font size from the bottom text box, the most recent value assigned to it sticks to the element. To prove it, enter the following statement in the top text box to see the current value:

```
myP.style.fontSize
```

Related Items: `document.recalc()`, `getExpression()`, `setExpression()` methods

`removeNode(`*`removeChildrenFlag`*`)`

Returns: `Node` object reference

Compatibility: WinIE5+, MacIE-, NN-, Moz-, Safari-, Opera-, Chrome-

You can use the `removeNode()` method to delete the current node from an element hierarchy in WinIE5+. The sole parameter is a Boolean value that directs the method to remove only itself (without its child nodes) or the node and all of its children (value of `true`). The method returns a reference to the node object removed. This removed object is no longer accessible to the DOM. But the returned value contains all properties of the object as it existed before you removed it (including properties such as `outerHTML` and explicitly set style sheet rules). Thus, you can use this value as a parameter to insert the node elsewhere in the document.

Although the W3C DOM does not have a `removeNode()` method, the cross-browser method whose behavior most closely resembles `removeNode()` is the `removeChild()` method. The scope of the `removeChild()` method is one level up the object hierarchy from the object you use for the `removeNode()` method.

Example

Examine Listing 26-21 for the `appendChild()` method to understand the difference between `removeChild()` and `removeNode()`. In the `restore()` function, you can replace this statement

```
mainObj.removeChild(oneChild);
```

in IE5+ with

```
oneChild.removeNode(true);
```

The difference is subtle, but it is important to understand. See Listing 26-31 later in this chapter for another example of the `removeNode()` method.

Related Items: `Node` object; `appendChild()`, `cloneChild()`, `removeChild()`, `replaceChild()`, `replaceNode()` methods

`replaceAdjacentText("location", "text")`

Returns: String

Compatibility: WinIE5+, MacIE-, NN-, Moz-, Safari-, Opera-, Chrome-

The `replaceAdjacentText()` method enables you to replace one chunk of document text with another in a specific position relative to the current object. Be aware that this method works only for plain text and not HTML tags. The returned value is the string of the text that you replace.

Both parameters are required. The first must be one of four possible case-insensitive locations for the insertion, shown in the following table:

Location	Description
`beforeBegin`	Before the current element's start tag
`afterBegin`	After the start tag but before any nested content
`beforeEnd`	Before the end tag but after all other nested content
`afterEnd`	After the end tag

This method is best used with inline (rather than block) elements when specifying the `beforeBegin` and `afterEnd` parameters. For example, if you attempt to use `replaceAdjacentText()` with `beforeBegin` on the second of two consecutive paragraph elements, the replacement text is inserted into the end of the first paragraph. You can think of the `replaceAdjacentText()` method in terms of text fragment nodes. The method replaces the text fragment node (given any one of the four position parameters) with new text. Replacing the text of a simple element with either the `afterBegin` or `beforeEnd` locations is the same as assigning that text to the object's `innerText` property.

Example

Use The Evaluator (see Chapter 4) to experiment with the `replaceAdjacentText()` method. Enter each of the following statements in the top text box, and watch the results in the `myP` element (and its nested `myEM` element) below the solid rule:

```
document.getElementById("myEM").replaceAdjacentText("afterBegin", "twenty")
```

Notice that the `myEM` element's new text picks up the behavior of the element. In the meantime, the replaced text (`all`) is returned by the method and displayed in the Results box:

```
document.getElementById("myEM").replaceAdjacentText("beforeBegin", "We need")
```

All characters of the text fragment, including spaces, are replaced. Therefore, you may need to supply a trailing space, as shown here, if the fragment you replace has a space:

```
document.getElementById("myP").replaceAdjacentText("beforeEnd", "good people.")
```

This is another way to replace the text fragment following the `myEM` element, but it is also relative to the surrounding `myP` element. If you now attempt to replace text after the end of the `myP` block-level element

```
document.getElementById("myP").replaceAdjacentText("afterEnd", "Hooray!")
```

the text fragment is inserted after the end of the `myP` element's tag set. The fragment is just kind of floating in the DOM as an unlabeled text node.

Related Items: `innerText`, `outerText` properties; `getAdjacentText()`, `insertAdjacentHTML()`, `insertAdjacentText()` methods

`replaceChild(`*`newNodeObject, oldNodeObject`*`)`

Returns: `Node` object reference

Compatibility: WinIE5+, MacIE5+, NN6+, Moz+, Safari+, Opera+, Chrome+

The `replaceChild()` method enables you to swap an existing child node object for a new node object. Parameters for the `replaceChild()` method are node object references, and they must be in the order of the new object followed by the object you want to replace. The old object must be an immediate child node of the parent used to invoke the method, and the new object must also be a legal child element within the document containment hierarchy.

The method returns a reference to the child object that you replaced with the new object. This reference can be used as a parameter to any of the node-oriented insertion or replacement methods.

Remember that `replaceChild()` is invoked from the point of view of a parent element. If you simply want to change an element, you can do so more directly in WinIE5+ with the `swapNode()` or `replaceNode()` method.

Example

You can see an example of `replaceChild()` as part of Listing 26-21 (for the `appendChild` property) earlier in this chapter.

Related Items: `appendChild()`, `removeChild()`, `replaceNode()`, `swapNode()` methods

`replaceNode("`*`newNodeObject`*`")`

Returns: `Node` object reference

Compatibility: WinIE5+, MacIE-, NN-, Moz-, Safari-, Opera-, Chrome-

The `replaceNode()` method is related to the `replaceChild()` method, but you invoke this method on the actual node you want to replace (instead of the object's parent). The sole parameter is a reference to a valid node object, which you can generate via the `document.createElement()` method or copy from an existing node. The value returned from the method is a reference to the

object that you replace. Thus, you can preserve a copy of the replaced node by storing the results in a variable for use later.

If the node you replace contains other nodes, the `replaceNode()` method removes all contained nodes of the original from the document. Therefore, if you want to change a wrapper node but want to maintain the original children, your script must capture the children and put them back into the new node as shown in the following example.

Example

Listing 26-31 demonstrates three node-related methods: `removeNode()`, `replaceNode()`, and `swapNode()`. These methods work in WinIE5+ only.

The page rendered from Listing 26-31 begins with a `ul` type list of four items. Four buttons control various aspects of the node structure of this list element. The first button invokes the `replace()` function, which changes the `ul` type to `ol`. To do this, the function must temporarily tuck away all child nodes of the original `ul` element so that they can be added back into the new `ol` element. At the same time, the old `ul` node is stored in a global variable (`oldNode`) for restoration in another function.

To replace the `ul` node with an `ol`, the `replace()` function creates a new, empty `ol` element and assigns the `myOL` ID to it. Next, the children (`li` elements) are stored en masse as an array in the variable innards. The child nodes are then inserted into the empty `ol` element, using the `insert-Before()` method. Notice that as each child element from the `innards` array is inserted into the `ol` element, the child element is removed from the `innards` array. That's why the loop to insert the child nodes is a `while` loop that constantly inserts the first item of the `innards` array to the new element. Finally, the `replaceNode()` method puts the new node in the old node's place, and the old node (just the `ul` element) is stored in `oldNode`.

The `restore()` function operates in the inverse direction of the `replace()` function. The same juggling of nested child nodes is required.

The third button invokes the `swap()` function, whose script exchanges the first and last nodes. The `swapNode()` method, like the others in this discussion, operates from the point of view of the node. Therefore, the method is attached to one of the swapped nodes, and the other node is specified as a parameter. Because of the nature of the `ol` element, the number sequence remains fixed, but the text of the `li` node swaps.

To demonstrate the `removeNode()` method, the fourth function removes the last child node of the list. Each call to `removeNode()` passes the `true` parameter to guarantee that the text nodes nested inside each `li` node are also removed. Experiment with this method by setting the parameter to `false` (the default). Notice how the parent-child relationship changes when you remove the `li` node.

LISTING 26-31

Using Node-Related Methods

HTML: jsb26-31.html

```
<!DOCTYPE html>
<html>
  <head>
```

continued

LISTING 26-31 *(continued)*

```
    <meta http-equiv="content-type" content="text/html;charset=utf-8">
    <title>removeNode(), replaceNode(), and swapNode() Methods</title>
    <script type="text/javascript" src="../jsb-global.js"></script>
    <script type="text/javascript" src="jsb26-31.js"></script>
  </head>
  <body>
   <h1>Node Methods</h1>
   <hr />
   Here is a list of items:
   <ul id="myUL">
     <li>First Item</li>
     <li>Second Item</li>
     <li>Third Item</li>
     <li>Fourth Item</li>
   </ul>
   <form>
    <input type="button" value="Change to OL List"
            onclick="replace()" />  
    <input type="button"
            value="Restore LI List" onclick="restore()" />  
    <input type="button" value="Swap First/Last" onclick="swap()" />  
    <input type="button" value="Remove Last" onclick="remove()" />
   </form>
  </body>
</html>
```

JavaScript: jsb26-31.js

```
// store original node between changes
var oldNode;

// replace UL node with OL
function replace()
{
   if (document.getElementById("myUL"))
   {
      var newNode = document.createElement("OL");
      newNode.id = "myOL";
      var innards = document.getElementById("myUL").children;
      while (innards.length > 0)
      {
         newNode.insertBefore(innards[0]);
      }
      oldNode = document.getElementById("myUL").replaceNode(newNode);
   }
}

// restore OL to UL
function restore()
{
```

```
    if (document.getElementById("myOL") && oldNode)
    {
       var innards = document.getElementById("myOL").children;
       while (innards.length > 0)
       {
          oldNode.insertBefore(innards[0]);
       }
       document.getElementById("myOL").replaceNode(oldNode);
    }
 }

 // swap first and last nodes
 function swap()
 {
    if (document.getElementById("myUL"))
    {
       document.getElementById("myUL").firstChild.swapNode(
          document.getElementById("myUL").lastChild);
    }
    if (document.getElementById("myOL"))
    {
       document.getElementById("myOL").firstChild.swapNode(
          document.getElementById("myOL").lastChild);
    }
 }

 // remove last node
 function remove()
 {
    if (document.getElementById("myUL"))
    {
       document.getElementById("myUL").lastChild.removeNode(true);
    }
    if (document.getElementById("myOL"))
    {
       document.getElementById("myOL").lastChild.removeNode(true);
    }
 }
```

You can accomplish the same functionality shown in Listing 26-31 in a cross-browser fashion using the W3C DOM. In place of the `removeNode()` and `replaceNode()` methods, use `removeChild()` and `replaceChild()` methods to shift the point of view (and object references) to the parent of the `ul` and `ol` objects: the `document.body`. Also, you need to change the `document.all` references to `document.getElementById()`.

Related Items: `removeChild()`, `removeNode()`, `replaceChild()`, `swapNode()` methods

scrollIntoView(*topAlignFlag*)

Returns: Nothing

Compatibility: WinIE4+, MacIE4+, NN7+, Moz+, Safari 2.02, Opera+, Chrome+

The `scrollIntoView()` method scrolls the page (vertically and/or horizontally as needed) such that the current object is visible within the window or frame that contains it. A single parameter, a Boolean value, controls the location of the element within the viewable space. A value of `true` (the default) causes the element to be displayed so that its top is aligned with the top of the window or frame (provided that the document beneath it is long enough to allow this amount of scrolling). But a value of `false` causes the bottom of the element to align with the bottom of the viewable area. In most cases, you want the former so that the beginning of a page section is at the top of the viewable area. But if you don't want a user to see content below a certain element when you jump to the new view, use the `false` parameter.

For form elements, you must use the typical form element reference (`document.formName.elementName.scrollIntoView()`) unless you also specify an ID attribute for the element (`document.getElementById("elementID").scrollIntoView()`).

Example

Use The Evaluator (see Chapter 4) to experiment with the `scrollIntoView()` method. Resize the browser window height so that you can see only the top text box and the Results text area. Enter each of the following statements in the top text box, and see where the `myP` element comes into view:

```
myP.scrollIntoView()
myP.scrollIntoView(false)
```

Expand the height of the browser window until you can see part of the table lower on the page. If you enter

```
myTable.scrollIntoView(false)
```

in the top text box, the page scrolls to bring the bottom of the table to the bottom of the window. But if you use the default parameter (`true` or empty)

```
myTable.scrollIntoView()
```

the page scrolls as far as it can in an effort to align the top of the element as closely as possible to the top of the window. The page cannot scroll beyond its normal scrolling maximum (although if the element is a positioned element, you can use dynamic positioning to place it wherever you want — including off the page). Also, if you shrink the window and try to scroll the top of the table to the top of the window, be aware that the `table` element contains a `caption` element, so the caption is flush with the top of the window.

Related Items: `window.scroll()`, `window.scrollBy()`, `window.scrollTo()` methods

setActive()

Returns: Nothing

Compatibility: WinIE5.5+, MacIE-, NN-, Moz-, Safari-, Opera-, Chrome-

The `setActive()` method lets a script designate an element object as the active element. However, unlike in the `focus()` method, the window does not scroll the active element into view. Any `onFocus` event handler defined for the element fires when `setActive()` is invoked without the browser's giving the element focus.

Example

Use The Evaluator (see Chapter 4) to compare the `setActive()` and `focus()` methods. With the page scrolled to the top and the window sized so that you cannot see the sample checkbox near the bottom of the page, enter the following statement in the top text box:

```
document.getElementById("myCheckbox").setActive()
```

Scroll down to see that the checkbox has operational focus (press the spacebar to see). Now scroll back to the top, and enter the following:

```
document.getElementById("myCheckbox").focus()
```

This time, the checkbox gets focus, and the page automatically scrolls the object into view.

Related Item: `focus()` method

setAttribute("*attributeName*", *value*[, *caseSensitivity*])

Returns: Nothing

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

The `setAttribute()` method assigns a new value to an existing attribute of the current object or inserts an entirely new attribute name–value pair among the attributes of the current object. This method represents an alternative syntax to setting a property of the object directly.

Note

The W3C DOM Level 2 standard recommends `getAttribute()` and `setAttribute()` for reading and writing element object attribute values, rather than reading and writing those values by way of their corresponding properties. Although using these methods is certainly advisable for XML elements, the same DOM standard sends conflicting signals by defining all kinds of properties for HTML element objects. Browsers, of course, will support access via properties well into the future, so don't feel obligated to change your ways just yet. ■

The first two parameters of `setAttribute()` are required. The first is the name of the attribute. The default behavior of this method respects the case of the attribute name. Therefore, if you use `setAttribute()` to adjust the value of an existing attribute in default mode, the first parameter must match the case of the attribute as known by the object model for the current document. Remember that all names of all attributes assigned as inline source-code attributes are automatically converted to lowercase letters.

A value you assign to the attribute is the second parameter. For cross-browser compatibility, the value should be either a string or Boolean data type.

IE provides an optional third parameter to control the case-sensitivity issue for the attribute name. The default value (`true`) has a different impact on your object depending on whether you use `setAttribute()` to assign a new attribute or reassign an existing one. In the former case, the third parameter as `true` means that the attribute name assigned to the object observes the case of the first parameter. In the latter case, the third parameter as `true` means that the attribute isn't reassigned unless the first parameter matches the case of the attribute currently associated with the object. Instead, a new attribute with a different case sequence is created.

Attempting to manage the case sensitivity of newly created attributes is fraught with peril, especially if you try to reuse names but with different case sequences. We strongly recommend using default case-sensitivity controls for `setAttribute()` and `getAttribute()`.

See also the W3C DOM facilities for treating attributes as node objects in the discussions of the `getAttributeNode()` and `removeAttributeNode()` methods earlier in this chapter.

Example

Use The Evaluator (see Chapter 4) to experiment with the `setAttribute()` method for the elements in the page. Setting attributes can have immediate impact on the layout of the page (just as setting an object's properties can). Enter the following sample statements in the top text box to view attribute values.

```
document.getElementById("myTable").setAttribute("width", "80%")
document.getElementById("myTable").setAttribute("border", "5")
```

Related Items: `attributes` property; `document.createAttribute()`, `getAttribute()`, `getAttributeNode()`, `removeAttribute()`, `removeAttributeNode()`, `setAttributeNode()` methods

`setAttributeNode()`

(See `removeAttributeNode()`)

`setAttributeNodeNS("`*`attributeNode`*`")`

Returns: `Attribute` object

Compatibility: WinIE-, MacIE-, NN6+, Moz+, Safari+, Opera+, Chrome+

This method inserts or replaces an attribute in the current element. The sole parameter is an attribute object, and the return value is a reference to the newly inserted attribute object. When the method is invoked, the browser looks for a pairing of local name and namespace URI between the nodes. If there is a match, the node replaces the matched node; otherwise, the node is inserted.

Related Items: `attributes`, `namespaceURI`, `localName` properties; `removeAttributeNS()`, `getAttributeNS()`, and `setAttributeNS()` methods

`setAttributeNS("`*`namespaceURI`*`", "`*`qualifiedName`*`", "`*`value`*`")`

Returns: Nothing

Compatibility: WinIE-, MacIE-, NN6+, Moz+, Safari+, Opera+, Chrome+

This method inserts or replaces an attribute in the current element, as specified in the three parameters. The first parameter of the method is a URI string matching a URI assigned to a label in the document. The second parameter is the local name portion of the attribute whose value you are getting. If a match is found among these parameters, the value in the third parameter is assigned to the existing attribute; otherwise, the value is inserted as a new attribute.

Related Items: `attributes`, `namespaceURI`, `localName` properties; `removeAttributeNS()`, `getAttributeNS()`, and `setAttributeNodeNS()` methods

setCapture(*containerBoolean*)

(See releaseCapture())

setExpression("*propertyName*", "*expression*",["*language*"])

Returns: Nothing

Compatibility: WinIE5+, MacIE-, NN-, Moz-, Safari-, Opera-, Chrome-

Use the setExpression() method to assign the result of an executable expression to the value of an element object property. This method can assign values to both HTML element objects and style objects that belong to them.

The setExpression() method is a scripted way of assigning expressions to attributes. But you can also assign expressions directly to style sheet definitions in the HTML tag of an element using the expression() syntax, as in the following example:

```
<p style="width:expression(document.body.style.width * 0.75)">
```

The setExpression() method requires three parameters. The first parameter is the name of the property (in string form) to which you assign the expression. Property names are case sensitive. The second parameter is a string form of the expression to be evaluated to supply a value for the property. Expressions can refer to global variables or properties of other objects in the same document (provided that the property is anything other than an array). An expression may also contain math operators.

Pay close attention to the data type of the evaluated value of the expression. The value must be a valid data type for the property. For example, the URL of the body background image must be a string. But for numeric values, you can generally use number and string types interchangeably because the values are converted to the proper type for the property. Even for expressions that evaluate to numbers, encase the expression inside quotes. It may not be necessary in all cases, but if you get into the habit of using quotes, you'll have fewer problems for strings or complex expressions that require them.

You are not limited to using JavaScript as the language for the expression because you can also specify the scripting language of the expression in the optional third parameter. Acceptable parameter values for the language are

```
JScript
JavaScript
VBScript
```

For all intents and purposes, JScript and JavaScript are the same. Both languages are ECMA-262 compatible. JScript is the default value for the language parameter.

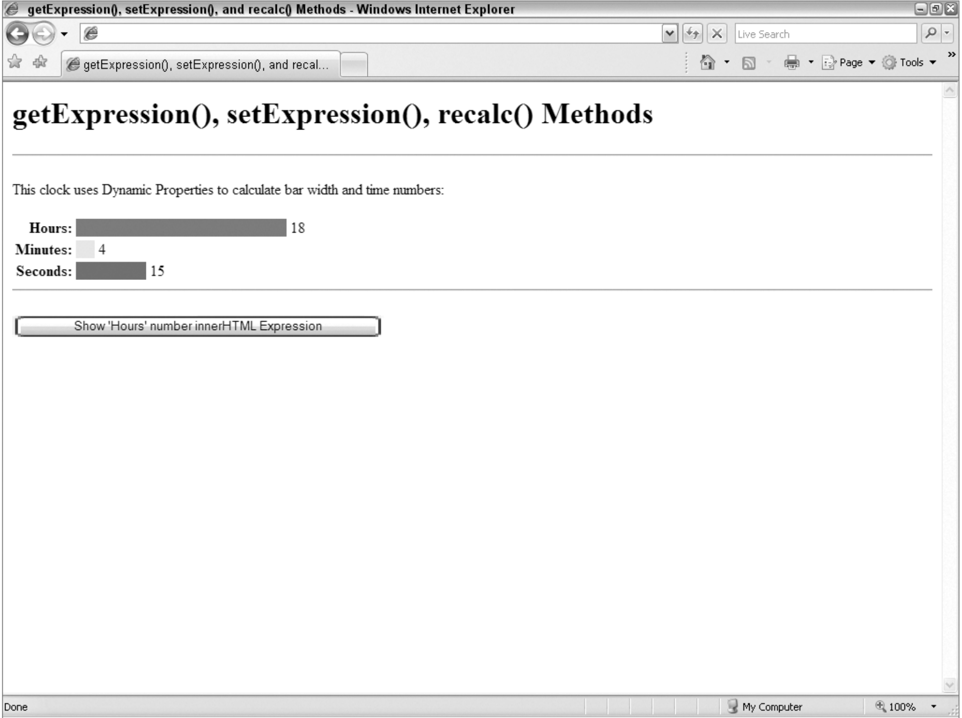
One reason to use setExpression() for dynamic properties is to let the property always respond to the current conditions on the page. For example, if you set a property that is dependent on the current width of the body, you want a recalculation that is applied to the property if the user resizes the window. The browser automatically responds to many events and updates any dynamic properties. In essence, the browser recalculates the expressions and applies the new values to the property. Keyboard events in particular trigger this kind of automatic recalculation for you. But if your scripts perform actions on their own (in other words, not triggered by events), your scripts need to force the recalculation of the expressions. The document.recalc() method takes care of this, but you must invoke it to force the recalculation of dynamic properties in these cases.

Example

Listing 26-32 shows the `setExpression()`, `recalc()`, and `getExpression()` methods at work in a DHTML-based clock. Figure 26-1 shows the clock. As time clicks by, the bars for hours, minutes, and seconds adjust their widths to reflect the current time. At the same time, the `innerHTML` of `span` elements to the right of each bar display the current numeric value for the bar.

FIGURE 26-1

A bar-graph clock created with dynamic expressions



The dynamically calculated values in this example are based on the creation of a new `Date` object over and over again to get the current time from the client computer clock. It is from the `Date` object (stored in the variable called `now`) that the hour, minute, and second values are retrieved. Some other calculations are involved so that a value for one of these time components is converted to a pixel value for the width of the bars. The bars are divided into 24 (for the hours) and 60 (for the minutes and seconds) parts, so the scale for the two types differs. For the 60-increment bars in this application, each increment is set to 5 pixels (stored in `shortWidth`); the 24-increment bars are 2.5 times the `shortWidth`.

As the document loads, the three `span` elements for the colored bars are given no width, which means that they assume the default width of zero. But after the page loads, the `onload` event handler invokes the `init()` function, which sets the initial values for each bar's width and the text (`innerHTML`) of the three labeled spans. After these initial values are set, the `init()` function invokes the `updateClock()` function.

In the `updateClock()` function, a new `Date` object is created for the current instant. The `document.recalc()` method is called, instructing the browser to recalculate the expressions that were set in the `init()` function and assign the new values to the properties. To keep the clock ticking, the `setTimeout()` method is set to invoke this same `updateClock()` function in 1 second.

To see what the `getExpression()` method does, you can click the button on the page. It simply displays the returned value for one of the attributes that you assign using `setExpression()`.

LISTING 26-32

Dynamic Properties

HTML: jsb26-32.html

```
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>getExpression(), setExpression(), and recalc() Methods</title>
      <style type="text/css">
         th
         {
            text-align:right;
         }
         span
         {
            vertical-align:bottom;
         }
      </style>
      <script type="text/javascript" src="../jsb-global.js"></script>
      <script type="text/javascript" src="jsb26-32.js"></script>
   </head>
   <body onload="init()">
      <h1>getExpression(), setExpression(), recalc() Methods</h1>
      <hr />
      <p>This clock uses Dynamic Properties to calculate bar width and time
         numbers:
      </p>
      <table border="0">
         <tr>
            <th>Hours:</th>
            <td>
               <span id="hoursBlock" style="background-color:red"></span>
                <span id="hoursLabel"></span>
            </td>
         </tr>
         <tr>
            <th>Minutes:</th>
            <td>
               <span id="minutesBlock" style="background-color:yellow"></span>
                <span id="minutesLabel"></span>
            </td>
```

continued

LISTING 26-32 *(continued)*

```
         </tr>
         <tr>
            <th>Seconds:</th>
            <td>
               <span id="secondsBlock" style="background-color:green"></span>
                <span id="secondsLabel"></span>
            </td>
         </tr>
      </table>
      <hr />
      <form>
         <input type="button" value="Show 'Hours' number innerHTML Expression"
                onclick="showExpr()" />
      </form>
   </body>
</html>
```

JavaScript: jsb26-32.js

```
var now = new Date();
var shortWidth = 5;
var multiple = 2.5;

function init()
{
   with (document.all)
   {
      hoursBlock.style.setExpression("width",
         "now.getHours() * shortWidth * multiple","jscript");
      hoursLabel.setExpression("innerHTML",
         "now.getHours()","jscript");
      minutesBlock.style.setExpression("width",
         "now.getMinutes() * shortWidth","jscript");
      minutesLabel.setExpression("innerHTML",
         "now.getMinutes()","jscript");
      secondsBlock.style.setExpression("width",
         "now.getSeconds() * shortWidth","jscript");
      secondsLabel.setExpression("innerHTML",
         "now.getSeconds()","jscript");
   }

   updateClock();
}

function updateClock()
{
   now = new Date();
   document.recalc();
   setTimeout("updateClock()",1000);
}
```

```
function showExpr()
{
   alert("Expression for the \'Hours\' innerHTML property is:\r\n" +
      document.getElementById("hoursLabel").getExpression("innerHTML") +
      ".");
}
```

Related Items: `document.recalc()`, `removeExpression()`, `setExpression()` methods

setUserData("*key*", *dataObj*, *dataHandler*)

Returns: Object

Compatibility: WinIE-, MacIE-, NN-, Moz1.7.2+, Safari+, Opera-, Chrome+

The `setUserData()` method is designed to allow for the addition of user data to a node. This user data comes in the form of an object and is associated with a node through a string key. By requiring a key for an object of user data, the `setUserData()` method allows you to set multiple pieces of data (objects) on a single node. The last parameter to the method is an event handler function reference that is called whenever the data object is cloned, imported, deleted, renamed, or adopted.

Although some support for the `setUserData()` method was added in Moz1.7.2, the method still isn't supported to the degree that you can actually use it, as of Moz1.8.1.

swapNode(*otherNodeObject*)

Returns: `Node` object reference

Compatibility: WinIE5+, MacIE-, NN-, Moz-, Safari-, Opera-, Chrome-

The `swapNode()` method exchanges the positions of two nodes within an element hierarchy. Contents of both nodes are preserved in their entirety during the exchange. The single parameter must be a valid node object (perhaps created with `document.createElement()` or copied from an existing node). A return value is a reference to the object whose `swapNode()` method was invoked.

Example

See Listing 26-31 (the `replaceNode()` method) for an example of the `swapNode()` method in action.

Related Items: `removeChild()`, `removeNode()`, `replaceChild()`, `replaceNode()` methods

tags("*tagName*")

Returns: Array of element objects

Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari+, Opera+, Chrome+

The `tags()` method does not belong to every element, but it is a method of every collection of objects (such as `all`, `forms`, and `elements`). The method is best thought of as a kind of filter for the elements that belong to the current collection. For example, to get an array of all `p` elements inside a document, use this expression:

```
document.all.tags("P")
```

You must pass a parameter string consisting of the tag name you wish to extract from the collection. The tag name is case insensitive.

The return value is an array of references to the objects within the current collection whose tags match the parameter. If there are no matches, the returned array has a length of zero. If you need cross-browser compatibility, use the `getElementsByTagName()` method described earlier in this chapter, and pass a wildcard value of `"*"`.

Example

Use The Evaluator (see Chapter 4) to experiment with the `tags()` method. Enter the following statements one at a time in the top text box, and study the results:

```
document.all.tags("div")
document.all.tags("div").length
myTable.all.tags("td").length
```

Because the `tags()` method returns an array of objects, you can use one of those returned values as a valid element reference:

```
document.all.tags("form")[1].elements.tags("input").length
```

The browsers that support tags do not all support it for the same HTML elements. For example, WebKit-based browsers do not support `tags` as part of `table`.

Related Item: `getElementsByTagName()` method

`toString("`*`param`*`")`

Returns: String

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

The `toString()` method returns a string representation of the element object, which unfortunately can mean different things to different browsers. Don't expect entirely consistent results across browsers, especially when you consider that IE simply returns a generic `"[object]"` string.

`urns("`*`behaviorURN`*`")`

Returns: Array of element objects

Compatibility: WinIE5+, MacIE-, NN-, Moz-, Safari-, opera-, Chrome-

The `urns()` method does not belong to every element, but it is a method of every collection of objects. You must pass a parameter string consisting of the URN (Uniform Resource Name) of a behavior resource (most typically `.htc`) assigned to one or more elements of the collection. The parameter does not include the extension of the filename. If there is no matching behavior URN for the specified parameter, the `urns()` method returns an array of zero length. This method is related to the `behaviorUrns` property, which contains an array of behavior URNs assigned to a single element object.

Example

In case the `urns()` method is reconnected in the future, you can add a button and function to Listing 26-19b that reveals whether the `makeHot.htc` behavior is attached to the `myP` element. Such a function looks like this:

```
function behaviorAttached()
{
   if (document.all.urns("makeHot"))
   {
      alert("There is at least one element set to \'makeHot\'.");
   }
}
```

Related Item: `behaviorUrns` property

Event handlers

onactivate
ondeactivate

Compatibility: WinIE5.5+, MacIE-, NN-, Moz-, Safari-, Opera-, Chrome-

The `onactivate` and `ondeactivate` event handlers are similar to the `onfocus` and `onblur` event handlers, respectively, as well as to the IE5.5+ `onfocusin` and `onfocusout` events. Starting with IE5.5+, it is possible to manage the activation of an element and the focus of an element separately. The `onactivate` and `ondeactivate` events correspond to the activation of an element, whereas `onfocusin` and `onfocusout` deal with focus. In many cases, activation and focus go hand in hand, but not always.

If an element receives activation, the `onactivate` event fires for that element just before the activation takes hold; conversely, just before the element loses activation, events fire in the sequence `onbeforedeactivate`, `ondeactivate`, `onblur`. Only elements that by their nature can accept activation (for example, links and form input controls) or that have a `tabindex` attribute set can become the active element (and, therefore, fire these events).

WinIE5.5+ maintains the original `onfocus` and `onblur` event handlers. But because the behaviors are so close to those of the `onactivate` and `ondeactivate` events, we don't recommend mixing the old and new event handler names in your coding style. If you script exclusively for WinIE5.5+, which is unlikely in this day and age, you can use the newer terminology throughout. And if you truly want to track the focus of an element in IE, consider using `onfocusin` and `onfocusout` instead.

Example

You can modify Listing 26-34 later in this chapter by substituting `onactivate` for `onfocus` and `ondeactivate` for `onblur`.

Use The Evaluator (see Chapter 4) to experiment with the `onbeforedeactivate` event handler. To begin, set the `myP` element so it can accept focus:

```
myP.tabIndex = 1
```

If you repeatedly press the Tab key, the `myP` paragraph will eventually receive focus — indicated by the dotted rectangle around it. To see how you can prevent the element from losing focus, assign an anonymous function to the `onbeforedeactivate` event handler, as shown in the following statement:

```
myP.onbeforedeactivate = new Function("event.returnValue=false")
```

Now you can press Tab all you like or click other focusable elements all you like, and the `myP` element will not lose focus until you reload the page (which clears away the event handler). Please do not do this on your pages unless you want to infuriate and alienate your site visitors.

Related Items: `onblur`, `onfocus`, `onfocusin`, `onfocusout` event handlers

onafterupdate
onbeforeupdate

Compatibility: WinIE4+, MacIE5+, NN-, Moz-, Safari-, Opera-, Chrome-

The `onafterupdate` and `onbeforeupdate` event handlers fire on a bound data object in IE whenever the data in the object is being updated. The `onbeforeupdate` event is fired just before the update occurs, whereas `onafterupdate` is fired after the data has been successfully updated.

onbeforecopy

Compatibility: WinIE5+, MacIE-, NN-, Moz-, Safari1.3+, Opera-, Chrome+

The `onbeforecopy` event handler fires before the actual copy action takes place whenever the user initiates a content copy action via the Edit menu (including the Ctrl+C, for Windows, and Command+C, for Mac, keyboard shortcuts) or the right-click context menu. In all browsers, if the user copies with a keyboard shortcut, the copy operation will be completed. If the user accesses the Copy command via the Edit or context menu different browsers behave differently. In WebKit-based browsers, the menu displays, the user selects copy and then the `onbeforecopy` event fires. The `onbeforecopy` event fires before either menu can display in IE. In practice, the event may fire twice even though you expect it only once. Just because the `onbeforecopy` event fires, it does not guarantee that a user will complete the copy operation (for example, the context menu may close before the user makes a selection).

Unlike paste-related events, the `onbeforecopy` event handler does not work with form input elements. Just about any other HTML element is fair game, however.

Example

In Listing 26-33, the function invoked by the "Latin" paragraph element's `onbeforecopy` event handler merely issues an alert. If your audience is limited in their browser use, you could use the `onbeforecopy` event handler to preprocess information prior to an actual copy action.

LISTING 26-33

The onbeforecopy Event Handler

HTML: jsb26-33.html

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>onbeforecopy event handler</title>
      <style type="text/css">
         .keyword
```

```
        {
           color: blue; font-weight:bold;
        }
     </style>
     <script type="text/javascript" src="../jsb-global.js"></script>
     <script type="text/javascript" src="jsb26-33.js"></script>
   </head>
   <body>
     <h1>onbeforecopy event handler</h1>
     <hr />
     <p>Select one or more characters in the Latin paragraph.
        then execute a copy command:
        <ul>
           <li>edit menu</li>
           <li>context menu</li>
           <li>ctrl-c (on Windows)</li>
           <li>command-c (on Mac)</li>
        </ul>
        Browsers that support <span class="keyword">onbeforecopy</span>,
        do not behave consistently. You may not be able to use all
        of the copy commands normally available to you. Test all the
        different ways to copy text and see what happens.
     </p>
     <p>
        Browsers that do not support <span class="keyword">onbeforecopy</span>
        will still allow you to copy -- you just won't see an alert.
     </p>
     <p id="myp" onbeforecopy="beforeCopy()">lorem ipsum dolor sit amet,
        consectetaur adipisicing elit, sed do eiusmod tempor incididunt ut
        labore et dolore magna aliqua. ut enim adminim veniam, quis nostrud
        exercitation ullamco laboris nisi ut aliquip ex commodo consequat.</p>
     <form>
        <p>paste results here:<br />
           <textarea name="output" cols="60" rows="5"></textarea>
        </p>
     </form>
   </body>
</html>
```

JavaScript: jsb26-33.js

```
function beforeCopy()
{
   alert("onbeforecopy is supported by your browser");
}
```

Related Items: onbeforecut, oncopy event handlers

onbeforecut

Compatibility: WinIE5+, MacIE-, NN-, Moz-, Safari 1.3+, Opera-, Chrome+

The `onbeforecut` event handler fires before the actual cut action takes place whenever the user initiates a content cut via the Edit menu (including the Ctrl+X keyboard shortcut) or the right-click context menu. If you add the `onbeforecut` event handler to an HTML element, the context menu usually disables the Cut menu item. But assigning a JavaScript call to this event handler brings the Cut menu item to life. If the user accesses the Cut command via the Edit or context menu, different browsers behave differently. In WebKit-based browsers, the menu displays, the user selects cut and then the `onbeforecut` event fires. In IE the `onbeforecut` event fires before either menu displays. In practice, the event may fire twice even though you expect it only once. Just because the `onbeforecut` event fires, it does not guarantee that a user will complete the cut operation (for example, the context menu may close before the user makes a selection).

Example

You can use the `onbeforecut` event handler to preprocess information prior to an actual cut action. You can try this by editing a copy of Listing 26-33, changing the `onbeforecopy` event handler to `onbeforecut`.

Related Items: `onbeforecopy`, `oncut` event handlers

onbeforedeactivate

Compatibility: WinIE5.5+, MacIE-, NN-, Moz-, Safari-

(See `onactivate` event handler)

onbeforeeditfocus

Compatibility: WinIE5+, MacIE-, NN-, Moz-, Safari-, Opera-, Chrome-

The `onbeforeeditfocus` event handler is triggered whenever you edit an element on a page in an environment such as Microsoft's DHTML Editing ActiveX control or with the editable page content feature of IE5.5+. This discussion focuses on the latter scenario because it is entirely within the scope of client-side JavaScript. The `onbeforeeditfocus` event fires just before the element receives its focus. (There may be no onscreen feedback that editing is turned on unless you script it yourself.) The event fires each time a user clicks the element, even if the element just received edit focus elsewhere in the same element.

Example

Use The Evaluator (see Chapter 4) to explore the `onbeforeeditfocus` in WinIE5.5+. In the following sequence, you assign an anonymous function to the `onbeforeeditfocus` event handler of the `myP` element. The function turns the text color of the element to red when the event handler fires:

```
document.getElementById("myP").onbeforeeditfocus = new
Function("document.getElementById('myP').style.color='red'")
```

Now turn on content editing for the `myP` element:

```
document.getElementById("myP").contentEditable = true
```

Now if you click inside the `myP` element on the page to edit its content, the text turns red before you begin editing. In a page scripted for this kind of user interface, you would include some control that turns off editing and changes the color to normal.

Related Items: `document.designMode`, `contentEditable`, `isContentEditable` properties

onbeforepaste

Compatibility: WinIE5+, MacIE-, NN-, Moz-, Safari 1.3+, Opera-, Chrome+

Like `onbeforecopy` and `onbeforecut`, the `onbeforepaste` event occurs differently in different browsers. In WebKit-based browsers, the menu displays, the user selects paste and then the `onbeforepaste` event fires. In IE, the `onbeforepaste` event fires just prior to the display of either the context or menu-bar Edit menu when the current object is selected (or has a selection within it). The primary value of this event comes when you use scripts to control the copy-and-paste process of a complex object. Such an object may have multiple kinds of data associated with it, but your script captures only one of the data types. Or you may want to put some related data about the copied item (for example, the `id` property of the element) into the clipboard. By using the `onbeforepaste` event handler to set the `event.returnValue` property to `false`, you guarantee that the pasted item is enabled in the context or Edit menu (provided that the clipboard is holding some content). A handler invoked by `onpaste` should then apply the specific data subset from the clipboard to the currently selected item.

Example

See Listing 26-44 for the `onpaste` event handler (later in this chapter) to see how the `onbeforepaste` and `onpaste` event handlers work together.

Related Items: `oncopy`, `oncut`, `onpaste` event handlers

onbeforeupdate

(See `onafterupdate`)

onblur

Compatibility: WinIE3+, MacIE3+, NN2+, Moz+, Safari+, Opera+, Chrome+

The `onblur` event fires when an element that has focus is about to lose focus because some other element is about to receive focus. For example, a text input element fires the `onblur` event when a user tabs from that element to the next one inside a form. The `onblur` event of the first element fires before the `onfocus` event of the next element.

The availability of the `onblur` event has expanded with succeeding generations of script-capable browsers. In the earlier versions, blur and focus were largely confined to text-oriented input elements (including the `select` element). These are safe to use with all scriptable browser versions. The `window` object received the `onblur` event handler starting with NN3 and IE4. IE4 also extended the event handler to more form elements, predominantly on the Windows operating system because that OS has a user interface clue (the dotted rectangle) when items such as buttons and links receive focus (so that you may act upon them by pressing the spacebar). For IE5+, the `onblur` event handler is available to virtually every HTML element. For most of those elements, however, blur and focus are not possible unless you assign a value to the `tabindex` attribute of the element's tag. For example, if you assign `tabindex="1"` inside a `<p>` tag, the user can bring focus to that paragraph (highlighted with the dotted rectangle in Windows in some of the browsers) by clicking the paragraph or pressing the Tab key until that item receives focus in sequence.

If you plan to use the `onblur` event handler on window or text-oriented input elements, be aware that there might be some unexpected and undesirable consequences of scripting for the event. For

example, in IE, a `window` object that has focus loses focus (and triggers the `onblur` event) if the user brings focus to any element on the page (or even clicks a blank area on the page). Similarly, the interaction between `onblur`, `onfocus`, and the `alert()` dialog box can be problematic with text input elements. This is why we generally recommend using the `onchange` event handler to trigger form validation routines. If you should employ both the `onblur` and `onchange` event handler for the same element, the `onchange` event fires before `onblur`. For more details about using this event handler for data validation, see Chapter 46 on the CD-ROM.

WinIE5.5+ added the `ondeactivate` event handler, which fires immediately before the `onblur` event handler. Both the `onblur` and `ondeactivate` events can be blocked if the `onbeforedeactivate` event handler function sets `event.returnValue` to `false`.

Example

More often than not, a page author uses the `onblur` event handler to exert extreme control over the user, such as preventing a user from exiting a text box unless that user types something in the box. This is not a web-friendly practice, and it is one that we discourage because there are intelligent ways to ensure that a field has something typed into it before a form is submitted (see Chapter 46 on the CD-ROM). Listing 26-34 simply demonstrates the impact of the `tabindex` attribute with respect to the `onblur` and `onfocus` events. Notice that as you press the Tab key, only the second paragraph issues the events, even though all three paragraphs have event handlers assigned to them.

LISTING 26-34

onblur and onfocus Event Handlers

HTML: jsb26-34.html

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>onblur and onfocus Event Handlers</title>
      <script type="text/javascript" src="../jsb-global.js"></script>
      <script type="text/javascript" src="jsb26-34.js"></script>
   </head>
   <body>
      <h1 id="H1" tabindex="2">onblur and onfocus Event Handlers</h1>
      <h2>Start tabbing and see what happens</h2>
      <hr />
      <p id="P1" onblur="showBlur()" onfocus="showFocus()">Lorem ipsum dolor
         sit amet, consectetaur adipisicing elit, sed do eiusmod tempor
         incididunt ut labore et dolore magna aliqua. Ut enim adminim veniam,
         quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea
         commodo consequat.
      </p>
      <p id="P2" tabindex="1" onblur="showBlur()" onfocus="showFocus()">Bis
         nostrud exercitation ullam mmodo consequet. Duis aute involuptate
         velit esse cillum dolore eu fugiat nulla pariatur. At vver eos et
         accusam dignissum qui blandit est praesent luptatum delenit
         aigueexcepteur sint occae.
      </p>
      <p id="P3" onblur="showBlur()" onfocus="showFocus()">Unte af phen
```

```
         neigepheings atoot Prexs eis phat eit sakem eit vory gast te Plok
         peish ba useing phen roxas. Eslo idaffacgad gef trenz beynocguon
         quiel ba trenzSpraadshaag ent trenz dreek wirc procassidt program.
      </p>
   </body>
</html>
```

JavaScript: jsb26-34.js

```
function showBlur()
{
   var id = event.srcElement.id;
   alert("Element \"" + id + "\" has blurred.");
}

function showFocus()
{
   var id = event.srcElement.id;
   alert("Element \"" + id + "\" has received focus.");
}
```

Related Items: `blur()`, `focus()` methods; `ondeactivate`, `onbeforedeactivate`, `onfocus`, `onactivate` event handlers

oncellchange

Compatibility: WinIE5+, MacIE-, NN-, Moz-, Safari-, Opera-, Chrome-

The `oncellchange` event handler is part of the data binding of IE and fires when data changes in the data provider, which is usually a bound control. When responding to this event, you can analyze the `dataFld` property to find out which field in the recordset has changed.

onclick

Compatibility: WinIE3+, MacIE3+, NN2+, Moz+, Safari+, Opera+, Chrome+

The `onclick` event fires when a user presses down (with the primary mouse button) and releases the button with the pointer atop the element (both the down and up strokes must be within the rectangle of the same element). The event also fires with non-mouse-click equivalents in operating systems such as Windows. For example, you can use the keyboard to give focus to a clickable object and then press the spacebar or Enter key to perform the same action as clicking the element. In IE, if the element object supports the `click()` method, the `onclick` event fires with the invocation of that method (notice that this does not apply to Navigator or other browsers).

The `onclick` event is closely related to other mouse events. The other related events are `onmousedown`, `onmouseup`, and `ondblclick`. The `onmousedown` event fires when the user makes contact with the mouse switch on the downstroke of a click action. Next comes the `onmouseup` event (when the contact breaks). Only then does the `onclick` event fire — provided that the `onmousedown` and `onmouseup` events have fired in the same object. See the discussions on the `onmousedown` and `onmouseup` events later in this chapter for examples of their usage.

Interaction with the ondblclick event is simple: The onclick event fires (after the first click), followed by the ondblclick event (after the second click). See the discussion of the ondblclick event handler later in this chapter for more about the interaction of these two event handlers.

When used with objects that have intrinsic actions when users click them (namely, links and areas), the onclick event handler can perform all of the actions — including navigating to the destination normally assigned to the href attribute of the element. For example, to be compatible with all scriptable browsers, you can make an image clickable if you surround its tag with an <a> link tag. This lets the onclick event of that tag substitute for the missing onclick event handler of earlier <img> tags. If you assign an onclick event handler without special protection, the event handler will execute, and the intrinsic action of the element will be carried out. Therefore, you need to block the intrinsic action. To accomplish this, the event handler must evaluate to the statement return false. You can do this in two ways. The first is to append a return false statement to the script statement assigned to the event handler:

```
<a href="#" onclick="yourFunction(); return false"><img...></a>
```

As an alternative, you can let the function invoked by the event handler supply the false part of the return false statement, as shown in the following sequence:

```
function yourFunction()
{
   // [statements that do something here]
   return false;
}
...
<a href="#" onclick="return yourFunction()"><img...></a>
```

Either methodology is acceptable. A third option is to not use the onclick event handler at all but assign a javascript: pseudo-URL to the href attribute (see the Link object in Chapter 30).

The event model in IE4+ provides one more way to prevent the intrinsic action of an object from firing when a user clicks it. If the onclick event handler function sets the returnValue property of the event object to false, the intrinsic action is canceled. Simply include the following statement in the function invoked by the event handler:

```
event.returnValue = false;
```

The event model of the W3C DOM has a different approach to canceling the default action. In the event handler function for an event, invoke the eventObj.cancelDefault() method.

A common mistake made by scripting beginners is to use a submit type input button as a button intended to perform some script action rather than submitting a form. The typical scenario is an input element of type submit assigned an onclick event handler to perform some local action. The submit input button has an intrinsic behavior, just like links and areas. Although you can block the intrinsic behavior, as just described, you should use an input element of type button.

If you are experiencing difficulty with an implementation of the onclick event handler (such as trying to find out which mouse button was used for the click), it may be that the operating system or default browser behavior is getting in the way of your scripting. But you can usually get what you need via the onmousedown event handler. (The onmouseup event may not fire when you use the secondary mouse button to click an object.) Use the onclick event handler whenever possible to capture user clicks, because this event behaves most like users are accustomed to in their daily computing work. But fall back on onmousedown in an emergency.

Example

The `onclick` event handler is one of the simplest to grasp and use. Listing 26-35 demonstrates its interaction with the `ondblclick` event handler and shows you how to prevent a link's intrinsic action from activating when combined with `click` events. As you click and/or double-click the link, the text in the `span` element displays a message associated with each event. Notice that if you double-click, the `click` event fires first, with the first message immediately replaced by the second. For demonstration purposes, we show both backward-compatible ways of canceling the link's intrinsic action. In practice, decide on one style and stick with it.

LISTING 26-35

Using onclick and ondblclick Event Handlers

HTML: jsb26-35.html

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>onclick and ondblclick Event Handlers</title>
      <script type="text/javascript" src="../jsb-global.js"></script>
      <script type="text/javascript" src="jsb26-35.js"></script>
   </head>
   <body>
      <h1>onclick and ondblclick Event Handlers</h1>
      <hr />
      <a href="#" onclick="showClick();return false"
         ondblclick="return showDblClick()">A sample link.</a>
      (Click type: <span id="clickType"></span>)
   </body>
</html>
```

JavaScript: jsb26-35.js

```
var timeout;
function clearOutput()
{
   document.getElementById("clickType").innerHTML = "";
}
function showClick()
{
   document.getElementById("clickType").innerHTML = "single";
   clearTimeout(timeout);
   timeout = setTimeout("clearOutput()", 3000);
}
function showDblClick()
{
   document.getElementById("clickType").innerHTML = "double";
   clearTimeout(timeout);
   timeout = setTimeout("clearOutput()", 3000);
   return false;
}
```

Related Items: `click()` method; `oncontextmenu`, `ondblclick`, `onmousedown`, `onmouseup` event handlers

oncontextmenu

Compatibility: WinIE5+, MacIE-, NN7+, Moz+, Safari+, Opera-, Chrome+

The `oncontextmenu` event fires when the user clicks an object with the secondary (usually the right) mouse button. The only click-related events that fire with the secondary button are `onmousedown` and `oncontextmenu`.

To block the intrinsic application menu display of the `oncontextmenu` event, use any of the event cancellation methodologies available in for your browser (as just described in the `onclick` event handler description: two variations of evaluating the event handler to `return false`; assigning `false` to the `event.returnValue` property). It is not uncommon to wish to block the context menu from appearing so that users are somewhat inhibited from downloading copies of images or viewing the source code of a frame. Be aware, however, that if a user turns Active Scripting off in WinIE5+, the event handler cannot prevent the context menu from appearing.

Another possibility for this event is to trigger the display of a custom context menu constructed with other DHTML facilities. In this case, you must also disable the intrinsic context menu so that both menus do not display at the same time.

Example

See Listing 26-30 earlier in this chapter for an example of using the `oncontextmenu` event handler with a custom context menu.

Related Items: `releaseCapture()`, `setCapture()` methods

oncontrolselect

Compatibility: WinIE5.5+, MacIE-, NN-, Moz-, Safari-, Opera-, Chrome-

The `oncontrolselect` event fires just before a user makes a selection on an editable element while the page is in edit mode. It's important to note that it is the element itself that is selected to trigger this event, not the content within the element.

Related Items: `onresizeend`, `onresizestart` event handlers

oncopy
oncut

Compatibility: WinIE5+, MacIE4+, NN-, Moz+, Safari1.3+, Opera-, Chrome+

The `oncopy` and `oncut` events fire immediately after the user or script initiates a copy or cut edit action on the current object. Each event is preceded by its associated before event, which fires before any Edit or context menu appears (or before the copy or cut action, if initiated by keyboard shortcut).

Use these event handlers to provide edit functionality to elements that don't normally allow copying or cutting. In such circumstances, you need to enable the Copy or Cut menu items in the context or Edit menu by setting the `event.returnValue` for the `onbeforecopy` or `onbeforecut` event handlers to `false`. Then your `oncopy` or `oncut` event handlers must manually stuff a value into the clipboard by way of the `setdata()` method of the IE `clipboardData` object. If you use the `setdata()` method in your `oncopy` or `oncut` event handler, you must also set the

`event.returnValue` property to `false` in the handler function to prevent the default copy or cut action from wiping out your clipboard contents.

Because you are in charge of what data is stored in the clipboard, you are not limited to a direct copy of the data. For example, you might wish to store the value of the `src` property of an image object so that the user can paste it elsewhere on the page.

In the case of the `oncut` event handler, your script is also responsible for cutting the element or selected content from the page. To eliminate all of the content of an element, you can set the element's `innerHTML` or `innerText` property to an empty string. For a selection, use the `selection.createRange()` method to generate a `TextRange` object whose contents you can manipulate through the `TextRange` object's methods.

Example

Listing 26-36 shows both the `onbeforecut` and `oncut` event handlers in action (as well as `onbeforepaste` and `onpaste`). Notice that the `handleCut()` function not only stuffs the selected word into the IE `clipboardData` object, but also erases the selected text from the table cell element from where it came. If you replace the `onbeforecut` and `oncut` event handlers with `onbeforecopy` and `oncopy` (and change `handleCut()` to not eliminate the inner text of the event source element), the operation works with copy and paste instead of cut and paste. We demonstrate this later in the chapter in Listing 26-44.

LISTING 26-36

Cutting and Pasting under Script Control

HTML: jsb26-36.html

```
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>onbeforecut and oncut Event Handlers Using a range object</title>
      <style type="text/css">
         td
         {  text-align:center;
         }
         th
         {  text-decoration:underline;
         }
         .blanks
         {  text-decoration:underline;
         }
      </style>
      <script type="text/javascript" src="../jsb-global.js"></script>
      <script type="text/javascript" src="jsb26-36.js"></script>
   </head>
   <body>
      <h1>onbeforecut and oncut Event Handlers</h1>
      <hr />
      <p>Your goal is to cut and paste one noun and one adjective from the
         following table into the blanks of the sentence. Select a word from
```

continued

LISTING 26-36 *(continued)*

```
         the table and use the Edit or context menu to cut it from the table.
         Select one or more spaces of the blanks in the sentence and choose
         Paste to replace the blank with the clipboard contents.
      </p>
      <table cellpadding="5" onbeforecut="selectWhole()" oncut="handleCut()">
         <tr>
            <th>Nouns</th>
            <th>Adjectives</th>
         </tr>
         <tr>
            <td>truck</td>
            <td>round</td>
         </tr>
         <tr>
            <td>doll</td>
            <td>red</td>
         </tr>
         <tr>
            <td>ball</td>
            <td>pretty</td>
         </tr>
      </table>
      <p id="myP" onbeforepaste="handleBeforePaste()" onpaste="handlePaste()">
         Pat said, "Oh my, the <span id="blank1"
         class="blanks">     </span> is so <span
         id="blank2" class="blanks">     </span> !"
      </p>
      <button onclick="location.reload()">Reset</button>
   </body>
</html>
```

JavaScript: jsb26-36.js

```
function selectWhole()
{
   var obj = window.event.srcElement;
   var range = document.body.createTextRange();
   range.moveToElementText(obj);
   range.select();
   event.returnValue = false;
}

function handleCut()
{
   var rng = document.selection.createRange();
   clipboardData.setData("Text",rng.text);
   var elem = event.srcElement;
   elem.innerText = "";
   event.returnValue = false;
}
```

```
function handlePaste()
{
   var elem = window.event.srcElement;
      if (elem.className == "blanks")
      {
         elem.innerHTML = clipboardData.getData("Text");
      }
   event.returnValue = false;
}

function handleBeforePaste()
{
   var elem = window.event.srcElement;
      if (elem.className == "blanks")
      {
         event.returnValue = false;
      }
}
```

Related Items: `onbeforecopy`, `onbeforecut`, `onbeforepaste`, and `onpaste` event handlers

ondataavailable
ondatasetchanged
ondatasetcomplete

Compatibility: WinIE4+, MacIE-, NN-, Moz-, Safari-, Opera-, Chrome-

These three events are part of the data binding of IE and are fired to help reflect the state of data that is being transmitted. The `ondataavailable` event fires when data is transmitted from the data source, whereas the `ondatasetcomplete` event indicates that the recordset has completely downloaded from the data source. The `ondatasetchanged` event is fired when the recordset of a data source has somehow changed.

ondblclick

Compatibility: WinIE4+, MacIE4+, NN4+, Moz+, Safari+, Opera+, Chrome+

The `ondblclick` event fires after the second click of a double-click sequence. The timing between clicks depends on the client's mouse control panel settings. The `onclick` event also fires, but only after the first of the two clicks.

In general, it is rarely a good design to have an element perform one task when the mouse is single-clicked and a different task if double-clicked. With the event sequence employed in modern browsers, this isn't practical anyway (the `onclick` event always fires, even when the user double-clicks). But it is not uncommon to have the mouse down action perform some helper action. You see this in most icon-based file systems: If you click a file icon, it is highlighted at mouse down to select the item; you can double-click the item to launch it. In either case, one event's action does not impede the other nor confuse the user.

Example

See Listing 26-35 (for the `onclick` event handler) to see the `ondblclick` event in action.

Related Items: `onclick`, `onmousedown`, `onmouseup` event handlers

`ondrag, ondragend, ondragstart`

Compatibility: WinIE5+, MacIE-, NN-, Moz+, Safari 1.3+, Opera-, Chrome+

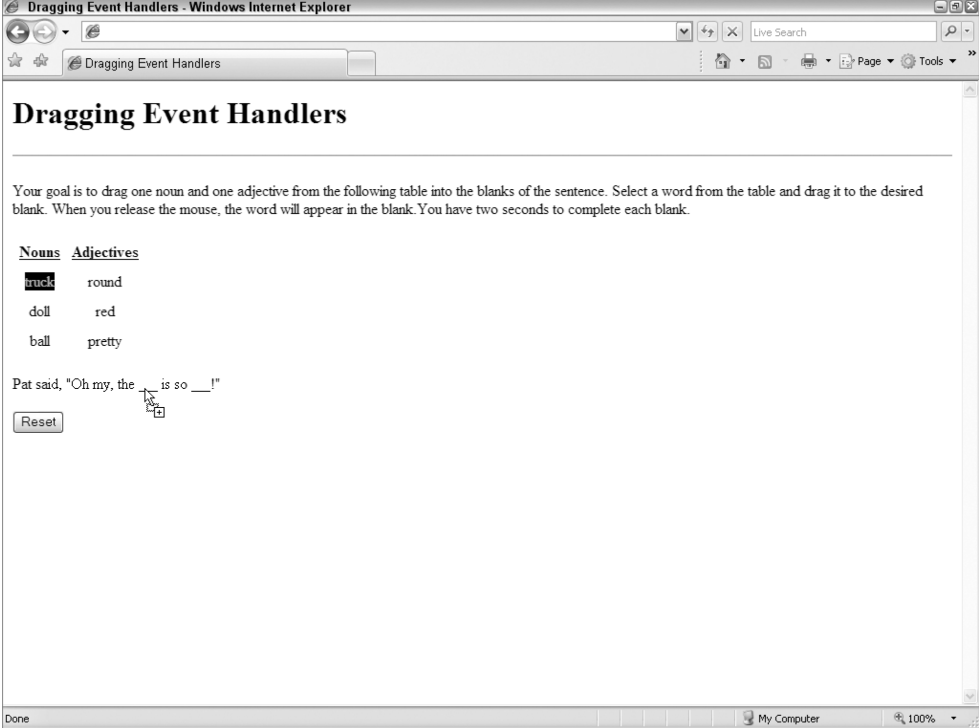
The `ondrag` event fires after the `ondragstart` event and continues firing repeatedly while the user drags a selection or object onscreen. Unlike the `onmousemove` event, which fires only as the cursor moves onscreen, the `ondrag` event continues to fire even when the cursor is stationary. In most browsers, users can drag objects to other browser windows or other applications. The event fires while the dragging extends beyond the browser window.

Because the event fires regardless of what is underneath the dragged object, you can use it in a game or training environment in which the user has only a fixed amount of time to complete a dragging operation (for example, matching similar pairs of objects). If the browser accommodates downloadable cursors, the `ondrag` event could cycle the cursor through a series of cursor versions to resemble an animated cursor.

Understanding the sequence of drag-related events during a user drag operation can be helpful if your scripts need to micromanage the actions (usually not necessary for basic drag-and-drop operations). Consider the drag-and-drop operation shown in Figure 26-2.

FIGURE 26-2

A typical drag-and-drop operation



It helps to imagine that the cells of the table with draggable content are named like spreadsheet cells: truck is cell A1; round is B1; doll is A2; and so on. During the drag operation, many objects are the targets of a variety of drag-related events. Table 26-11 lists the event sequence and the event targets.

In practice, some of the events shown in Table 26-11 may not fire. Much has to do with how many event handlers you trap that need to execute scripts along the way. The other major factor is the physical speed at which the user performs the drag-and-drop operation (which interacts with the CPU processing speed). The kinds of events that are most likely to be skipped are the `ondragenter` and `ondragleave` events, and perhaps some `ondragover` events if the user flies over an object before its `ondragover` event has a chance to fire.

Despite this uncertainty about drag-related event reliability, you can count on several important ones to fire all the time. The `ondragstart`, `ondrop` (if over a drop target), and `ondragend` events — as well some interstitial `ondrag` events — will definitely fire in the course of dragging onscreen. All but `ondrop` direct their events to the source element, whereas `ondrop` fires on the target.

Example

Listing 26-37 shows several drag-related event handlers in action. The page resembles the example in Listing 26-36, but the scripting behind the page is quite different. In this example, the user is encouraged to select individual words from the Nouns and Adjectives columns and drag them to the blanks of the sentence. To beef up the demonstration, Listing 26-37 shows you how to pass the equivalent of array data from a drag source to a drag target. At the same time, the user has a fixed amount of time (2 seconds) to complete each drag operation.

The `ondragstart` and `ondrag` event handlers are placed in the `<body>` tag because those events bubble up from any element that the user tries to drag. The scripts invoked by these event handlers filter the events so that the desired action is triggered only by the hot elements inside the table. This approach to event handlers prevents you from having to duplicate event handlers for each table cell.

The `ondragstart` event handler invokes `setupDrag()`. This function cancels the `ondragstart` event except when the target element (the one about to be dragged) is one of the `td` elements inside the table. To make this application smarter about what kind of word is dragged to which blank, it passes not only the word's text, but also some extra information about the word. This lets another event handler verify that a noun has been dragged to the first blank, whereas an adjective has been dragged to the second blank. To help with this effort, class names are assigned to the `td` elements to distinguish the words from the Nouns column from the words of the Adjectives column. The `setupDrag()` function generates an array consisting of the `innerText` of the event's source element plus the element's class name. But the `event.dataTransfer` object cannot store array data types, so the `Array.join()` method converts the array to a string with a colon separating the entries. This string, then, is stuffed into the `event.dataTransfer` object. The object is instructed to render the cursor display during the drag-and-drop operation so that when the cursor is atop a drop target, the cursor is the copy style. Finally, the `setupDrag()` function is the first to execute in the drag operation, so a timer is set to the current clock time to time the drag operation.

The `ondrag` event handler (in the `body`) captures the `ondrag` events that are generated by whichever table cell element is the source element for the action. Each time the event fires (which is a lot during the action), the `timeIt()` function is invoked to compare the current time against the reference time (global `timer`) set when the drag starts. If the time exceeds 2 seconds (2,000 milliseconds), an alert dialog box notifies the user. To close the alert dialog box, the user must unclick the mouse button to end the drag operation.

TABLE 26-11

Events and Their Targets during a Typical Drag-and-Drop Operation

Event	Target	Discussion
`ondragstart`	cell `A1`	The very first event that fires during a drag-and-drop operation.
`ondrag`	cell `A1`	Fires continually on this target throughout the entire operation. Other events get interspersed, however.
`ondragenter`	cell `A1`	Even though the cursor hasn't moved from cell A1 yet, the `ondragenter` event fires upon first movement within the source element.
`ondragover`	cell `A1`	Fires continually on whatever element the cursor rests on at that instant. If the user simply holds the mouse button down and does not move the cursor during a drag, the `ondrag` and `ondragover` events fire continually, alternating between the two.
`(repetition)`	cell `A1`	`ondrag` and `ondragover` events fire alternately while the cursor remains atop cell A1.
`ondragenter`	`table`	The `table` element, represented by the border and/or cell padding, receives the `ondragenter` event when the cursor touches its space.
`ondragleave`	cell `A1`	Notice that the `ondragleave` event fires after the `ondragenter` event fires on another element.
`ondrag`	cell `A1`	Still firing away.
`ondragover`	`table`	The source element for this event shifts to the table because that's what the cursor is over at this instant. If the cursor doesn't move from this spot, the `ondrag` (cell A1) and `ondragover` (table) events continue to fire in turn.
`ondragenter`	cell `B1`	The drag is progressing from the `table` border space to cell B1.
`ondragleave`	`table`	The `table` element receives the `ondragleave` event when the cursor exits its space.
`ondrag`	cell `A1`	The `ondrag` event continues to fire on the cell A1 object.
`ondragover`	cell `B1`	The cursor is atop cell B1 now, so the `ondragover` event fires for that object. Fires multiple times (depending on the speed of the computer and the user's drag action), alternating with the previous `ondrag` event.
		More of the same as the cursor progresses from cell B1 through the `table` border again to cell B2, the `table` again, cell B3, and the outermost edge of the `table`.
`ondragenter`	`body`	Dragging is free of the `table` and is floating free on the bare `body` element.
`ondragleave`	`table`	Yes, you just left the `table`.
`ondrag`	cell `A1`	Still alive and receiving this event.

Event	Target	Discussion
ondragover	body	That's where the cursor is now. Fires multiple times (depending on the speed of the computer and the user's drag action), alternating with the previous ondrag event.
ondragenter	blank1	The cursor reaches the span element whose ID is blank1, where the empty underline is.
ondragleave	body	Just left the body for the blank.
ondrag	cell A1	Still kicking.
ondragover	blank1	That's where the cursor is now. Fires multiple times (depending on the speed of the computer and the user's drag action), alternating with the previous ondrag event.
ondrop	blank1	The span element gets the notification of a recent drop.
ondragend	cell A1	The original source element gets the final word that dragging is complete. This event fires even if the drag does not succeed because the drag does not end on a drop target.

To turn the blank span elements into drop targets, their ondragenter, ondragover, and ondrop event handlers must set event.returnValue to false; also, the event.dataTransfer .dropEffect property should be set to the desired effect (copy, in this case). These event handlers are placed in the p element that contains the two span elements, again for simplicity. Notice, however, that the cancelDefault() functions do their work only if the target element is one of the span elements whose ID begins with blank.

As the user releases the mouse button, the ondrop event handler invokes the handleDrop() function. This function retrieves the string data from event.dataTransfer and restores it to an array data type (using the String.split() method). A little bit of testing makes sure that the word type (noun or adjective) is associated with the desired blank. If so, the source element's text is set to the drop target's innerText property; otherwise, an error message is assembled to help the user know what went wrong.

LISTING 26-37

Using Drag-Related Event Handlers

HTML: jsb26-37.html

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Dragging Event Handlers</title>
      <style type="text/css">
         td
         {
            text-align:center;
```

continued

LISTING 26-37 *(continued)*

```
         }
         th
         {
            text-decoration:underline;
         }
         .blanks
         {
            text-decoration:underline;
         }
      </style>
      <script type="text/javascript" src="../jsb-global.js"></script>
      <script type="text/javascript" src="jsb26-37.js"></script>
   </head>
   <body ondragstart="setupDrag()" ondrag="timeIt()">
      <h1>Dragging Event Handlers</h1>
      <hr />
      <p>Your goal is to drag one noun and one adjective from the following
         table into the blanks of the sentence. Select a word from the table
         and drag it to the desired blank. When you release the mouse, the word
         will appear in the blank. You have two seconds to complete each blank.
      </p>
      <table cellpadding="5">
         <tr>
            <th>Nouns</th>
            <th>Adjectives</th>
         </tr>
         <tr>
            <td class="noun">truck</td>
            <td class="adjective">round</td>
         </tr>
         <tr>
            <td class="noun">doll</td>
            <td class="adjective">red</td>
         </tr>
         <tr>
            <td class="noun">ball</td>
            <td class="adjective">pretty</td>
         </tr>
      </table>
      <p id="myP" ondragenter="cancelDefault()" ondragover="cancelDefault()"
         ondrop="handleDrop()">Pat said, "Oh my, the <span id="blank1"
         class="blanks">     </span> is so <span
         id="blank2" class="blanks">     </span> !"
      </p>
      <button onclick="location.reload()">Reset</button>
   </body>
</html>
```

JavaScript: jsb26-37.js

```
var timer;
```

```
function setupDrag()
{
   if (event.srcElement.tagName != "TD")
   {
      // don't allow dragging for any other elements
      event.returnValue = false;
   }
   else
   {
      // setup array of data to be passed to drop target
      var passedData = [event.srcElement.innerText, event.srcElement.className];
      // store it as a string
      event.dataTransfer.setData("Text", passedData.join(":"));
      event.dataTransfer.effectAllowed = "copy";
      timer = new Date();
   }
}

function timeIt()
{
   if (event.srcElement.tagName == "TD" && timer)
   {
      if ((new Date()) - timer > 2000)
      {
         alert("Sorry, time is up. Try again.");
         timer = 0;
      }
   }
}

function handleDrop()
{
   var elem = event.srcElement;
   var passedData = event.dataTransfer.getData("Text");
   var errMsg = "";
      if (passedData)
      {
         // reconvert passed string to an array
         passedData = passedData.split(":");
         if (elem.id == "blank1")
         {
            if (passedData[1] == "noun")
            {
               event.dataTransfer.dropEffect = "copy";
               event.srcElement.innerText = passedData[0];
            }
            else
            {
               errMsg = "You can't put an adjective into the noun placeholder.";
            }
         }
         else if (elem.id == "blank2")
```

continued

LISTING 26-37 *(continued)*

```
         {
            if (passedData[1] == "adjective")
            {
               event.dataTransfer.dropEffect = "copy";
               event.srcElement.innerText = passedData[0];
            }
            else
            {
               errMsg = "You can't put a noun into the adjective placeholder.";
            }
         }
         if (errMsg)
         {
            alert(errMsg);
         }
      }
}

function cancelDefault()
{
   if (event.srcElement.id.indexOf("blank") == 0)
   {
      event.dataTransfer.dropEffect = "copy";
      event.returnValue = false;
   }
}
```

One event handler not shown in Listing 26-37 is `ondragend`. You can use this event to display the elapsed time for each successful drag operation. Because the event fires on the drag source element, you can implement it in the `<body>` tag and filter events similar to the way the `ondragstart` or `ondrag` event handlers filter events for the `td` element.

Related Items: `event.dataTransfer` object; `ondragenter`, `ondragleave`, `ondragover`, `ondrop` event handlers

ondragenter
ondragleave
ondragover

Compatibility: WinIE5+, MacIE-, NN-, Moz+, Safari 1.3+, Opera-, Chrome+

These events fire during a drag operation. When the cursor enters the rectangular space of an element on the page, the `ondragenter` event fires on that element. Immediately thereafter, the `ondragleave` event fires on the element from which the cursor came. Although this may seem to occur out of sequence from the physical action, the events always fire in this order. Depending on the speed of the client computer's CPU and the speed of the user's dragging action, one or the other of these events may not fire — especially if the physical action outstrips the computer's capability to fire the events in time.

The `ondragover` event fires continually while a dragged cursor is atop an element. In the course of dragging from one point on the page to another, the `ondragover` event target changes with the element beneath the cursor. If no other drag-related events are firing (the mouse button is still down in the drag operation, but the cursor is not moving), the `ondrag` and `ondragover` events fire continually, alternating between the two.

You should have the `ondragover` event handler of a drop target element set the `event.returnValue` property to `false`. See the discussion of the `ondrag` event handler earlier in this chapter for more details on the sequence of drag-related events.

Example

Listing 26-38 shows the `ondragenter` and `ondragleave` event handlers in use. The simple page displays (via the status bar) the time of entry to one element of the page. When the dragged cursor leaves the element, the `ondragleave` event handler hides the status-bar message. No drop target is defined for this page, so when you drag the item, the cursor remains the no-drop cursor.

LISTING 26-38

Using ondragenter and ondragleave Event Handlers

HTML: jsb26-38.html

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>ondragenter and ondragleave Event Handlers</title>
      <script type="text/javascript" src="../jsb-global.js"></script>
      <script type="text/javascript" src="jsb26-38.js"></script>
   </head>
   <body>
      <h1 ondragenter="showEnter()" ondragleave="clearMsg()">
         ondragenter and ondragleave Event Handlers
      </h1>
      <hr />
      <p>Select any character(s) from this paragraph, and slowly drag it around
         the page. When the dragging action enters the large header above, the
         status bar displays when the onDragEnter event handler fires. When you
         leave the header, the message is cleared via the onDragLeave event
         handler.
      </p>
   </body>
</html>
```

JavaScript: jsb26-38.js

```
function showEnter()
{
   status = "Entered at: " + new Date();
   event.returnValue = false;
}
```

continued

LISTING 26-38 *(continued)*

```
function clearMsg()
{
   status = "";
   event.returnValue = false;
}
```

Related Items: `ondrag`, `ondragend`, `ondragstart`, `ondrop` event handlers

ondragstart

(See `ondrag`)

ondrop

Compatibility: WinIE5+, MacIE-, NN-, Moz+, Safari 1.3+, Opera-, Chrome+

The `ondrop` event fires on the drop target element as soon as the user releases the mouse button at the end of a drag-and-drop operation. For IE, Microsoft recommends that you denote a drop target by applying the `ondragenter`, `ondragover`, and `ondrop` event handlers to the target element. In each of those event handlers, you should set the `dataTransfer.dropEffect` to the transfer effect you wish to portray in the drag-and-drop operation (signified by a different cursor for each type). These settings should match the `dataTransfer.effectAllowed` property that is usually set in the `ondragstart` event handler. Each of the three drop-related handlers should also override the default event behavior by setting the `event.returnValue` property to `false`. See the discussion of the `ondrag` event handler earlier in this chapter for more details on the sequence of drag-related events.

Example

See Listing 26-37 of the `ondrag` event handler to see how to apply the `ondrop` event handler in a typical drag-and-drop scenario.

Related Items: `event.dataTransfer` object; `ondrag`, `ondragend`, `ondragenter`, `ondragleave`, `ondragover`, `ondragstart` event handlers

onerrorupdate

Compatibility: WinIE4+, MacIE5+, NN-, Moz-, Safari-, Opera-, Chrome-

The `onerrorupdate` event handler is part of the data binding of IE and fires when an error occurs while updating the data in the data source object.

onfilterchange

Compatibility: WinIE4+, MacIE-, NN-, Moz-, Safari-, Opera-, Chrome-

The `onfilterchange` event fires whenever an object's visual filter switches to a new state or a transition completes (a transition may be extended over time). Only objects that accommodate filters and transitions in IE (primarily block elements and form controls) receive the event.

A common usage of the `onfilterchange` event is to trigger the next transition within a sequence of transition activities. This may include an infinite loop transition, for which the object receiving the event toggles between two transition states. If you don't want to get into a loop of that kind, place the different sets of content in their own positionable elements, and use the `onfilterchange` event handler in one to trigger the transition in the other.

Example

Listing 26-39 demonstrates how the `onfilterchange` event handler can trigger a second transition effect after another one completes. The `onload` event handler triggers the first effect. Although the `onfilterchange` event handler works with most of the same objects in IE4 as IE5, the filter object transition properties are not reflected in a convenient form. The syntax shown in Listing 26-39 uses the more modern ActiveX filter control found in IE5.5+ (described in Chapter 38).

LISTING 26-39

Using the onFilterChange Event Handler

HTML: jsb26-39.html

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>onfilterchange  Event Handler</title>
      <style type="text/css">
         #image1
         {
            position:absolute; top:150px; left:150px;
            filter:progID:DXImageTransform.Microsoft.Iris(irisstyle='CIRCLE',
                motion='in')
         }
         #image2
         {
            position:absolute; top:150px; left:150px;
            filter:progID:DXImageTransform.Microsoft.Iris(irisstyle='CIRCLE',
                motion='out')
         }
         .anImage
         {
            height:90px; width:120px;
         }
      </style>
      <script type="text/javascript" src="../jsb-global.js"></script>
      <script type="text/javascript" src="jsb26-39.js"></script>
   </head>
   <body onload="init()">
      <h1>onfilterchange  Event Handler</h1>
      <hr />
      <p>The completion of the first transition ("circle-in") triggers the
         second ("circle-out").
         <button onclick="location.reload()">Play It Again</button>
```

continued

LISTING 26-39 *(continued)*

```
      </p>
      <div id="image1" style="visibility:visible;"
           onfilterchange="finish()">
         <img class="anImage" alt="image" src="desk1.gif" />
      </div>
      <div id="image2" style="visibility:hidden;" >
         <img class="anImage" alt="image" src="desk3.gif" />
      </div>
   </body>
</html>
```

JavaScript: jsb26-39.js

```
function init()
{
   image1.filters[0].apply();
   image2.filters[0].apply();
   start();
}

function start()
{
   image1.style.visibility = "hidden";
   image1.filters[0].play();
}

function finish()
{
   // verify that first transition is done (optional)
   if (image1.filters[0].status == 0)
   {
      image2.style.visibility = "visible";
      image2.filters[0].play();
   }
}
```

Related Item: `filter` object

onfocus

Compatibility: WinIE3+, MacIE3+, NN2+, Moz+, Safari+, Opera+, Chrome+

The `onfocus` event fires when an element receives focus, usually following some other object's losing focus. (The element losing focus receives the `onblur` event before the current object receives the `onfocus` event.) For example, a text input element fires the `onfocus` event when a user tabs to that element while navigating through a form via the keyboard. Clicking an element also gives that element focus, as does making the browser the frontmost application on the client desktop.

The availability of the `onfocus` event has expanded with succeeding generations of script-capable browsers. In earlier versions, blur and focus were largely confined to text-oriented input elements such as the `select` element. The `window` object received the `onfocus` event handler starting with NN3 and IE4. IE4 also extended the event handler to more form elements, predominantly on the Windows operating system because that OS has a user interface clue (the dotted rectangle) when items such as buttons and links receive focus (so that users may act on them by pressing the spacebar). For IE5+, the `onfocus` event handler is available to virtually every HTML element. For most of those elements, however, you cannot use blur and focus unless you assign a value to the `tabindex` attribute of the element's tag. For example, if you assign `tabindex="1"` inside a `<p>` tag, the user can bring focus to that paragraph (highlighted with the dotted rectangle in Windows) by clicking the paragraph or pressing the Tab key until that item receives focus in sequence.

WinIE5.5 adds the `onfocusin` event handler, which fires immediately before the `onfocus` event handler. You can use one or the other, but there is little need to include both event handlers for the same object unless you wish to block an item temporarily from receiving focus. To prevent an object from receiving focus in IE5.5+, include an `event.returnValue=false` statement in the `onfocusin` event handler for the same object. In other browsers, you can usually get away with assigning `onfocus="this.blur()"` as an event handler for elements such as form controls. However, this is not a foolproof way to prevent a user from changing a control's setting. Unfortunately, there are few reliable alternatives short of disabling the control.

Example

See Listing 26-34 earlier in this chapter for an example of the `onfocus` and `onblur` event handlers.

Related Items: `onactivate`, `onblur`, `ondeactivate`, `onfocusin`, `onfocusout` event handlers

onfocusin
onfocusout

Compatibility: WinIE6+, MacIE-, NN-, Moz-, Safari-, Opera-, Chrome-

The `onfocusin` and `onfocusout` events fire to indicate that an element is about to receive focus or has just lost focus. These events are closely related to `onactivate` and `ondeactivate` except that in IE5.5+ activation and focus can be distinguished from each other. For example, if you set an element as the active element through `setActive()`, the element becomes active, but it does not gain the input focus. However, if you set the focus of an element with a call to `focus()`, the element is activated and gains input focus.

Related Items: `onactivate`, `onblur`, `ondeactivate`, `onfocus` event handlers

onhelp

Compatibility: WinIE4+, MacIE5+, NN-, Moz-, Safari-, Opera-, Chrome-

The `onhelp` event handler fires in Windows whenever an element of the document has focus and the user presses the F1 function key on a Windows PC. As of MacIE5, the event fires only on the window (in other words, event handler specified in the `<body>` tag) and does so via the dedicated Help key on a Mac keyboard. Browser Help menu choices do not activate this event. To prevent the browser's Help window from appearing, the event handler must evaluate to `return false` (for IE4+) or set the `event.returnValue` property to `false` (IE5+). Because the event handler can be associated with individual elements of a document in the Windows version, you can create a context-sensitive help system. However, if the focus is in the Address field of the browser window, you cannot intercept the event. Instead, the browser's Help window appears.

***elementObject*.onhelp**

Example

Listing 26-40 is a rudimentary example of a context-sensitive help system that displays help messages tailored to the kind of text input required by different text boxes. When the user gives focus to either of the text boxes, a small legend appears to remind the user that help is available by a press of the F1 help key. MacIE5 provides only generic help.

LISTING 26-40

Creating Context-Sensitive Help

HTML: jsb26-40.html

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>onhelp Event Handler</title>
      <style type="text/css">
         #legend
         {
            font-size:10px;
         }
      </style>
      <script type="text/javascript" src="../jsb-global.js"></script>
      <script type="text/javascript" src="jsb26-40.js"></script>
   </head>
   <body onload="init()" onhelp="return showGenericHelp()">
      <h1>onhelp Event Handler</h1>
      <hr />
      <p id="legend" style="visibility:hidden;"> </p>
      <form>
         Name:
         <input type="text" name="name" size="30"
                onfocus="showLegend()" onblur="hideLegend()"
                onhelp="return showNameHelp()" />
         <br />
         Year of Birth:
         <input type="text" name="YOB" size="30"
                onfocus="showLegend()" onblur="hideLegend()"
                onhelp="return showYOBHelp()" />
      </form>
   </body>
</html>
```

JavaScript: jsb26-40.js

```
function showNameHelp()
{
   alert("Enter your first and last names.");
   event.cancelBubble = true;
```

```
   return false;
}
function showYOBHelp()
{
   alert("Enter the four-digit year of your birth. For example: 1972");
   event.cancelBubble = true;
   return false;
}
function showGenericHelp()
{
   alert("All fields are required.");
   event.cancelBubble = true;
   return false;
}
function showLegend()
{
   document.getElementById("legend").style.visibility = "visible";
}
function hideLegend()
{
   document.getElementById("legend").style.visibility = "hidden";
}
function init()
{
   var msg = "";
      if (navigator.userAgent.indexOf("Mac") != -1)
      {
         msg = "Press \'help\' key for help.";
      }
      else if (navigator.userAgent.indexOf("Win") != -1)
      {
         msg = "Press F1 for help.";
      }
   document.getElementById("legend").style.visibility = "hidden";
   document.getElementById("legend").innerHTML = msg;
}
```

Related Items: `window.showHelp()`, `window.showModalDialog()` methods

onkeydown
onkeypress
onkeyup

Compatibility: WinIE4+, MacIE4+, NN4+, Moz+, Safari+, Opera+, Chrome+

When someone presses and releases a keyboard key, a sequence of three events fires in quick succession. The `onkeydown` event fires when the key makes its first contact. This is followed immediately by the `onkeypress` event. When contact is broken by the key release, the `onkeyup` event fires.

If you hold a character key down until it begins autorepeating, the `onkeydown` and `onkeypress` events fire with each repetition of the character.

The sequence of events can be crucial in some keyboard event handling. Consider the scenario that wants the focus of a series of text boxes to advance automatically after the user enters a fixed number of characters (for example, date, month, and two-digit year). By the time the `onkeyup` event fires, the character associated with the key-press action is already added to the box and you can accurately determine the length of text in the box, as shown in this simple example:

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Simple entry form</title>
      <script type="text/javascript">
         function jumpNext(fromFld, toFld)
         {
            if (fromFld.value.length == 2)
            {
               document.dateForm.elements[toFld].focus();
               document.dateForm.elements[toFld].select();
            }
         }
      </script>
   </head>
   <body>
      <form name="dateForm">
         Month: <input name="month" type="text" size="3" value=""
                      onkeyup="jumpNext(this, 'day')" maxlength="2" />
         Day: <input name ="day" type="text" size="3" value=""
                    onkeyup ="jumpNext(this, 'year')" maxlength="2" />
         Year: <input name="year" type="text" size="3" value=""
                     onkeyup ="jumpNext(this, 'month')" maxlength="2" />
      </form>
   </body>
</html>
```

These three events do not fire for all keys of the typical PC keyboard on all browser versions that support keyboard events. The only keys that you can rely on supporting the events in all browsers shown in the preceding compatibility chart are the alphanumeric keys represented by ASCII values, including the spacebar and Enter (Return on the Mac), but excluding all function keys, arrow keys, and other navigation keys. Modifier keys, such as Shift, Ctrl (PC), Alt (PC), Command (Mac), and Option (Mac), generate some events on their own (depending on browser and version). However, functions invoked by other key events can always inspect the pressed states of these modifier keys.

Caution

The `onkeydown` event handler works in Mozilla-based browsers only starting with Mozilla 1.4 (and Netscape 7.1). ■

Scripting keyboard events almost always entails examining which key is pressed so that some processing or validation can be performed on that key press. This is where the situation gets very complex if you are writing for cross-browser implementation. In some cases, even writing just for Internet

Explorer gets tricky because nonalphanumeric keys generate only the `onkeydown` and `onkeyup` events.

In fact, to comprehend keyboard events fully, you need to make a distinction between *key codes* and *character codes*. Every PC keyboard key has a key code associated with it. This key code is always the same regardless of what other keys you press at the same time. Only the alphanumeric keys (letters, numbers, spacebar, and so on), however, generate character codes. The code represents the typed character produced by that key. The value might change if you press a modifier key. For example, if you press the A key by itself, it generates a lowercase *a* character (character code 97); if you also hold down the Shift key, that same key produces an uppercase *A* character (character code 65). The key code for that key (65 for Western-language keyboards) remains the same no matter what.

That brings us, then, to where these different codes are made available to scripts. In all cases, the code information is conveyed as one or two properties of the browser's `event` object. IE's `event` object has only one such property: `keyCode`. It contains key codes for `onkeydown` and `onkeyup` events but character codes for `onkeypress` events. The NN6+/Moz event object, on the other hand, contains two separate properties: `charCode` and `keyCode`. You can find more details and examples about these `event` object properties in Chapter 32.

The bottom-line script consideration is to use either `onkeydown` or `onkeyup` event handlers when you want to look for nonalphanumeric key events (for example, function keys, arrow and page-navigation keys, and so on). To process characters as they appear in text boxes, use the `onkeypress` event handler. You can experiment with these events and codes in Listing 26-41, as well as in examples from Chapter 32.

Common keyboard event tasks

WinIE4+ enables you to modify the character that a user who is editing a text box enters. The `onkeypress` event handler can modify the `event.keyCode` property and allow the event to continue (in other words, don't evaluate to `return false` or set the `event.returnValue` property to `false`). The following IE function (invoked by an `onkeypress` event handler) makes sure that text entered in a text box is all uppercase, even if you type it as lowercase:

```
function assureUpper()
{
   if (event.keyCode >= 97 && event.keyCode <= 122)
   {
      event.keyCode = event.keyCode - 32;
   }
}
```

Doing this might confuse (or frustrate) users, so think carefully before implementing such a plan.

To prevent a key press from becoming a typed character in a text box, the `onkeypress` event handler prevents the default action of the event. For example, the following HTML page shows how to inspect a text box's entry for numbers only:

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Keyboard Capture</title>
      <script type="text/javascript">
```

```
            function checkIt(evt)
            {
               var charCode = (evt.charCode) ?
                     evt.charCode : ((evt.which) ? evt.which : evt.keyCode);
                  if (charCode > 31 && (charCode < 48 || charCode > 57))
                  {
                     alert("Please make sure entries are numbers only.");
                     return false;
                  }
               return true;
            }
         </script>
      </head>
      <body>
         <form>
            Enter any positive integer:
            <input type="text" name="numeric" onkeypress="return checkIt(event)">
         </form>
      </body>
   </html>
```

Whenever a user enters a non-number, the user receives a warning, and the character is not appended to the text box's text.

Keyboard events also enable you to script the submission of a form when a user presses the Enter (Return on the Mac) key within a text box. The ASCII value of the Enter/Return key is 13. Therefore, you can examine each key press in a text box and submit the form whenever value 13 arrives, as shown in the following function:

```
function checkForEnter(evt)
{
   evt = (evt) ? evt : event;
   var charCode = (evt.charCode) ?
         evt.charCode : ((evt.which) ? evt.which : evt.keyCode);

      if (charCode == 13)
         {
         document.forms[0].submit();
         return false;
         }
      return true;
}
```

By assigning the `checkForEnter()` function to each box's `onkeypress` event handler, you suddenly add some extra power to a typical HTML form.

You can intercept Ctrl+keyboard combinations (letters only) in HTML pages most effectively in IE, but only if the browser itself does not use the combination. In other words, you cannot redirect Ctrl+key combinations that the browser uses for its own control. The `onkeypress keyCode` value for Ctrl+key combinations ranges from 1 through 26 for letters A through Z (except for those used by the browser, in which case no keyboard event fires).

Example

Listing 26-41 is a working laboratory that you can use to better understand the way keyboard event codes and modifier keys work in IE5+ and W3C browsers. The actual code of the listing is less important than watching the page while you use it. For every key or key combination that you press, the page shows the `keyCode` value for the `onkeydown`, `onkeypress`, and `onkeyup` events. If you hold down one or more modifier keys while performing the key press, the modifier-key name is highlighted for each of the three events. Note that when run in NN6+/Moz-based browsers/WebKit-based browsers, the `keyCode` value is not the character code. In older browsers, you may need to click the page for the `document` object to recognize the keyboard events.

The best way to watch what goes on during keyboard events is to press and hold a key to see the key codes for the `onkeydown` and `onkeypress` events. Then release the key to see the code for the `onkeyup` event. Notice, for instance, that if you press the A key without any modifier key, the `onkeydown` event key code is 65 (A), but the `onkeypress` key code in IE (and the `charCode` property in NN6+/Moz-based browsers/WebKit-based browsers) is 97 (a). If you then repeat the exercise but hold the Shift key down, all three events generate the 65 (A) key code (and the Shift modifier labels are highlighted). Releasing the Shift key causes the `onkeyup` event to show the key code for the Shift key.

In another experiment, press any of the four arrow keys. No key code is passed for the `onkeypress` event because those keys don't generate those events. They do, however, generate `onkeydown` and `onkeyup` events.

LISTING 26-41

Keyboard Event Handler Laboratory

HTML: jsb26-41.html

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Keyboard Event Handler Lab</title>
      <style type="text/css">
         td
         {
            text-align:center;
         }
      </style>
      <script type="text/javascript" src="../jsb-global.js"></script>
      <script type="text/javascript" src="jsb26-41.js"></script>
   </head>
   <body onload="init()">
      <h1>Keyboard Event Handler Lab</h1>
      <hr />
      <form>
         <table border="2" cellpadding="2">
            <tr>
               <th></th>
```

continued

LISTING 26-41 *(continued)*

```
               <th>onKeyDown</th>
               <th>onKeyPress</th>
               <th>onKeyUp</th>
            </tr>
            <tr>
               <th>Key Codes</th>
               <td id="downKeyCode">0</td>
               <td id="pressKeyCode">0</td>
               <td id="upKeyCode">0</td>
            </tr>
            <tr>
               <th>Char Codes (IE5/Mac; Moz; WebKit)</th>
               <td id="downCharCode">0</td>
               <td id="pressCharCode">0</td>
               <td id="upCharCode">0</td>
            </tr>
            <tr>
               <th rowspan="3">Modifier Keys</th>
               <td><span id="shiftDown">Shift</span></td>
               <td><span id="shift">Shift</span></td>
               <td><span id="shiftUp">Shift</span></td>
            </tr>
            <tr>
               <td><span id="ctrlDown">Ctrl</span></td>
               <td><span id="ctrl">Ctrl</span></td>
               <td><span id="ctrlUp">Ctrl</span></td>
            </tr>
            <tr>
               <td><span id="altDown">Alt</span></td>
               <td><span id="alt">Alt</span></td>
               <td><span id="altUp">Alt</span></td>
            </tr>
         </table>
      </form>
   </body>
</html>
```

JavaScript: jsb26-41.js

```
function init()
{
   document.onkeydown = showKeyDown;
   document.onkeyup = showKeyUp;
   document.onkeypress = showKeyPress;
}

function showKeyDown(evt)
{
```

```
    evt = (evt) ? evt : window.event;
    document.getElementById("pressKeyCode").innerHTML = 0;
    document.getElementById("upKeyCode").innerHTML = 0;
    document.getElementById("pressCharCode").innerHTML = 0;
    document.getElementById("upCharCode").innerHTML = 0;
    restoreModifiers("");
    restoreModifiers("Down");
    restoreModifiers("Up");
    document.getElementById("downKeyCode").innerHTML = evt.keyCode;
       if (evt.charCode)
       {
          document.getElementById("downCharCode").innerHTML = evt.charCode;
       }
    showModifiers("Down", evt);
 }

 function showKeyUp(evt)
 {
    evt = (evt) ? evt : window.event;
    document.getElementById("upKeyCode").innerHTML = evt.keyCode;
       if (evt.charCode)
       {
          document.getElementById("upCharCode").innerHTML = evt.charCode;
       }
    showModifiers("Up", evt);
    return false;
 }

 function showKeyPress(evt)
 {
    evt = (evt) ? evt : window.event;
    document.getElementById("pressKeyCode").innerHTML = evt.keyCode;
       if (evt.charCode)
       {
          document.getElementById("pressCharCode").innerHTML = evt.charCode;
       }
    showModifiers("", evt);
    return false;
 }

 function showModifiers(ext, evt)
 {
    restoreModifiers(ext);
    if (evt.shiftKey)
    {
       document.getElementById("shift" + ext).style.backgroundColor = "#ff0000";
    }
    if (evt.ctrlKey)
    {
       document.getElementById("ctrl" + ext).style.backgroundColor = "#00ff00";
    }
```

continued

LISTING 26-41 *(continued)*

```
   if (evt.altKey)
   {
      document.getElementById("alt" + ext).style.backgroundColor = "#0000ff";
   }
}

function restoreModifiers(ext)
{
   document.getElementById("shift" + ext).style.backgroundColor = "#ffffff";
   document.getElementById("ctrl" + ext).style.backgroundColor = "#ffffff";
   document.getElementById("alt" + ext).style.backgroundColor = "#ffffff";
}
```

Spend some time with this lab, and try all kinds of keys and key combinations until you understand the way the events and key codes work.

Related Item: `String.fromCharCode()` method

onlayoutcomplete

Compatibility: WinIE5.5+, MacIE-, NN-, Moz-, Safari-, Opera-, Chrome-

The `onlayoutcomplete` event handler fires when a print or print-preview layout operation completes on the current layout rectangle (`LayoutRect` object). This event is primarily used as the basis for overflowing content from one page to another during printing. In response to the `onlayoutcomplete` event, the `contentOverflow` property can be inspected to determine whether page content has indeed overflowed the current layout rectangle.

onlosecapture

Compatibility: WinIE5+, MacIE-, NN-, Moz-, Safari-, Opera-, Chrome-

The `onlosecapture` event handler fires whenever an object that has event capture turned on no longer has that capture. Event capture is automatically disengaged when the user performs any of the following actions:

- Gives focus to any other window
- Displays any system modal dialog box (for example, alert window)
- Scrolls the page
- Opens a browser context menu (right-clicking)
- Tabs to give focus to the Address field in the browser window

A function associated with the `onlosecapture` event handler should perform any cleanup of the environment due to an object's no longer capturing mouse events.

Example

See Listing 26-30 earlier in this chapter for an example of how to use `onlosecapture` with an event-capturing scenario for displaying a context menu. The `onlosecapture` event handler hides the context menu when the user performs any action that causes the menu to lose mouse capture.

Related Items: `releaseCapture()`, `setCapture()` methods

onmousedown
onmouseup

Compatibility: WinIE4+, MacIE4+, NN4+, Moz+, Safari+, Opera+, Chrome+

The `onmousedown` event handler fires when the user presses any button on a mouse. The `onmouseup` event handler fires when the user releases the mouse button, provided that the object receiving the event also received an `onmousedown` event. When a user performs a typical click of the mouse button atop an object, mouse events occur in the following sequence: `onmousedown`, `onmouseup`, and `onclick`. But if the user presses the mouse atop an object and then slides the cursor away from the object, only the `onmousedown` event fires.

These events enable authors and designers to add more applicationlike behavior to images that act as action or icon buttons. If you notice the way most buttons work, the appearance of the button changes while you press the mouse button and reverts to its original style when you release the mouse button (or you drag the cursor out of the button). These events enable you to emulate that behavior.

The event object created with every mouse button action has a property that reveals which mouse button the user pressed. NN4's event model called that property the `which` property. IE4+ and NN6+/Moz-based browsers/WebKit-based browsers call it the `button` property (but with different values for the buttons). It is most reliable to test for the mouse button number on either the `onmousedown` or `onmouseup` event rather than on `onclick`. The `onclick` event object does not always contain the button information.

Example

To demonstrate a likely scenario of changing button images in response to rolling atop an image, pressing down on it, releasing the mouse button, and rolling away from the image, Listing 26-42 presents a pair of small navigation buttons (left- and right-arrow buttons). Images are preloaded into the browser cache as the page loads so that response to the user is instantaneous the first time the user calls upon new versions of the images.

LISTING 26-42

Using onmousedown and onmouseup Event Handlers

HTML: jsb26-42.html

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>onmousedown and onmouseup Event Handlers</title>
      <style type="text/css">
         #imgHolder
         {
```

continued

LISTING 26-42 *(continued)*

```
            text-align:center;
         }
         .images
         {
            height:16px; width:16px; border:0px;
         }
      </style>
      <script type="text/javascript" src="../jsb-global.js"></script>
      <script type="text/javascript" src="jsb26-42.js"></script>
   </head>
   <body>
      <h1>onmousedown and onmouseup Event Handlers</h1>
      <hr />
      <p>Roll atop and click on the buttons to see how the link event handlers
         swap the subtlety different images:
      </p>
      <div id="imgHolder">
         <img alt="image" class="images"
              name="Left" src="LeftNorm.gif"
              onmouseover="return setImage('Left','Up')"
              onmousedown="return setImage('Left','Down')"
              onmouseup="return setImage('Left','Up')"
              onmouseout="return setImage('Left','Norm')" />
           
         <img alt="image" class="images"
              name="Right" src="RightNorm.gif"
               onmouseover="return setImage('Right','Up')"
              onmousedown="return setImage('Right','Down')"
              onmouseup="return setImage('Right','Up')"
              onmouseout="return setImage('Right','Norm')" />
      </div>
   </body>
</html>
```

JavaScript: jsb26-42.js

```
var RightNormImg = new Image(16,16);
var RightUpImg = new Image(16,16);
var RightDownImg = new Image(16,16);
var LeftNormImg = new Image(16,16);
var LeftUpImg = new Image(16,16);
var LeftDownImg = new Image(16,16);

RightNormImg.src = "RightNorm.gif";
RightUpImg.src = "RightUp.gif";
RightDownImg.src = "RightDown.gif";
LeftNormImg.src = "LeftNorm.gif";
LeftUpImg.src = "LeftUp.gif";
LeftDownImg.src = "LeftDown.gif";
```

```
function setImage(imgName, type)
{
   var imgFile = eval(imgName + type + "Img.src");
   document.images[imgName].src = imgFile;
   return false;
}
```

Related Item: `onclick` event handler

onmouseenter
onmouseleave

Compatibility: WinIE5.5+, MacIE-, NN-, Moz-, Safari-, Opera-, Chrome-

WinIE5.5 introduced the `onmouseenter` and `onmouseleave` event handlers. Both event handlers operate just like the `onmouseover` and `onmouseout` event handlers, respectively. Microsoft simply offers an alternative terminology. The old and new events continue to fire in IE5.5+. The old ones fire just before the new ones for each act of moving the cursor atop, and exiting from atop, the object. If you are scripting exclusively for IE5.5+, you should use the new terminology; otherwise, stay with the older versions.

Example

You can modify Listing 26-43 with the IE5.5 syntax by substituting `onmouseenter` for `onmouseover` and `onmouseleave` for `onmouseout`. The effect is the same.

Related Items: `onmouseover`, `onmouseout` event handlers

onmousemove

Compatibility: WinIE4+, MacIE4+, NN4+, Moz+, Safari+, Opera+, Chrome+

The `onmousemove` event handler fires whenever the cursor is atop the current object and the mouse is moved, even by a single pixel. You do not have to press the mouse button for the event to fire, although the event is most commonly used in element dragging — especially in NN/FF3.4-/Opera, where no `ondrag` event handler is available.

Even though the granularity of this event can be at the pixel level, you should not use the number of event firings as a measurement device. Depending on the speed of cursor motion and the performance of the client computer, the event may not fire at every pixel location.

In IE4+ and W3C DOM-compatible browsers, you can assign the `onmousemove` event handler to any element (although you can drag only with positioned elements). When designing a page that encourages users to drag multiple items on a page, it is most common to assign the `onmousemove` event handler to the `document` object and let all such events bubble up to the document for processing.

Example

See Chapter 43 and Chapter 59 on the CD-ROM for examples of using mouse events to control element dragging on a page.

Related Items: `ondrag`, `onmousedown`, `onmouseup` event handlers

onmouseout
onmouseover

Compatibility: WinIE3+, MacIE3+, NN2+, Moz+, Safari+, Opera+, Chrome+

The `onmouseover` event fires for an object whenever the cursor rolls into the rectangular space of the object on the screen. The `onmouseout` event handler fires when you move the cursor outside the object's rectangle. These events most commonly display explanatory text about an object in the window's status bar and effect image swapping (so-called mouse rollovers). Use the `onmouseover` event handler to change the state to a highlighted version; use the `onmouseout` event handler to restore the image or status bar to its normal setting.

Although these two events have been in object models of scriptable browsers since the beginning, they were not available to most objects in earlier browsers. IE4+ and W3C DOM-compatible browsers provide support for these events on every element that occupies space onscreen. IE5.5+ includes an additional pair of event handlers — `onmouseenter` and `onmouseleave` — that duplicates the `onmouseover` and `onmouseout` events but with different terminology. The old event handlers fire just before the new versions.

Note

The `onmouseout` event handler commonly fails to fire if the event is associated with an element that is near a frame or window edge and the user moves the cursor quickly outside the current frame. ■

Example

Listing 26-43 uses the U.S. Pledge of Allegiance with four links to demonstrate how to use the `onmouseover` and `onmouseout` event handlers. Notice that for each link, the handler runs a general-purpose function that sets the window's status message. The function returns a `true` value, which the event handler call evaluates to replicate the required `return true` statement needed for setting the status bar. In one status message, we supply a URL in parentheses to let you evaluate how helpful you think it is for users.

LISTING 26-43

Using onmouseover and onmouseout Event Handlers

HTML: jsb26-43.html

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>onmouseover and onmouseout Event Handlers</title>
      <script type="text/javascript" src="../jsb-global.js"></script>
      <script type="text/javascript" src="jsb26-43.js"></script>
   </head>
   <body>
      <h1>onmouseover and onmouseout Event Handlers</h1>
      <hr />
      <h1>Pledge of Allegiance</h1>
      <hr />
      I pledge
      <a href="javascript:emulate()"
```

```
            onmouseover="return setStatus('View dictionary definition')"
            onmouseout="return setStatus('')">allegiance</a>
      to the
      <a href="javascript:emulate()"
         onmouseover="return setStatus('Learn about the U.S. flag↩
            (http://lcweb.loc.gov)')"
         onmouseout="return setStatus('')">flag</a>
      of the
      <a href="javascript:emulate()"
         onmouseover="return setStatus('View info about the U.S. government')"
         onmouseout="return setStatus('')">United States of America</a>,
      and to the Republic for which it stands, one nation
      <a href="javascript:emulate()"
         onmouseover="return setStatus('Read about the history of this phrase↩
            in the Pledge')"
         onmouseout="return setStatus('')">under God</a>,
      indivisible, with liberty and justice for all.
   </body>
</html>
```

JavaScript: jsb26-43.js

```
function setStatus(msg)
{
   window.status = msg;
   return true;
}

// destination of all link HREFs
function emulate()
{
   alert("Not going there in this demo.");
}
```

Related Items: `onmouseenter`, `onmouseleave`, `onmousemove` event handlers

onmousewheel

Compatibility: WinIE6+, MacIE-, NN-, Moz+, Safari+, Opera+, Chrome+

The `onmousewheel` event handler fires in response to the user's rotating the mouse wheel. When responding to the `onmousewheel` event, you can check the `wheelDelta` property to find out how far the mouse wheel has been rotated. The `wheelDelta` property expresses mouse wheel rotations in multiples of 120, with positive values indicating a rotation away from the user and negative values corresponding to a rotation toward the user. The W3C equivalent is `DOMMouseScroll`.

Related Item: `onmousemove` event handler

onmove

Compatibility: WinIE5.5+, MacIE-, NN-, Moz-, Safari-, Opera-, Chrome-

Not to be confused with `onmousemove`, the `onmove` event has nothing to do with the mouse. Instead, the `onmove` event is fired whenever a positionable *element* is moved. For example, if a `div` element is created as an absolutely positioned moveable element, you can track its movement by responding to the `onmove` event. The `offsetLeft` and `offsetTop` properties can be used within this event handler to determine the exact location of the element as it is moving.

Related Items: `onmoveend`, `onmovestart` event handlers

onmoveend
onmovestart

Compatibility: WinIE5.5+, MacIE-, NN-, Moz-, Safari-, Opera-, Chrome-

The `onmovestart` and `onmoveend` event handlers fire in response to a positionable element's being moved on a page. More specifically, the `onmovestart` event is triggered when an element first starts moving and the `onmoveend` event fires when the element stops moving. In between the `onmovestart` and `onmoveend` events firing, multiple `onmove` events may be sent out to indicate the movement of the element.

Related Item: `onmove` event handler

onpaste

Compatibility: WinIE5+, MacIE-, NN-, Moz+, Safari 1.3+, Opera-, Chrome+

The `onpaste` event fires immediately after the user or script initiates a paste edit action on the current object. The event is preceded by the `onbeforepaste` event, which fires prior to any edit or context menu that appears (or before the paste action if initiated by keyboard shortcut).

Use this event handler to provide edit functionality to elements that don't normally allow pasting. In such circumstances, you need to enable the Paste menu item in the context or Edit menu by setting the `event.returnValue` for the `onbeforepaste` event handler to `false`. Then your `onpaste` event handler must manually retrieve data from the clipboard (by way of the `getData()` method of the IE `clipboardData` object) and handle the insertion into the current object.

Because you are in charge of what data is stored in the clipboard, you are not limited to a direct copy of the data. For example, you might wish to store the value of the `src` property of an image object so that you can paste it elsewhere on the page.

Example

Listing 26-44 demonstrates how to use the `onbeforepaste` and `onpaste` event handlers (in conjunction with `onbeforecopy` and `oncopy`) to let scripts control the data-transfer process during a copy-and-paste user operation. A table contains words to be copied (one column of nouns, one column of adjectives) and then pasted into blanks in a paragraph. The `onbeforecopy` and `oncopy` event handlers are assigned to the `table` element because the events from the `td` elements bubble up to the `table` container and there is less HTML code to contend with.

Inside the paragraph, two `span` elements contain underscored blanks. To paste text into the blanks, the user must first select at least one character of the blanks. (See Listing 26-37, which gives a drag-and-drop version of this application.) The `onbeforepaste` event handler in the paragraph (which gets the event as it bubbles up from either `span`) sets the `event.returnValue` property to `false`, thus allowing the Paste item to appear in the context and Edit menus (not a normal occurrence in HTML body content).

At paste time, the `innerHTML` property of the target `span` is set to the text data stored in the clipboard. The `event.returnValue` property is set to `false` here as well to prevent normal system pasting from interfering with the controlled version.

LISTING 26-44

Using onbeforepaste and onpaste Event Handlers

HTML: jsb26-44.html

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>onbeforepaste and onpaste Event Handlers</title>
      <style type="text/css">
         td
         {
            text-align:center;
         }
         th
         {
            text-decoration:underline;
         }
         .blanks
         {
            text-decoration:underline;
         }
      </style>
      <script type="text/javascript" src="../jsb-global.js"></script>
      <script type="text/javascript" src="jsb26-44.js"></script>
   </head>
   <body>
      <h1>onbeforepaste and onpaste Event Handlers</h1>
      <h6>Be very precise when selecting characters</h6>
      <hr />
      <p>Your goal is to copy and paste one noun and one adjective from the
         following table into the blanks of the sentence. Select a word from
         the table and use a menu to copy it to the clipboard. Select one or
         more spaces of the blanks in the sentence and choose Paste from a
         menu to replace the blank with the clipboard contents.
      </p>
      <table cellpadding="5" onbeforecopy="selectWhole()"
             oncopy="handleCopy()">
         <tr>
            <th>Nouns</th>
            <th>Adjectives</th>
         </tr>
         <tr>
            <td>truck</td>
            <td>round</td>
```

continued

***elementObject*.onpaste**

LISTING 26-44 *(continued)*

```
         </tr>
         <tr>
            <td>doll</td>
            <td>red</td>
         </tr>
         <tr>
            <td>ball</td>
            <td>pretty</td>
         </tr>
      </table>
      <p id="myP" onbeforepaste="handleBeforePaste()" onpaste="handlePaste()">
         Pat said, "Oh my, the <span id="blank1" class="blanks">
              </span>
         is so <span id="blank2" class="blanks">
              </span> !"
      </p>
      <button onclick="location.reload()">Reset</button>
   </body>
</html>
```

JavaScript: jsb26-44.js

```
function selectWhole()
{
   var obj = window.event.srcElement;
   var range = document.body.createTextRange();
   range.moveToElementText(obj);
   range.select();
   event.returnValue = false;
}
function handleCopy()
{
   var rng = document.selection.createRange();
   clipboardData.setData("Text",rng.text);
   event.returnValue = false;
}

function handlePaste()
{
   var elem = window.event.srcElement;
      if (elem.className == "blanks")
      {
         elem.innerHTML = clipboardData.getData("Text");
      }
   event.returnValue = false;
}
function handleBeforePaste()
{
   var elem = window.event.srcElement;
```

```
      if (elem.className == "blanks")
      {
         event.returnValue = false;
      }
}
```

Related Items: `oncopy`, `oncut`, `onbeforepaste` event handlers

onpropertychange

Compatibility: WinIE5+, MacIE-, NN-, Moz-, Safari-, Opera-, Chrome-

The `onpropertychange` event fires in WinIE5+ whenever a script modifies an object's property. This includes changes to the properties of an object's style. Changing properties by way of the `setAttribute()` method also triggers this event.

A script can inspect the nature of the property change because the `event.propertyName` property contains the name (as a string) of the property that was just changed. In the case of a change to an object's `style` object, the `event.propertyName` value begins with `"style."` as in `style.backgroundcolor`.

You can use this event handler to localize any object-specific postprocessing of changes to an object's properties. Rather than include the postprocessing statements inside the function that makes the changes, you can make that function generalized (perhaps to modify properties of multiple objects).

Example

Listing 26-45 shows how you can respond programmatically to an object's properties being changed. The page generated by the listing contains four radio buttons that alter the `innerHTML` and `style.color` properties of a paragraph. The paragraph's `onpropertychange` event handler invokes the `showChange()` function, which extracts information about the event and displays the data in the status bar of the window. Notice that the property name includes `style.` when you modify the style sheet property.

LISTING 26-45

Using the onPropertyChange Property

HTML: jsb26-45.html

```
<!DOCTYPE html>
<html>
   <head>
     <meta http-equiv="content-type" content="text/html;charset=utf-8">
     <title>onpropertychange Event Handler</title>
     <script type="text/javascript" src="../jsb-global.js"></script>
     <script type="text/javascript" src="jsb26-45.js"></script>
   </head>
   <body onload="init()";>
     <h1>onpropertychange Event Handler</h1>
```

continued

LISTING 26-45 *(continued)*

```
      <hr />
      <p id="myP" onpropertychange="showChange()">This is a sample paragraph.</p>
      <form>
         Text:
         <input type="radio" name="btn1" checked="checked"
                onclick="normalText()" />Normal
         <input type="radio" name="btn1"
                onclick="shortText()" />Short
         <br />
         Color:
         <input type="radio" name="btn2" checked="checked"
                onclick="normalColor()" />Black
         <input type="radio" name="btn2"
                onclick="hotColor()" />Red
      </form>
   </body>
</html>
```

JavaScript: jsb26-45.js

```
function init()
{
   elem = document.getElementById("myP");
}
function normalText()
{
   if (elem.textContent)
   {
      elem.textContent = "This is a sample paragraph.";
   }
   else
   {
      elem.innerText = "This is a sample paragraph.";
   }
}
function shortText()
{
   if (elem.textContent)
   {
      elem.textContent = "Short stuff.";
   }
   else
   {
      elem.innerText = "Short stuff.";
   }
}
function normalColor()
{
   elem.style.color = "black";
}
```

```
function hotColor()
{
   elem.style.color = "red";
}
function showChange()
{
   var objID = event.srcElement.id;
   var propName = event.propertyName;
   var newValue = eval(objID + "." + propName);
   var msg = "The " + propName + " property of the " + objID;
   msg += " object has changed to \"" + newValue + "\".";
   alert(msg);
}
```

Related Items: `style` property; `setAttribute()` method

onreadystatechange

Compatibility: WinIE4+, MacIE-, NN7+, Moz1.0.1+, Safari 1.2+, Opera+, Chrome+

The `onreadystatechange` event handler fires whenever the ready state of an object changes. See details about these states in the discussion of the `readyState` property earlier in this chapter (and notice the limits for IE4). The change of state does not guarantee that an object is in fact ready for script statements to access its properties. Always check the `readyState` property of the object in any script that the `onreadystatechange` event handler invokes.

This event fires for objects that are capable of loading data: `applet`, `document`, `frame`, `frameset`, `iframe`, `img`, `link`, `object`, `script`, and XML objects. The event doesn't fire for other types of objects unless a Microsoft DHTML behavior is associated with the object. The `onreadystatechange` event does not bubble; neither can you cancel it.

Related Item: `readyState` property

onresize

Compatibility: WinIE4+, MacIE4+, NN4+, Moz+, Safari+, Opera+, Chrome+

The `onresize` event handler fires whenever an object is resized in response to a variety of user or scripted actions. Most elements include this event handler, provided that the object has dimensional style attributes (for example, height, width, or position) assigned to it.

The `onresize` event does not bubble. Resizing the browser window or frame does not cause the window's `onload` event handler to fire.

Example

If you want to capture the user's resizing of the browser window (or frame), you can assign a function to the `onresize` event handler either via script

```
window.onresize = handleResize;
```

or by an HTML attribute of the `body` element:

```
<body onresize="handleResize()">
```

Related Item: `window.resize()` method

onresizeend
onresizestart

Compatibility: WinIE5.5+, MacIE-, NN-, Moz-, Safari-, Opera-, Chrome-

The `onresizeend` and `onresizestart` event handlers fire only on a resizable object in Windows edit mode.

Related Item: `oncontrolselect` event handler

onrowenter
onrowexit

Compatibility: WinIE4+, MacIE-, NN-, Moz-, Safari-, Opera-, Chrome-

The `onrowenter` and `onrowexit` events fire in response to changes in the current row of recordset data in IE data binding. More specifically, `onrowenter` is triggered when the data for the current row of data has changed and new data is available on the data source object. The `onrowexit` event is triggered when the current row is changing, meaning that another row of data is being selected; the event is fired just before the row changes.

Related Items: `onrowsdelete`, `onrowsinserted` event handlers

onrowsdelete
onrowsinserted

Compatibility: WinIE5+, MacIE-, NN-, Moz-, Safari-, Opera-, Chrome-

The `onrowsdelete` event fires when one or more rows of data are about to be deleted from the recordset in IE data binding. Conversely, the `onrowsinserted` event is triggered when one or more rows of data have been inserted into the recordset.

Related Items: `onrowenter`, `onrowexit` event handlers

onscroll

Compatibility: WinIE4+, MacIE4+, NN-, Moz+, Safari+, Opera_, Chrome+

The `onscroll` event handler fires whenever the scroll box within a scroll bar of an element is repositioned. In simpler terms, when the user clicks and drags the scroll box with the mouse, the `onscroll` event is fired. That's not the only action that triggers the event, however. An `onscroll` event is also fired in response to the user's clicking the scroll arrow, clicking the scroll bar, or pressing any of the following keys: Home, End, Space, Page Up, or Page Down. A call to the `doScroll()` method also triggers the event, as does the user's holding down the Up Arrow or Down Arrow key.

Related Item: `doScroll()` method

onselectstart

Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari 1.3+, Opera-, Chrome+

The `onselectstart` event handler fires when a user begins to select content on the page. Selected content can be inline text, images, or text within an editable text box. If the user selects more than one object, the event fires in the first object affected by the selection.

Example

Use the page from Listing 26-46 to see how the `onselectstart` event handler works when a user selects across multiple elements on a page. As the user begins a selection anywhere on the page, the ID of the object receiving the event appears in the status bar. Notice that the event doesn't fire until you actually make a selection. When no other element is under the cursor, the `body` element fires the event.

LISTING 26-46

Using the onselectstart Event Handler

HTML: jsb26-46.html

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>onselectstart Event Handler</title>
      <style type="text/css">
         h1, th
         {
            padding:5px;
         }
         td
         {
            text-align:center;
         }
      </style>
      <script type="text/javascript" src="../jsb-global.js"></script>
      <script type="text/javascript" src="jsb26-46.js"></script>
   </head>
   <body id="myBody" onselectstart="showObj()">
      <h1 id="myH1">
         onselectstart Event Handler
      </h1>
      <hr id="myHR" />
      <p id="myP">This is a sample paragraph.</p>
      <table border="1">
         <tr id="row1">
            <th id="header1">Column A</th>
            <th id="header2">Column B</th>
            <th id="header3">Column C</th>
         </tr>
         <tr id="row2">
            <td id="cellA2">text</td>
            <td id="cellB2">text</td>
```

continued

elementObject.onselectstart

LISTING 26-46 *(continued)*

```
            <td id="cellC2">text</td>
         </tr>
         <tr id="row3">
            <td id="cellA3">text</td>
            <td id="cellB3">text</td>
            <td id="cellC3">text</td>
         </tr>
      </table>
   </body>
</html>
```

JavaScript: jsb26-46.js

```
function showObj()
{
   var objID = event.srcElement.id;
   alert("Selection started with object: " + objID);
}
```

Related Item: `onselect` event handler for a variety of objects

CHAPTER 27

Window and Frame Objects

IN THIS CHAPTER

Scripting communication among multiple frames

Creating and managing new windows

Controlling the size, position, and appearance of the browser window

Details of window, frame, frameset, and iframe objects

A quick look at the basic document object model diagram in Chapter 25, "Document Object Model Essentials," (refer to Figure 25-1) reveals that the `window` object is the outermost global container of all document-related objects that you script with JavaScript. All HTML and JavaScript activity takes place inside a window. That window may be a standard Windows, Mac, or XWindows application-style window, complete with scroll bars, toolbars, and other chrome; you can also generate windows that have only some of a typical window's chrome. A frame is also a window, even though a frame doesn't have many accoutrements beyond scroll bars. The `window` object is where everything begins when JavaScript references objects. Modern browsers treat the frameset as a special kind of `window` object, so it is also covered in this chapter.

Of all the objects associated with browser scripting, the `window` and window-related objects have by far the most object-specific terminology associated with them. This necessitates a rather long chapter to keep the whole discussion in one place.

Window Terminology

The `window` object is often a source of confusion when you first learn about the document object model. A number of synonyms for `window` objects muck up the works: `top`, `self`, `parent`, and `frame`. Aggravating the situation is the fact that these terms are also properties of a `window` object. Under some conditions, a window is its own parent, but if you define a frameset with two frames, you have only one parent among a total of three `window` objects. It doesn't take long before the whole subject can make your head hurt.

If you do not use frames in your web applications, these headaches never appear. But if frames are part of your design plan, you should get to know how frames affect the object model.

Frames

The application of frames has become a religious issue among web designers: Some swear by them; others swear at them. We believe, after careful consideration, that there can be compelling reasons to use frames at times. Historically, any humungous document with a hierarchy, or that depends on a file system, might be a good candidate. For example, if you have a document that requires considerable scrolling to get through, you may want to maintain a static set of navigation controls that are visible at all times. By placing those controls — be they links or image maps — in a separate frame, you have made the controls available for immediate access, regardless of the scrolled condition of the main document. However, you could use an Ajax framework to provide such navigational controls instead of using HTML frames. Be aware that creating bookmarks and printing documents in frames can range anywhere from impossible to difficult. Frames can be a problem for folks with disabilities. Most web designers provide an alternative flat HTML document with links for those types of situations.

Creating frames

The task of defining frames in a document remains the same whether or not you're using JavaScript. The simplest framesetting document consists of tags that are devoted to setting up the frameset, as follows:

```
<html>
    <head>
      <title>My Frameset</title>
    </head>
<frameset>
    <frame name="Frame1" src="document1.html">
    <frame name="Frame2" src="document2.html">
</frameset>
</html>
```

The preceding HTML document, which the user never sees, defines the frameset for the entire browser window. Each frame must have a URL reference (specified by the `src` attribute) for a document to load into that frame. Assigning a name to each frame with the `name` attribute greatly simplifies the scripting of frame content.

The frame object model

Perhaps the key to successful frame scripting is understanding that the object model in the browser's memory at any given instant is determined by the HTML tags in the currently loaded documents. All canned object model graphics in this book, such as Figure 27-1, do not reflect the precise object model for your document or document set.

For a single, frameless document, the object model starts with just one `window` object, which contains one document, as shown in Figure 27-1. In this simple structure, the `window` object is the starting point for all references to any loaded object. Because the window is always there — it must be there for a document to load into — a reference to any object in the document can omit a reference to the current window.

In a simple two-framed frameset model (see Figure 27-2), the browser treats the container of the initial, framesetting, document as the parent window. The only visible evidence that the document exists is the framesetting document's title in the browser window title bar.

FIGURE 27-1

The simplest window-document relationship.

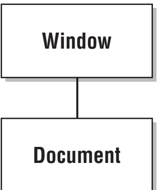
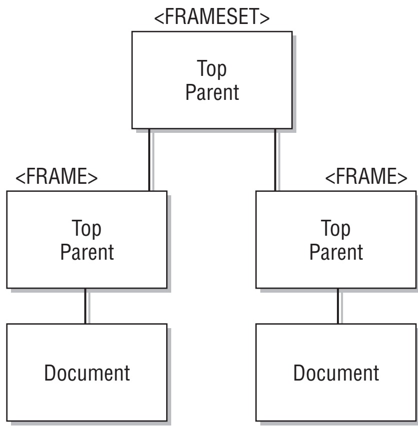


FIGURE 27-2

The parent and frames are part of the object model.

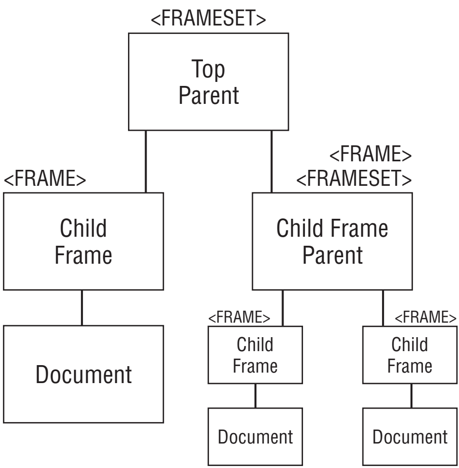


Each `<frame>` tag inside the `<frameset>` tag set creates another `window` object into which a document is loaded. Each of those frames, then, has a `document` object associated with it. From the point of view of a given document, it has a single window container, just as in the model shown in Figure 27-1. And although the `parent` object is not visible to the user, it remains in the object model in memory. The presence of the parent often makes it a convenient repository for variable data that needs to be shared by multiple child frames, or that must persist between loading different documents inside a child frame.

In even more complex arrangements, as shown in Figure 27-3, a child frame itself may load a framesetting document. In this situation, the difference between the `parent` and `top` objects starts to come into focus. The top window is the only one in common with all frames in Figure 27-3. As you see in a moment, when frames need to communicate with other frames (and their documents), you must fashion references to the distant object via the `window` object that they all have in common.

FIGURE 27-3

Three generations of window objects.



Referencing frames

The purpose of an object reference is to help JavaScript locate the desired object in the object model currently held in memory. A reference is a road map for the browser to follow so that it can track down, say, the value of a particular text field in a particular document. Therefore, when you construct a reference, think about where the script appears in the object model and how the reference can help the browser determine where it should go to find the distant object. In a two-generation scenario, such as the one shown in Figure 27-2, three intergenerational references are possible:

- Parent-to-child
- Child-to-parent
- Child-to-child

Assuming that you need to access an object, function, or variable in the relative's frame, the following are the corresponding reference structures: `frameName.objFuncVarName`, `parent.objFuncVarName`. and `parent.frameName.objFuncVarName`.

The rule is this: Whenever a reference must point to another frame, begin the reference with the `window` object that the two destinations have in common. To demonstrate that rule on the complex model in Figure 27-3, if the left child frame's document needs to reference the document at the bottom right of the map, the reference structure is

```
top.frameName.frameName.document. ...
```

Follow the map from the top `window` object, down through two frames, to the final document. JavaScript has to take this route, so your reference must help it along.

Top versus parent

After seeing the previous object maps and reference examples, you may be wondering, Why not use top as the leading object in all transframe references? From an object model point of view, you'll have no problem doing that: A parent in a two-generation scenario is also the top window. What you can't count on, however, is your framesetting document's always being the top window object in someone's browser. Take the instance where a web site loads other web sites into one of its frames. At that instant, the top window object belongs to someone else. If you always specify top in references intended just for your parent window, your references won't work, and will probably lead to script errors for the user. Our advice, then, is to use parent in references whenever you mean one generation above the current document.

Preventing framing

You can use your knowledge of top and parent references to prevent your pages from being displayed inside another web site's frameset. Your top-level document must check whether it is loaded into its own top, or parent, window. When a document is in its own top window, a reference to the top property of the current window is equal to a reference to the current window (the window synonym self seems most grammatically fitting here). If the two values are not equal, you can script your document to reload itself as a top-level document. When it is critical that your document be a top-level document, include the script in Listing 27-1 in the head portion of your document:

LISTING 27-1

Preventing a Document from Getting Framed

```
<script type="text/javascript">
   if (top != self)
   {
      top.location = location;
   }
</script>
```

Your document may appear momentarily inside the other site's frameset, but then the slate is wiped clean, and your top-level document rules the browser window.

Ensuring framing

When you design a web application around a frameset, you may want to make sure that a page always loads the complete frameset. Consider the possibility that a visitor adds only one of your frames to a bookmarks list. On the next visit, only the bookmarked page appears in the browser, without your frameset, which may contain valuable navigation aids to the site.

A script can make sure that a page always loads into its frameset by comparing the URLs of the top and self windows. If the URLs are the same, it means that the page needs to load the frameset. Listing 27-2 shows the simplest version of this technique, which loads a fixed frameset. For a more complete implementation that passes a parameter to the frameset so that it opens a specific page in one of the frames, see the location.search property in Chapter 28, "Location and History Objects."

LISTING 27-2

Forcing a Frameset to Load

```
<script type="text/javascript">
   if (top.location.href == window.location.href)
   {
      top.location.href = " myFrameset.html";
   }
</script>
```

Switching from frames to frameless

Some sites load themselves in a frameset by default, and offer users the option of getting rid of the frames. Modern browsers let you modify a frameset's `cols` or `rows` properties on the fly to simulate adding or removing frames from the current view (see the `frameset` element object later in this chapter). Legacy browsers, on the other hand, don't allow you to change the makeup of a frameset dynamically after it has loaded, but you can load the content page of the frameset into the main window. The workaround for older browsers is to include a button or link whose action loads that document into the `top` window object:

```
top.location.href = "mainBody.html";
```

A switch back to the frame version entails nothing more complicated than loading the framesetting document.

Inheritance versus containment

Scripters who have experience in object-oriented programming environments probably expect frames to inherit properties, methods, functions, and variables, defined in a parent object. That's not the case with scriptable browsers. You can, however, still access those parent items when you make a call to the item with a complete reference to the parent. For example, if you want to define a deferred function in the framesetting parent document that all frames can share, the scripts in the frames refer to that function with this reference:

```
parent.myFunc()
```

You can pass arguments to such functions and expect returned values.

Frame synchronization

A pesky problem for some scripters' plans is that including immediate scripts in the framesetting document is dangerous. Such scripts tend to rely on the presence of documents in the frames being created by this framesetting document. But if the frames have not yet been created, and their documents have not yet loaded, the immediate scripts will likely crash and burn.

One way to guard against this problem is to trigger all such scripts from the frameset's `onload` event handler. In theory, this handler won't trigger until all documents have successfully loaded into the child frames defined by the frameset. At the same time, be careful with `onload` event handlers in the

documents that go into a frameset's frames. If one of those scripts relies on the presence of a document in another frame (one of its brothers or sisters), you're doomed to eventual failure. Anything coming from a slow network or server to a slow modem can get in the way of other documents loading into frames in the ideal order.

One way to work around these problems is to create a Boolean variable in the parent document to act as a flag for the successful loading of subsidiary frames. When a document loads into a frame, its `onload` event handler can set that flag to `true` to indicate that the document has loaded. Any script that relies on a page being loaded should use an `if` construction to test the value of that flag before proceeding.

It is best to construct the code so that the parent's `onload` event handler triggers all the scripts that you want to run after loading. You should also test your pages thoroughly for any residual effects that may accrue if someone resizes a window or clicks Reload.

Blank frames

Often, you may find it desirable to create a frame in a frameset, but not put any document in it until the user has interacted with various controls or other user interface elements in other frames. Most modern browsers have a somewhat empty document in one of their internal URLs (`about:blank`). However, this URL is not guaranteed to be available on all browsers. If you need a blank frame, let your framesetting document write a generic HTML document to the frame directly from the `src` attribute for the frame, as shown in the skeletal code in Listing 27-3. Loading an empty HTML document requires no additional transactions.

LISTING 27-3

Creating a Blank Frame

```
<html>
   <head>
      <script type="text/javascript">
      <! —
         function blank()
         {
            return "<html></html>";
         }
      // — >
      </script>
   </head>
   <frameset>
      <frame name="Frame1" src="someURL.html">
      <frame name="Frame2" src="javascript:parent.blank()">
   </frameset>
</html>
```

Viewing frame source code

Studying other scripters' work is a major tool for learning JavaScript (or any programming language). With most scriptable browsers, you can easily view the source code for any frame, including those

frames whose content is generated entirely, or in part, by JavaScript. Click the desired frame to activate it (a subtle border may appear just inside the frame on some browser versions, but don't be alarmed if the border doesn't appear). Then select Frame Source (or equivalent) from the View menu (or from the right-click submenu). You can also print or save a selected frame.

Frames versus frame element objects

With the expansion of object models that expose every HTML element to scripting, a terminology conflict comes into play. Everything that you have read about frames thus far in the chapter refers to the original object model, where a frame is just another kind of window with a slightly different referencing approach. That still holds true, even in the latest browsers.

But when the object model also exposes HTML elements, the notion of the `frame` element object is somewhat distinct from the frame object of the original model. The `frame` element object represents an object whose properties are dominated by the attributes you set inside the `<frame>` tag. This provides access to settings, such as the frame border and scrollability — the kinds of properties that are not exposed to the original frame object.

References to the frame and `frame` element objects are also different. You've seen plenty of examples of how to reference an old-fashioned frame earlier in this chapter. But access to a `frame` element object is either via the element's `id` attribute or through the child node relationship of the enclosing `frameset` element (you cannot use the `parentNode` property to back your way out of the current document to the `frame` element that encloses the document). The strategy we prefer is assigning an `id` attribute to `<frame>` tags and accessing the `frame` element object by way of the `document` object that lives in the parent (or top) of the frameset hierarchy. Therefore, to access the `frameBorder` property of a `frame` element object from a script living in any frame of a frameset, the syntax is

```
parent.document.all.frame1ID.frameBorder
```

or (for IE5+/Moz/W3C)

```
parent.document.getElementById("frame1ID").frameBorder
```

When the reference goes through the `frame` element object, you can still reach the document object in that frame via the element object's `contentWindow` or `contentDocument` properties (see the `frame` element object later in this chapter).

window Object

Properties	Methods	Event Handlers
appCore	addEventListener()†	onabort††
clientInformation	alert()	onafterprint
clipboardData	attachEvent()†	onbeforeprint
closed	back()	onbeforeunload

Properties	Methods	Event Handlers
components[]	blur()†	onblur†
content	clearInterval()	onchange††
controllers[]	clearTimeout()	onclick††
crypto	close()	onclose††
defaultStatus	confirm()	onerror
dialogArguments	createPopup()	onfocus†
dialogHeight	detachEvent()†	onhelp
dialogLeft	dispatchEvent()†	onkeydown††
dialogTop	dump()	onkeypress††
dialogWidth	execScript()	onkeyup††
directories	find()	onload
document	fireEvent()†	onmousedown††
event Event	focus()†	onmousemove††
external	forward()	onmouseout††
frameElement	geckoActiveXObject()	onmouseover††
frames[]	getComputedStyle()	onmouseup††
fullScreen	getSelection()	onmove
history	home()	onreset††
innerHeight	moveBy()	onresize
innerWidth	moveTo()	onscroll
length	navigate()	onselect††
location	open()	onsubmit††
locationbar	openDialog()	onunload
menubar	print()	
name	prompt()	
navigator	removeEventListener()†	
netscape	resizeBy()	
offscreenBuffering	resizeTo()	
opener	scroll()	
outerHeight	scrollBy()	

Properties	Methods	Event Handlers
outerWidth	scrollByLines()	
pageXOffset	scrollByPages()	
pageYOffset	scrollTo()	
parent	setActive()†	
personalbar	setInterval()	
pkcs11	setTimeout()	
prompter	showHelp()	
returnValue	showModalDialog()	
screen	showModelessDialog()	
screenLeft	sizeToContent()	
screenTop	stop()	
screenX		
screenY		
scrollbars		
scrollMaxX		
scrollMaxY		
scrollX		
scrollY		
self		
sidebar		
status		
statusbar		
toolbar		
top		
window		

† See Chapter 26, ''Generic HTML Element Objects.''

†† To handle captured or bubbled events of other objects in IE4+ and W3C DOM browsers.

Syntax

Creating a window:

```
var windowObject = window.open([parameters]);
```

Accessing window properties or methods:

```
window.property | method([parameters])

self.property | method([parameters])

windowObject.property | method([parameters])
```

Compatibility: WinIE3+, MacIE3+, NN2+, Moz+, Safari+, Opera+, Chrome+

About this object

The `window` object has the unique position of being at the top of the object hierarchy, encompassing even the almighty `document` object. This exalted position gives the `window` object a number of properties and behaviors unlike those of any other object.

Chief among its unique characteristics is that because everything takes place in a window, you can usually omit the `window` object from object references. You've seen this behavior in previous chapters when we invoked document methods, such as `document.write()`. The complete reference is `window.document.write()`. But because the activity was taking place in the window that held the document running the script, that window was assumed to be part of the reference. For single-frame windows, this concept is simple enough to grasp.

As previously stated, among the list of properties for the `window` object is one called `self`. This property is synonymous with the `window` object itself (which is why it shows up in hierarchy diagrams as an object). Having a property of an object that is the same name as the object may sound confusing, but this situation is not that uncommon in object-oriented environments. We discuss the reasons why you may want to use the `self` property as the window's object reference in the `self` property description that follows.

As indicated earlier in the syntax definition, you don't always have to create a `window` object specifically in JavaScript code. Your browser usually opens a window after you start it up. That window is a valid `window` object, even if the window is blank. Therefore, after a user loads your page into the browser, the `window` object part of that document is automatically created for your script to access as it pleases.

One conceptual trap to avoid is believing that a `window` object's event handler or custom property assignments outlive the document whose scripts make the assignments. Except for some obvious physical properties of a window, each new document that loads into the window starts with a clean slate of window properties and event handlers.

Your script's control over an existing (already open) window's user interface elements varies widely with the browser and browser version for which your application is intended. Before the version 4 browsers, the only change you could make to an open window was to the status line at the bottom of the browser window. Version 4 browsers added the ability to control such properties as the size, location, and (with signed scripts in Navigator and Mozilla) the presence of chrome elements (toolbars and scroll bars, for example) on the fly. Many of these properties can be changed beyond specific safe limits only if you cryptographically sign the scripts (see Chapter 49, "Security and Netscape Signed Scripts," on the CD-ROM) and/or the user grants permission for your scripts to make those modifications.

Window properties are far more flexible on all browsers when your scripts generate a new window (with the `window.open()` method): You can influence the size, toolbar, or other view options of a

window. Recent browser versions provide even more options for new windows, including the position of the window and whether the window should display a title bar. Again, if an option can conceivably be used to deceive a user (for example, silently hiding one window that monitors activity in another window), signed scripts and/or user permission are necessary.

The `window` object is also the level at which a script asks the browser to display any of three styles of dialog boxes (a plain alert dialog box, an OK/Cancel confirmation dialog box, or a prompt for user text entry). Although dialog boxes are extremely helpful for cobbling together debugging tools for your own use (see Chapter 48, "Debugging Scripts," on the CD-ROM), they can be very disruptive to visitors who navigate through web sites. Because most JavaScript dialog boxes are modal (that is, you cannot do anything else in the browser until you dismiss the dialog box), use them sparingly, if at all. Remember that some users may create macros on their computers to visit sites unattended. Should such an automated visitor to your site encounter a modal dialog box, it is trapped on your page until a human intervenes.

All dialog boxes generated by JavaScript identify themselves as being generated by JavaScript. This is primarily a security feature to prevent deceitful scripts from creating system- or application-style dialog boxes that convince visitors to enter private information. It should also discourage dialog box usage in web-page design. And that's good, because dialog boxes tend to annoy users.

With the exception of the IE- and Safari-specific modal, and IE-specific modeless, dialog boxes (see the `window.showModalDialog()` and `window.showModeless()` methods), JavaScript dialog boxes are not particularly flexible in letting you fill them with text or graphic elements beyond the basics. In fact, you can't even change the text of the dialog-box buttons or add buttons. With Dynamic HTML (DHTML)-capable browsers, you can use positioned `div` or `iframe` elements to simulate dialog-box behavior in a cross-browser way.

With respect to the W3C DOM, the window is outside the scope of the standard through DOM Level 2. The closest that the standard comes to acknowledging a window at all is the `document.defaultView` property, which evaluates to the `window` object in today's browsers (predominantly Mozilla). But the formal DOM standard specifies no properties or methods for this view object.

Properties

appCore
components[]
content
controllers[]
prompter
sidebar

Values: (See text) Read-Only

Compatibility: WinIE-, MacIE-, NN6+, Moz+, Safari-, Opera-, Chrome-

NN6+/Mozilla provide scriptable access to numerous services that are part of the xpconnect package (*xp* stands for *cross-platform*), which is part of the larger NPAPI (Netscape Plugin Application Programming Interface). The xpconnect services allow scripts to work with COM objects and the mozilla.org XUL (XML-based User Interface Language) facilities — lengthy subjects that extend well beyond the scope of this book. You can begin to explore this subject within the context of Mozilla-based browsers and scripting at `http://www.mozilla.org/scriptable/`.

clientInformation

Value: navigator object Read-Only

Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari 1.2+, Opera-, Chrome+

In an effort to provide scriptable access to browser-level properties while avoiding reference to the Navigator browser brand, Microsoft created the clientInformation property. Its value is identical to that of the navigator object — an object name that is also available in IE. Although Safari 1.2 adopted usage of the clientInformation property, you should use the navigator object for cross-browser applications. (See Chapter 42, "The Navigator and Other Environment Objects," on the CD-ROM.)

Related Item: navigator object

clipboardData

Value: Object Read/Write

Compatibility: WinIE5+, MacIE-, NN-, Moz-, Safari-, Opera-, Chrome-

Use the clipboardData object to transfer data for such actions as cutting, copying, and pasting under script control. The object contains data of one or more data types associated with a transfer operation. Use this property only when editing processes via the Edit menu (or keyboard equivalents) or the context menu controlled by script — typically in concert with edit-related event handlers.

Working with the clipboardData object requires knowing about its three methods, shown in Table 27-1. Familiarity with the edit-related event handlers (before and after versions of cut, copy, and paste) is also helpful (see Chapter 26).

TABLE 27-1

window.clipboardData Object Methods

Method	Returns	Description
clearData([*format*])	Nothing	Removes data from the clipboard. If no format parameter is supplied, all data is cleared. Data formats can be one or more of the following strings: Text, URL, File, HTML, Image.
getData(*format*)	String	Retrieves data of the specified format from the clipboard. The format is one of the following strings: Text, URL, File, HTML, Image. The clipboard is not emptied when you get the data so that the data can be retrieved in several sequential operations.
setData(*format*, *data*)	Boolean	Stores string data in the clipboard. The format is one of the following strings: Text, URL, File, HTML, Image. For nontext data formats, the data must be a string that specifies the path or URL to the content. Returns true if the transfer to the clipboard is successful.

You cannot use the clipboardData object to transfer data between pages that originate from different domains or arrive via different protocols (http versus https).

Example

See Listings 26-36 and 26-44 in Chapter 26, "Generic HTML Element Objects," to see how the `clipboardData` object is used with a variety of edit-related event handlers.

Related Items: `event.dataTransfer` property; `onbeforecopy`, `onBeforeCut`, `onbeforepaste`, `oncopy`, `oncut`, `onpaste` event handlers

closed

Value: Boolean Read-Only

Compatibility: WinIE4+, MacIE4+, NN3+, Moz+, Safari+, Opera+, Chrome+

When you create a subwindow with the `window.open()` method, you may need to access object properties from that subwindow, such as setting the value of a text field. Access to the subwindow is via the `window` object reference that is returned by the `window.open()` method, as in the following code fragment:

```
var newWind = window.open("someURL.html","subWind");
...
newWind.document.entryForm.ZIP.value = "00000";
```

In this example, the `newWind` variable is not linked live to the window, but is only a reference to that window. If the user should close the window, the `newWind` variable still contains the reference to the now-missing window. Thus, any script reference to an object in that missing window will likely cause a script error. What you need to know before accessing items in a subwindow is whether the window is still open.

The `closed` property returns `true` if the `window` object has been closed, either by script or by the user. Any time you have a script statement that can be triggered after the user has an opportunity to close the window, test for the `closed` property before executing that statement.

Example

Listing 27-4 is a basic window-opening and window-closing example. The script begins by initializing a global variable, `newWind`, which is used to hold the object reference to the second window. This value needs to be global so that other functions can reference the window for tasks, such as closing.

For this example, the new window contains some HTML code written dynamically to it, rather than loading an existing HTML file into it. Therefore, the URL parameter of the `window.open()` method is left as an empty string. Next comes a brief delay to allow Internet Explorer (especially versions 3 and 4) to catch up with opening the window, so that content can be written to it. The delay (using the `setTimeout()` method described later in this chapter) invokes the `finishNewWindow()` function, which uses the global `newWind` variable to reference the window for writing. The `document.close()` method closes writing to the document — a different kind of close from a window close. A separate function, `closeWindow()`, is responsible for closing the subwindow.

As a final test, an `if` condition looks at two conditions: (1) whether the `window` object has ever been initialized with a value other than `null` (in case you click the window-closing button before ever having created the new window) and (2) whether the window's `closed` property is `null` or `false`. If either condition is true, the `close()` method is sent to the second window.

Note

This is a simple example, so the property assignment event-handling technique employed uses the more modern approach of binding events, using the `addEventListener()` (NN6+/Moz/W3C) or `attachEvent()` (IE5+) methods. This modern, cross-browser event-handling technique is explained in detail in Chapter 32, "Event Objects." The event-handling technique employed throughout most of the code in this chapter and much of the book is a deliberate simplification to make the code more readable. It is generally better to use the more modern approach of binding events, as we do in this example. ■

LISTING 27-4

Checking Before Closing a Window

```
HTML: jsb27-04.html

<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>window.closed Property</title>
      <script type="text/javascript" src="../jsb-global.js"></script>
      <script type="text/javascript" src="jsb27-04.js"></script>
   </head>
   <body>
      <h1>window.closed Property</h1>
      <hr />
      <p>The new window might open behind this window.</p>
      <form id="newWinForm" name="newWinForm">
         <input id="openNewWin" name="openNewWin" type="button"
                value="Open Window" />
         <br />
         <input id="closeNewWin" name="closeNewWin" type="button"
                value="Close it if Still Open" />
      </form>
   </body>
</html>

JavaScript: jsb27-04.js

// initialize when the page has loaded
addEvent(window, "load", initialize);

// global variables
   var oNewWinForm;
   var oOpenNewWin;
   var oCloseNewWin;

// initialize global var for new window object
// so it can be accessed by all functions on the page
var newWind;
```

continued

LISTING 27-4 *(continued)*

```
function initialize ()
{
   // get IE4+ or W3C DOM reference for an ID
   if (document.getElementById)
   {
      oNewWinForm = document.getElementById("newWinForm");
      oOpenNewWin = document.getElementById("openNewWin");
      oCloseNewWin = document.getElementById("closeNewWin");
   }
   else
   {
      oNewWinForm = document.newWinForm;
      oOpenNewWin = document.oNewWinForm.openNewWin;
      oCloseNewWin = document.oNewWinForm.closeNewWin;
   }
   addEvent(oOpenNewWin, 'click', newWindow);
   addEvent(oCloseNewWin, 'click', closeWindow);
}

// make the new window and put some stuff in it
function newWindow()
{
   newWind = window.open("","subwindow","height=200,width=200");
   setTimeout("finishNewWindow()", 100);
}
function finishNewWindow()
{
   var output = "";
   output += "<html><body><h1>A Sub-window</h1>";
   output += "<form><input type='button' value='Close Main Window'";
   output +="onclick='window.opener.close()'></form></body></html>";
   newWind.document.write(output);
   newWind.document.close();
}

// close subwindow
function closeWindow()
{
   if (newWind && !newWind.closed)
   {
      newWind.close();
   }
   else
   {
      alert("The window is no longer open");
   }
}
```

To complete the example of the window opening and closing, notice that the subwindow is given a button whose `onclick` event handler (bound in the `initialization()` function) closes the main window/tab. In some modern browsers, the user is presented with an alert asking to confirm the closure of the main browser window/tab. Other browsers simply ignore the request with, in some cases, a warning in the error console. Safari closes the main window/tab without any warning.

Related Items: `window.open()`, `window.close()` methods

components

(See `appCore`)

controllers

(See `appCore`)

crypto
pkcs11

Values: Object references Read-Only

Compatibility: WinIE-, MacIE-, NN6+, Moz+, Safari-, Opera-, Chrome-

The `crypto` and `pkcs11` properties return references to browser objects that are relevant to internal public-key cryptography mechanisms. These subjects are beyond the scope of this book, but you can read more about Netscape's efforts on this front at `http://www.mozilla.org/projects/security/`.

defaultStatus

Value: String Read/Write

Compatibility: WinIE3+, MacIE3+, NN2+, Moz+, Safari+, Opera+, Chrome-

After a document is loaded into a window or frame, the status bar's message field can display a string that is visible any time the mouse pointer is not atop an object that takes precedence over the status bar (such as a link object or an image map). The `window.defaultStatus` property is normally an empty string, but you can set this property at any time. Any setting of this property will be temporarily overridden when a user moves the mouse pointer on top of a link object (see `window.status` property for information about customizing this temporary status-bar message).

Probably the most common time to set the `window.defaultStatus` property is when a document loads into a window. You can do this as an immediate script statement that executes from the `head` or `body` portion of the document or as part of a document's `onload` event handler.

Example

Unless you plan to change the default status-bar text while a user spends time at your web page, the best time to set the property is when the document loads. In Listing 27-5, notice that we also read this property to reset the status bar in an `onmouseout` event handler. Setting the `status` property to empty also resets the status bar to the `defaultStatus` setting. When testing, remember that even though modern browsers support the `status` property, some of them do not display scripted status-bar text associated with links, in order to prevent link spoofing.

LISTING 27-5

Setting the Default Status Message

```
<!DOCTYPE html>
<html>
   <head>
     <meta http-equiv="content-type" content="text/html;charset=utf-8">
     <title>window.defaultStatus property</title>
     <script type="text/javascript">
        window.defaultStatus = "Welcome to my Web site.";
     </script>
   </head>
   <body>
     <h1>window.defaultStatus property</h1>
     <hr />
     <p><a href="http://www.microsoft.com"
         onmouseover="window.status = 'Visit Microsoft\'s Home page.'; return true;"
         onmouseout="window.status = '';return true;">Microsoft</a>
     </p>
     <p><a href="http://mozilla.org"
         onmouseover="window.status = 'Visit Mozilla\'s Home page.'; return true;"
         onmouseout="window.status = window.defaultStatus;return true;">Mozilla</a>
     </p>
   </body>
</html>
```

If you need to display single or double quotes in the status bar (as in both links in Listing 27-5), use escape sequences (`\'` and `\"`) as part of the strings being assigned to these properties.

Related Item: `window.status` property

dialogArguments

Value: Varies Read-Only

Compatibility: WinIE4+, MacIE4+, NN-, Moz+, Safari+, Opera-, Chrome+

The `dialogArguments` property is available only in a window that is generated by the `showModalDialog()` or the IE-specific `showModelessDialog()` methods. Those methods allow a parameter to be passed to the dialog-box window, and the `dialogArguments` property lets the dialog box window's scripts access that parameter value. The value can be in the form of a string, number, or JavaScript array (convenient for passing multiple values).

Example

See Listing 27-36 for the `window.showModalDialog()` method, to see how arguments can be passed to a dialog box and retrieved via the `dialogArguments` property.

Related Items: `window.showModalDialog()`, `window.showModelessDialog()` methods

dialogHeight
dialogWidth

Value: String Read/Write

Compatibility: WinIE4+, MacIE-, NN-, Moz-, Safari-, Opera-, Chrome-

Scripts in a document located inside a modal or IE-specific modeless dialog box (generated by `showModalDialog()` or `showModelessDialog()`) can read or modify the height and width of the dialog-box window via the `dialogHeight` and `dialogWidth` properties. Scripts can access these properties from the main window only for modeless dialog boxes, which remain visible while the user can control the main window contents.

Values for these properties are strings and include the unit of measure, the pixel (`px`).

Example

Dialog boxes sometimes provide a button or icon that reveals more details or more complex settings for advanced users. You can create a function that handles the toggle between two sizes. The following function assumes that the document in the dialog box has a button whose label also toggles between Show Details and Hide Details. The button's `onclick` event handler invokes the function as `toggleDetails(this)`:

```
function toggleDetails(btn)
{
   if (dialogHeight == "200px")
   {
      dialogHeight = "350px";
      btn.value = "Hide Details";
   }
   else
   {
      dialogHeight = "200px";
      btn.value = "Show Details";
   }
}
```

In practice, you also have to toggle the `display` style sheet property of the extra material between `none` and `block` to make sure that the dialog box does not display scroll bars in the smaller dialog-box version.

Related Items: `window.dialogLeft`, `window.dialogTop` properties

dialogLeft
dialogTop

Value: String Read/Write

Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-, Opera-, Chrome-

Scripts in a document located inside a modal or IE-specific modeless dialog box (generated by `showModalDialog()` or `showModelessDialog()`) can read or modify the left and top coordinates of the dialog-box window via the `dialogLeft` and `dialogTop` properties. Scripts can access these

properties from the main window only for modeless dialog boxes, which remain visible while the user can control the main window contents.

Values for these properties are strings and include the unit of measure, the pixel (`px`). If you attempt to change these values so that any part of the dialog-box window would be outside the video monitor, the browser overrides the settings to keep the entire window visible.

Example

Although usually not a good idea because of the potentially jarring effect on a user, you can reposition a dialog-box window that has been resized by script (or by the user if you let the dialog box be resizable). The following statements in a dialog-box window document's script re-center the dialog-box window:

```
dialogLeft = (screen.availWidth/2) - (parseInt(dialogWidth)/2) + "px";
dialogHeight = (screen.availHeight/2) - (parseInt(dialogHeight)/2) + "px";
```

Note that the `parseInt()` functions are used to read the numeric portion of the `dialogWidth` and `dialogHeight` properties so that the values can be used for arithmetic.

Related Items: `window.dialogHeight`, `window.dialogTopWidth` properties

directories
locationbar
menubar
personalbar
scrollbars
statusbar
toolbar

Value: Object Read/Write (with signed scripts)

Compatibility: WinIE-, MacIE-, NN4+, Moz+, Safari-, Opera-, Chrome-

Beyond the rectangle of the content region of a window (where your documents appear), the Netscape browser window displays an amalgam of bars and other features known collectively as *chrome*. All browsers can elect to remove these chrome items when creating a new window (as part of the third parameter of the `window.open()` method), but until signed scripts were available in Navigator 4, these items could not be turned on and off in the main browser window, or in any existing window.

Navigator 4 promoted these elements to first-class objects contained by the `window` object. Navigator 6 added one more feature, called the *directories bar* — a framelike device that can be opened or hidden from the left edge of the browser window. At the same time, however, NN6+/Mozilla browsers no longer permit hiding and showing the browser window's scroll bars. Chrome objects have but one property: `visible`. Reading this Boolean value (possible without signed scripts) lets you inspect the visitor's browser window for the elements currently engaged.

Changing the visibility of these items on the fly alters the relationship between the inner and outer dimensions of the browser window. If you must carefully size a window to display content, you should adjust the chrome elements before sizing the window. But before you start changing chrome visibility before the eyes of your page visitors, weigh the decision carefully. Experienced users have fine-tuned the look of their browser windows to just the way they like them. If you mess with that look, you may anger your visitors. Fortunately, changes you make to a chrome element's visibility are not stored to the user's preferences. However, the changes you make survive an unloading of the

page. If you change the settings, be sure that you first save the initial settings and restore them with an `onunload` event handler.

Tip

The Macintosh menu bar is not part of the browser's window chrome. Therefore, its visibility cannot be adjusted from a script. ■

Example

In Listing 27-6, you can experiment with the look of a browser window with any of the chrome elements turned on or off. To run this script, you must either sign the scripts or turn on codebase principals (see Chapter 49, "Security and Netscape Signed Scripts," on the CD-ROM). Java must also be enabled to use the signed script statements.

As the page loads, it stores the current state of each chrome element. One button for each chrome element triggers the `toggleBar()` function. This function inverts the visible property for the chrome object passed as a parameter to the function. Finally, the Restore button returns the visibility properties to their original settings. Notice that the `restore()` function is also called by the `onunload` event handler for the document.

LISTING 27-6

Controlling Window Chrome

HTML: jsb27-06.html

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Bars Bars Bars</title>
      <script type="text/javascript" src="jsb27-06.js"></script>
   </head>
   <body onunload="restore()">
      <h1>Bars Bars Bars</h1>
      <hr />
      <form>
         <b>Toggle Window Bars</b>
         <br />
         <input type="button" value="Location Bar"
                onclick="toggleBar(window.locationbar)" />
         <br />
         <input type="button" value="Menu Bar"
                onclick="toggleBar(window.menubar)" />
         <br />
         <input type="button" value="Personal Bar"
                onclick="toggleBar(window.personalbar)" />
         <br />
         <input type="button" value="Scrollbars"
                onclick="toggleBar(window.scrollbars)" />
         <br />
         <input type="button" value="Status Bar"
```

continued

LISTING 27-6 *(continued)*

```
                  onclick="toggleBar(window.statusbar)" />
         <br />
         <input type="button" value="Tool Bar"
                  onclick="toggleBar(window.toolbar)" />
         <br />
         <hr />
         <input type="button" value="Restore Original Settings"
                  onclick="restore()" />
      </form>
   </body>
</html>
```

JavaScript: jsb27-06.js

```
// store original outer dimensions as page loads
var originalLocationbar = window.locationbar.visible;
var originalMenubar = window.menubar.visible;
var originalPersonalbar = window.personalbar.visible;
var originalScrollbars = window.scrollbars.visible;
var originalStatusbar = window.statusbar.visible;
var originalToolbar = window.toolbar.visible;

// generic function to set inner dimensions
function toggleBar(bar)
{
   netscape.security.PrivilegeManager.enablePrivilege("UniversalBrowserWrite");
   bar.visible = !bar.visible;
   netscape.security.PrivilegeManager.revertPrivilege("UniversalBrowserWrite");
}
// restore settings
function restore()
{
   netscape.security.PrivilegeManager.enablePrivilege("UniversalBrowserWrite");
   window.locationbar.visible = originalLocationbar;
   window.menubar.visible = originalMenubar;
   window.personalbar.visible = originalPersonalbar;
   window.scrollbars.visible = originalScrollbars;
   window.statusbar.visible = originalStatusbar;
   window.toolbar.visible = originalToolbar;
   netscape.security.PrivilegeManager.revertPrivilege("UniversalBrowserWrite");
}
```

Related Item: `window.open()` method

document

Value: Object Read-Only

Compatibility: WinIE3+, MacIE3+, NN2+, Moz+, Safari+, Opera+, Chrome+

We list the `document` property here primarily for completeness. Each `window` object contains a single `document` object. The value of the `document` property is the `document` object, which is not a displayable value. Instead, you use the `document` property as you build references to properties and methods of the document and to other objects contained by the document, such as a form and its elements. To load a different document into a window, use the `location` object (see Chapter 28, "Location and History Objects"). The `document` object is described in detail in Chapter 29, "Document and Body Objects."

Related Item: `document` object

event
Event

Value: Object Read/Write

Compatibility: WinIE4+, MacIE4+, NN+, Moz+, Safari 1+, Opera+, Chrome+

IE4+ treats the `event` object as a property of the `window` object. The W3C DOM passes an instance of the `Event` object as an argument to event handler functions. The connection with the `window` object is relatively inconsequential because all action involving the `event` object occurs in event handler functions. The only difference is that the object can be treated as a more global object when one event handler function invokes another. Instead of having to pass the `event` object parameter to the next function, functions can access the `event` object directly (with or without the `window.` prefix in the reference).

For complete details about the `event` object in all browsers, see Chapter 32, "Event Objects."

Related Item: `event` object

external

Value: Object Read-Only

Compatibility: WinIE4+, MacIE-, NN-, Moz+, Safari-, Opera-, Chrome-

The `external` property is useful only when the browser window is a component in another application. The property provides a gateway between the current browser window and the application that acts as a host to the browser window component.

With WinIE4+ acting as a component to the host operating system, the `external` property can be used to access several methods that influence behaviors outside the browser. Perhaps the three most useful methods to regular web-page scripters are `AddDesktopComponent()`, `AddFavorite()`, and `NavigateAndFind()`. The first two methods display the same kind of alert dialog box that users get after making these choices from the browser or desktop menus, so that you won't be able to sneak your web site onto desktops or Favorites listings without the visitor's approval. IE7 and Firefox2+ allow `AddSearchProvider()` for installing search plug-ins. As with the first two methods, an alert is displayed asking the user permission to install the search plug-in. You can find information about OpenSearch and the format for the XML file passed to the method at `http://www.opensearch.org/Specifications/OpenSearch/1.1`. Table 27-2 describes the parameters for these methods.

TABLE 27-2

Popular window.external Object Methods

Method	Description
`AddDesktopComponent("URL", "type"[, left, top, width, height])`	Adds a web site or image to the Active Desktop (if turned on in the user's copy of Windows). The `type` parameter value is either `website` or `image`. Dimensional parameters (optional) are all integer values.
`AddFavorite("URL"[, "title"])`	Adds the specified URL to the user's Favorites list. The optional title string parameter is how the URL should be listed in the menu (if missing, the URL appears in the list).
`AddSearchProvider("URL")`	Adds the search provider, specified in the URL with an `OpenSearchDescription` XML file.
`NavigateAndFind("URL", "findString", "target")`	Navigates to the URL in the first parameter and opens the page in the target frame (an empty string opens in the current frame). The `findString` is text to be searched for on that page and highlighted when the page loads.

Example

The first example asks the user whether it is okay to add a web site to the Active Desktop. If Active Desktop is not enabled, the user is given the choice of enabling it at this point:

```
external.AddDesktopComponent("http://www.nytimes.com","website", 200,
      100, 400, 400);
```

In the next example, the user is asked to approve the addition of a URL to the Favorites list. The user can follow the normal procedure for filing the item in a folder in the list:

```
external.AddFavorite("http://www.dannyg.com/support/update13.html",
   "JSBible 6 Support Center");
```

The final example assumes that a user makes a choice from a `select` list of items. The `onchange` event handler of the `select` list invokes the following function to navigate to a fictitious page and locate listings for a chosen sports team on the page:

```
function locate(list)
{
   var choice = list.options[list.selectedIndex].value;
   external.NavigateAndFind("http://www.collegesports.net/scores.html",
      choice, "scores");
}
```

frameElement

Values: `frame` or `iframe` object reference   Read-Only

Compatibility: WinIE5.5+, MacIE-, NN7+, Moz1.0.1+, Safari 1.2+, Opera+, Chrome+

If the current window exists as a result of a `<frame>` or `<iframe>` tag, the window's `frameElement` property returns a reference to the hosting element. As is made clear in the discussion about the `frame` element object later in this chapter, a reference to a `frame` or `iframe` element object provides access to the properties that echo the attributes of the HTML element object. For a window that is not part of a frameset, the `frameElement` property returns `null`.

The convenience of this property becomes apparent when a single document is loaded into multiple framesets. A script in the document can still refer to the containing `frame` element, even when the ID of the element changes from one frameset to another. The `frameset` element is also accessible via the `parentElement` property of the `frameElement` property:

```
var frameSetObj = self.frameElement.parentElement;
```

A reference to the `frameset` element opens possibilities for adjusting frame sizes.

Related Items: `frame`, `iframe` objects

frames

Value: Array Read-Only

Compatibility: WinIE3+, MacIE3+, NN2+, Moz+, Safari+, Opera+, Chrome+

In a multiframe window, the top or parent window contains any number of separate frames, each of which acts as a full-fledged `window` object. The `frames` property (note the plural use of the word as a property name) plays a role when a statement must reference an object located in a different frame. For example, if a button in one frame is scripted to load a document in another frame, the button's event handler must be able to tell JavaScript precisely where to display the new HTML document. The `frames` property assists in that task.

To use the `frames` property to communicate from one frame to another, it should be part of a reference that begins with the `parent` or `top` property. This lets JavaScript make the proper journey through the hierarchy of all currently loaded objects to reach the desired object. To find out how many frames are currently active in a window, use this expression:

```
parent.frames.length
```

This expression returns a number indicating how many frames the parent window defines. This value does not, however, count further nested frames, should a third generation of frame be defined in the environment. In other words, no single property exists that you can use to determine the total number of frames in the browser window if multiple generations of frames are present.

The browser stores information about all visible frames in a numbered (indexed) array, with the first frame (that is, the topmost `<frame>` tag defined in the framesetting document) as number `0`:

```
parent.frames[0]
```

Therefore, if the window shows three frames (whose indexes are `frames[0]`, `frames[1]`, and `frames[2]`, respectively), the reference for retrieving the `title` property of the document in the second frame is

```
parent.frames[1].document.title
```

This reference is a road map that starts at the parent window and extends to the second frame's document and its `title` property. Other than the number of frames defined in a parent window and each

frame's name (`top.frames[i].name`), no values from the frame definitions are directly available from the frame object via scripting until you get to IE4 and NN6/Moz/W3C (see the `frame` element object later in this chapter). In these browsers, individual `frame` element objects have several properties that reveal `<frame>` tag attributes.

Using index values for frame references is not always the safest tactic, however, because your frameset design may change over time, in which case the index values will also change. Instead, you should take advantage of the `name` attribute of the `<frame>` tag, and assign a unique, descriptive name to each frame. A value you assign to the `name` attribute is also the name that you use for `target` attributes of links in order to force a linked page to load in a frame other than the one containing the link. You can use a frame's name as an alternative to the indexed reference. For example, in Listing 27-7, two frames are assigned distinctive names. To access the title of a document in the `JustAKid2` frame, the complete object reference is

```
parent.JustAKid2.document.title
```

with the frame name (case sensitive) substituting for the `frames[1]` array reference. Or, in keeping with JavaScript flexibility, you can use the object name in the array index position:

```
parent.frames["JustAKid2"].document.title
```

The supreme advantage to using frame names in references is that no matter how the frameset structure may change over time, a reference to a named frame will always find that frame, even though its index value (that is, its position in the frameset) may change.

Example

Listing 27-7 and Listing 27-8 demonstrate how JavaScript treats values of frame references from objects inside a frame. The same document is loaded into each frame. A script in that document extracts information about the current frame and the entire frameset.

LISTING 27-7

Framesetting Document for Listing 27-8

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
<title>window.frames property</title>
   </head>
   <frameset cols="50%,50%">
      <frame name="JustAKid1" src="jsb27-08.html" />
      <frame name="JustAKid2" src="jsb27-08.html" />
   </frameset>
</html>
```

A call to determine the number (length) of frames returns `0`, from the point of view of the current frame referenced. That's because each frame here is a window that has no nested frames within it. But add the `parent` property to the reference, and the scope zooms out to take into account all frames generated by the parent window's document.

LISTING 27-8

Showing Various Window Properties

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Window Revealer II</title>
      <script type="text/javascript">
         function gatherWindowData()
         {
            var msg = "";
            msg += "<p><b>From the point of view of this frame:</b><br />";
            msg += "window.frames.length: " + window.frames.length + "<br />";
            msg += "window.name: " + window.name + "</p>";
            msg += "<p><b>From the point of view of the framesetting ↩
                          document:</b><br />";
            msg += "parent.frames.length: " + parent.frames.length + "<br />";
            msg += "parent.frames[0].name: " + parent.frames[0].name + "</p>";
            return msg;
         }
      </script>
   </head>
   <body>
      <script type="text/javascript">
         document.write(gatherWindowData());
      </script>
   </body>
</html>
```

The last statement in the example shows how to use the array syntax (brackets) to refer to a specific frame. All array indexes start with `0` for the first entry. Because the document asks for the name of the first frame (`parent.frames[0]`), the response is `JustAKid1` for both frames.

Related Items: `frame`, `frameset` objects; `window.parent`, `window.top` properties

fullScreen

Values: Boolean Read-Only

Compatibility: WinIE-, MacIE-, NN7.1+, Moz1.4+, Safari-, Opera-, Chrome-

The intent of the `fullScreen` property is to indicate whether the browser is in full-screen mode, which can be set in Mozilla browsers through the Full Screen command in the View menu. Unfortunately, the property isn't reliable (as of Mozilla 1.8.1) and always returns `false`, regardless of the browser's actual full-screen setting.

history

Value: Object Read-Only

Compatibility: WinIE3+, MacIE3+, NN2+, Moz+, Safari+, Opera+, Chrome+

(See the discussion of the `history` object in Chapter 28, “Location and History Objects.”)

innerHeight
innerWidth
outerHeight
outerWidth

Value: Integer Read/Write (see text)

Compatibility: WinIE-, MacIE-, NN4+, Moz+, Safari+, Opera+, Chrome+

NN4+/Moz/WebKit- and Presto-based browers let scripts adjust the height and width of any window, including the main browser window, by setting properties. This adjustment can be helpful if your page shows itself best with the browser window sized to a particular height and width. Rather than relying on the user to size the browser window for optimum viewing of your page, you can dictate the size of the window (although the user can always resize the main window manually). And because you can examine the operating system of the visitor via the `navigator` object (see Chapter 42), you can size a window to adjust for the differences in font and form element rendering on different platforms.

Supporting browsers provide two different points of reference for measuring the height and width of a window: inner and outer. Both are measured in pixels. The inner measurements are that of the active document area of a window (sometimes known as a window’s content region). If the optimum display of your document depends on the document display area being a certain number of pixels high and/or wide, the `innerHeight` and `innerWidth` properties are the ones to set.

By contrast, the outer measurements reference the outside boundary of the entire window, including whatever chrome is showing in the window: scroll bars, status bar, and so on. Setting the `outerHeight` and `outerWidth` is generally done in concert with a reading of `screen` object properties (see Chapter 42). Perhaps the most common use of the outer properties is to set the browser window to fill the available screen area of the visitor’s monitor.

A more efficient way of modifying both outer dimensions of a window is with the `window.resizeTo()` method, which is also available in IE4+. The method takes pixel width and height (as integer values) as parameters, thus accomplishing a window resizing in one statement. Be aware that resizing a window does not adjust the location of a window. Therefore, just because you set the outer dimensions of a window to the available space returned by the `screen` object, you can’t count on the window suddenly filling the available space on the monitor. Application of the `window.moveTo()` method is necessary to ensure that the top-left corner of the window is at screen coordinates `0,0`.

Despite the freedom that these properties afford the page author, Netscape and Mozilla-based browsers have built in a minimum size limitation for scripts that are not cryptographically signed. You cannot set these properties such that the outer height and width of the window is smaller than 100 pixels on a side. This limitation is to prevent an unsigned script from setting up a small or nearly invisible window that monitors activity in other windows. With signed scripts, however, windows can be made smaller than 100×100 pixels, with the user’s permission. IE4+ maintains a smaller minimum size to prevent resizing a window to zero size.

Caution

Users may dislike your scripts messing with their browser window sizes and positions. NN7+/Moz/WebKit- and Presto-based browsers do not allow scripts to resize windows unless the script is signed. ■

Example

In Listing 27-9, several buttons let you see the results of setting the `innerHeight`, `innerWidth`, `outerHeight`, and `outerWidth` properties. Safari ignores scripted adjustments to these properties, whereas Mozilla users can set preferences that prevent scripts from moving and resizing windows.

LISTING 27-9

Setting Window Height and Width

HTML: jsb27-09.html

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Window Sizer</title>
      <script type="text/javascript" src="jsb27-09.js"></script>
   </head>
   <body>
      <h1>Window Sizer</h1>
      <hr />
      <form>
         <b>Setting Inner Sizes</b><br />
         <input type="button" value="600 Pixels Square"
                onclick="setInner(600,600)" /><br />
         <input type="button" value="300 Pixels Square"
                onclick="setInner(300,300)" /><br />
         <input type="button" value="Available Screen Space"
                onclick="setInner(screen.availWidth, screen.availHeight)" /><br />
         <hr />
         <b>Setting Outer Sizes</b><br />
         <input type="button" value="600 Pixels Square"
                onclick="setOuter(600,600)" /><br />
         <input type="button" value="300 Pixels Square"
                onclick="setOuter(300,300)" /><br />
         <input type="button" value="Available Screen Space"
                onclick="setOuter(screen.availWidth, screen.availHeight)" /><br />
         <hr />
         <input type="button" value="Cinch up for Win95"
                onclick="setInner(273,304)" /><br />
         <input type="button" value="Cinch up for Mac"
                onclick="setInner(273,304)" /><br />
         <input type="button" value="Restore Original"
                onclick="restore()" /><br />
      </form>
   </body>
</html>
```

continued

LISTING 27-9 *(continued)*

```
JavaScript: jsb27-09.js

// store original outer dimensions as page loads
var originalWidth = window.outerWidth;
var originalHeight = window.outerHeight;
alert("original dimensions: \nwindow.outerWidth="
      + window.outerWidth
      + ". \nwindow.outerHeight ="
      + window.outerHeight
      + ". \nwindow.innerWidth = "
      + window.innerWidth
      + ". \nwindow.outerHeight = "
      + window.outerHeight);

// generic function to set inner dimensions
function setInner(width, height)
{
   window.innerWidth = width;
   window.innerHeight = height;
}

// generic function to set outer dimensions
function setOuter(width, height)
{
   window.outerWidth = width;
   window.outerHeight = height;
}

// restore window to original dimensions
function restore()
{
   window.outerWidth = originalWidth;
   window.outerHeight = originalHeight;
}
```

Related Items: `window.resizeTo()`, `window.moveTo()` methods; `screen` object; `navigator` object

`location`

Value: Object Read/Write

Compatibility: WinIE3+, MacIE3+, NN2+, Moz+, Safari+, Opera+, Chrome+

(See the discussion of the `location` object in Chapter 28, "Location and History Objects.")

`locationbar`

(See `directories`)

name

Value: String Read/Write

Compatibility: WinIE3+, MacIE3+, NN2+, Moz+, Safari+, Opera+, Chrome+

All `window` objects can have names assigned to them. Names are particularly useful for working with frames, because a good naming scheme for a multiframe environment can help you determine precisely which frame you're working with in references coming from other frames.

The main browser window, however, has no name attached to it by default. Its value is an empty string. There aren't many reasons to assign a name to the window, because JavaScript and HTML provide plenty of other ways to refer to the `window` object (the `top` property, the `_top` constant for `target` attributes, and the `opener` property from subwindows).

If you want to attach a name to the main window, you can do so by setting the `window.name` property at any time. But be aware that because this is one window property whose life extends beyond the loading and unloading of any given document, chances are that your scripts would use the reference in only one document or frameset. Unless you restore the default empty string, your programmed window name will be present for any other document that loads later. Our suggestion in this regard is to assign a name in a window's or frameset's `onload` event handler and then reset it to empty in a corresponding `onunload` event handler:

```
<body onload="self.name = 'Main'" onunload="self.name = ''">
```

You can see an example of this application in Listing 27-15, where setting a parent window name is helpful for learning the relationships between parent and child windows.

Related Items: `top` property; `window.open()`, `window.sizeToContent()` methods

navigator

Value: Object Read-Only

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

Although the `navigator` object appears as a property of the `window` object only in modern browsers, this object has been around since the very beginning (see Chapter 42). In previous browsers, the `navigator` object was referenced as a stand-alone object. And because you can omit any reference to the `window` object for a `window` object's properties, you can use the same windowless reference syntax for compatibility across all scriptable browsers (at least for the `navigator` object properties that exist across all browsers). That's the way we recommend referring to the `navigator` object.

Example

This book is littered with examples of using the `navigator` object, primarily for performing browser detection. You can find examples of specific `navigator` object properties in Chapter 42.

Related Item: `navigator` object

netscape

Value: Object Read-Only

Compatibility: WinIE-, MacIE-, NN3+, Moz+, Safari-, Opera-, Chrome-

Given its name, you might think that the `netscape` property somehow works in tandem with the `navigator` property, but this is not the case. The `netscape` property is unique to NN/Moz browsers and provides access to functionality that is specific to the Netscape family of browsers, such as the privilege manager.

Example

The `netscape` property is commonly used as a means of accessing the NN/Moz-specific `PrivilegeManager` object to enable or disable security privileges. Following is an example of how this access is carried out:

```
netscape.security.PrivilegeManager.enablePrivilege("UniversalBrowserRead");
```

offscreenBuffering

Value: Boolean or string Read/Write

Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari 1.2+, Opera-, Chrome+

IE4+/Safari 1.2+ by default initially render a page in a buffer (a chunk of memory) before it is blasted to the video screen. You can control this behavior explicitly by modifying the `window.offscreenBuffering` property.

The default value of the property is the string `auto.` You can also assign Boolean `true` or `false` to the property to override the normal automatic handling of this behavior.

Example

If you want to turn off buffering for an entire page, include the following statement at the beginning of your script statements:

```
window.offscreenBuffering = false;
```

onerror

Value: Function Read/Write

Compatibility: WinIE4+, MacIE4+, NN3+, Moz+, Safari-, Opera-, Chrome-

The `onerror` property is an exception to this book's rule of not describing event handlers as properties within object reference sections. The reason is that the `onerror` event brings along some special properties that are useful to control by setting the event handler property in scripts.

Modern browsers (IE5+, NN4+, and W3C) are designed to prevent script errors from being intrusive if a user encounters a script error while loading or interacting with a page. But even the subtle hints about problems (messages or icons in the status bar) can be confusing for users who have no idea what JavaScript is. JavaScript lets you turn off the display of script error windows or messages as someone executes a script on your page. The question is: When should you turn off these messages?

Script errors generally mean that something is wrong with your script. The error may be the result of a coding mistake or, conceivably, a bug in JavaScript (perhaps on a platform version of the browser that you haven't been able to test). If such errors occur, often the script won't continue to do what you intended. Hiding the script error from yourself during development would be foolhardy, because you'd never know whether unseen errors are lurking in your code. It can be equally dangerous to turn off error dialog boxes for users who may believe that the page is operating normally, when in fact it's not. Some data values may not be calculated or displayed correctly.

That said, we can see some limited instances of when you may want to keep such dialog-box windows from appearing. For example, if you know for a fact that a platform-specific bug trips the error message without harming the execution of the script, you may want to prevent that error alert dialog box from appearing in the files posted to your web site. You should do this only after extensive testing to ensure that the script ultimately behaves correctly, even with the bug or error.

Note

IE fires the `onerror` event handler only for runtime errors. This means that if you have a syntactical error in your script that trips the browser as the page loads, the `onerror` event doesn't fire, and you cannot trap that error message. Moreover, if the user has the IE script debugger installed, any code you use to prevent browser error messages from appearing will not work. ■

When the browser starts, the `window.onerror` property is `<undefined>`. In this state, all errors are reported via the normal JavaScript error window or message. To turn off error alerts, set the `window.onerror` property to invoke a function that does absolutely nothing:

```
function doNothing() { return true; }
window.onerror = doNothing;
```

To restore the error messages, reload the page.

You can, however, also assign a custom function to the `window.onerror` property. This function then handles errors in a more friendly way under your script control. Whenever error messages are turned on (the default behavior), a script error (or Java applet or class exception) invokes the function assigned to the `onerror` property, passing three parameters:

- Error message
- URL of the document causing the error
- Line number of the error

You can essentially trap for all errors and handle them with your own interface (or no user notification at all). The last statement of this function must be `return true` if you do not want the JavaScript script error message to appear.

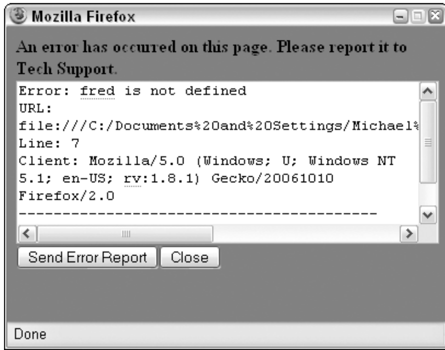
If you are using the NPAPI to communicate with a Java applet directly from your scripts, you can use the same scheme to handle any exception that Java may throw. A Java exception is not necessarily a mistake kind of error: Some methods assume that the Java code will trap for exceptions to handle special cases — for example, reacting to a user's denial of access when prompted by a signed script dialog box. See Chapter 47, "Scripting Java Applets and Plug-Ins" (on the CD-ROM), for an example of trapping for a specific Java exception. Also, see Chapter 21, "Control Stuctures and Exception Handling," for JavaScript exception handling introduced for W3C DOM–compatible browsers.

Example

In Listing 27-10, one button triggers a script that contains an error. We added an error-handling function to process the error so that it opens a separate window and fills in a `textarea` form element (see Figure 27-4). A Submit button is also provided to mail the bug information to a support center email address — an example of how to handle the occurrence of a bug in your scripts.

FIGURE 27-4

An example of a self-reporting error window.



LISTING 27-10

Controlling Script Errors

HTML: jsb27-10.html

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Error Dialog Control</title>
      <script type="text/javascript" src="jsb27-10.js"></script>
   </head>
   <body>
      <h1>Error Dialog Control</h1>
      <hr />
      <form name="myform">
         <input type="button" value="Cause an Error" onclick="goWrong()" />
         <p><input type="button" value="Turn Off Error Dialogs"
                   onclick="errOff()" />
            <input type="button" value="Turn On Error Dialogs"
                   onclick="errOn()" />
         </p>
      </form>
   </body>
</html>
```

JavaScript: jsb27-10.js

```
// function with invalid variable value
function goWrong()
{
   var x = fred;
```

```
}
// turn off error dialogs
function errOff()
{
   window.onerror = doNothing;
}
// turn on error dialogs with hard reload
function errOn()
{
   window.onerror = handleError;
}

// assign default error handler
window.onerror = handleError;

// error handler when errors are turned off...prevents error dialog
function doNothing() { return true; }

function handleError(msg, URL, lineNum)
{
   var errWind = window.open("","errors","height=270,width=400");
   var wintxt = "<html><body bgcolor=red>";
   wintxt += "<b>An error has occurred on this page. "
          + "Please report it to Tech Support.</b>";
   wintxt += "<form method=POST enctype='text/plain' "
          + "action=mailTo:support4@dannyg.com >";
   wintxt += "<textarea name='errMsg' cols=45 rows=8 wrap=VIRTUAL>";
   wintxt += "Error: "
          + msg + "\n";
   wintxt += "URL: "
          + URL + "\n";
   wintxt += "Line: "
          + lineNum + "\n";
   wintxt += "Client: "
          + navigator.userAgent + "\n";
   wintxt += "-----------------------------------------\n";
   wintxt += "Please describe what you were doing when the error occurred:";
   wintxt += "</textarea><br />";
   wintxt += "<input type=SUBMIT value='Send Error Report'>";
   wintxt += "<input type=button value='Close' onclick='self.close()'>";
   wintxt += "</form></body></html>";
   errWind.document.write(wintxt);
   errWind.document.close();
   return true;
}
```

We provide a button that performs a hard reload, which in turn resets the `window.onerror` property to its default value. With error dialog boxes turned off, the error handling function does not run.

Related Items: `location.reload()` method; JavaScript exception handling (Chapter 21, “Control Structures and Exception Handling”); debugging scripts (Chapter 48, “Debugging Scripts,” on the CD-ROM)

opener

Value: Window object reference Read/Write

Compatibility: WinIE3+, MacIE3+, NN3+, Moz+, Safari+, Opera+, Chrome+

Many scripters make the mistake of thinking that a new browser window created with the `window.open()` method has a child-parent relationship similar to the one that frames have with their parents. That's not the case at all. New browser windows, when created, have a very slim link to the window from whence they came: via the `opener` property. The purpose of the `opener` property is to provide scripts in the new window with a valid reference back to the original window. For example, the original window may contain some variable values or general-purpose functions that a new window at this web site wants to use. The original window may also have form elements whose settings are of value to the new window, or that get set by user interaction in the new window.

Because the value of the `opener` property is a reference to a genuine `window` object, you can begin references with the property name. Or, you may use the more complete `window.opener` or `self.opener` reference. But then the reference must include some object or property of that original window, such as a window method or a reference to something contained by that window's document.

If a subwindow opens yet another subwindow, the chain is still valid, albeit one step longer. The third window can reach the main window with a reference that begins

```
opener.opener....
```

It's a good idea for the third window to store in a global variable the value of `opener.opener` while the page loads. Then, if the user closes the second window, the variable can be used to start a reference to the main window.

When a script that generates a new window is within a frame, the `opener` property of the subwindow points to that frame. Therefore, if the subwindow needs to communicate with the main window's parent or another frame in the main window, you have to very carefully build a reference to that distant object. For example, if the subwindow needs to get the `checked` property of a checkbox in a sister frame of the one that created the subwindow, the reference is

```
opener.parent.sisterFrameName.document.formName.checkboxName.checked
```

It is a long way to go, indeed, but building such a reference is always a case of mapping out the path from where the script is, to where the destination is, step by step.

Example

To demonstrate the importance of the `opener` property, take a look at how a new window can define itself from settings in the main window (see Listing 27-11). The `doNew()` function generates a small subwindow and loads the file in Listing 27-12 into the window. Notice the initial conditional statements in `doNew()` that make sure that if the new window already exists, it comes to the front, by invoking the new window's `focus()` method.

LISTING 27-11

Contents of a Main Window Document That Generates a Second Window

HTML: jsb27-11.html

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Master of all Windows</title>
      <script type="text/javascript" src="jsb27-11.js"></script>
   </head>
   <body>
      <form name="input">
         Select a color for a new window:
         <input type="radio" name="color" value="red" checked="checked" />Red
         <input type="radio" name="color" value="yellow" />Yellow
         <input type="radio" name="color" value="blue" />Blue
         <input type="button" name="storage" value="Make a Window" onclick="doNew()"/>
         <hr />
         This field will be filled from an entry in another window:
         <input type="text" name="entry" size="25" />
      </form>
   </body>
</html>
```

JavaScript: jsb27-11.js

```
var myWind;
function doNew()
{
   if (!myWind || myWind.closed)
   {
      myWind = window.open("jsb27-12.html","subWindow","height=200, ↵
               width=350,resizable");
   }
   else
   {
      // bring existing subwindow to the front
      myWind.focus();
   }
}
```

LISTING 27-12

References to the opener Property

HTML: jsb27-12.html

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>New Window on the Block</title>
      <script type="text/javascript" src="jsb27-12.js"></script>
      <script type="text/javascript">
         document.write("<body bgcolor='" + getColor() + "'>")
      </script>
   </head>
   <body>
      <form>
         <input type="button" value="Who's in the Main window?"
                onclick="alert(self.opener.document.title)" />
         <p>Type text here for the main window:
            <input type="text" size="25"
                   onchange="self.opener.document.forms[0].entry.value ↩
                   = this.value" />
        </p>
      </form>
   </body>
</html>
```

JavaScript: jsb27-12.js

```
function getColor()
{
   // shorten the reference
   colorButtons = self.opener.document.forms[0].color;
   // see which radio button is checked
   for (var i = 0; i < colorButtons.length; i++)
   {
      if (colorButtons[i].checked)
      {
         return colorButtons[i].value;
      }
   }
   return "white";
}
```

In the `getColor()` function, the multiple references to the radio-button array can be very long. To simplify the references, the `getColor()` function starts by assigning the radio-button array to a variable we arbitrarily call `colorButtons.` That shorthand now stands in for lengthy references as we loop through the radio buttons to determine which button is checked, and retrieve its value property.

A button in the second window simply fetches the title of the opener window's document. Even if another document loads in the main window in the meantime, the `opener` reference still points to the main window: Its `document` object, however, will change.

Finally, the second window contains a text input object. Enter any text there that you like and then either tab or click out of the field. The `onchange` event handler updates the field in the opener's document (provided that the document is still loaded).

Related Items: `window.open()`, `window.focus()` methods

outerHeight
outerWidth

(See `innerHeight` and `innerWidth`, earlier in this chapter)

pageXOffset
pageYOffset

Value: Integer Read-Only

Compatibility: WinIE-, MacIE-, NN4+, Moz+, Safari+, Opera+, Chrome+

The top-left corner of the content (inner) region of the browser window is an important geographical point for scrolling documents. When a document is scrolled all the way to the top and flush left in the window (or when a document is small enough to fill the browser window without displaying scroll bars), the document's location is said to be `0,0`, meaning zero pixels from the top and zero pixels from the left. If you were to scroll the document, some other coordinate point of the document would be under that top-left corner. That measure is called the *page offset,* and the `pageXOffset` and `pageYOffset` properties let you read the pixel value of the document at the inner window's top-left corner: `pageXOffset` is the horizontal offset, and `pageYOffset` is the vertical offset.

The value of these measures becomes clear if you design navigation buttons in your pages to carefully control paging of content being displayed in the window. For example, you might have a two-frame page in which one of the frames features navigation controls and the other displays the primary content. The navigation controls take the place of scroll bars, which, for aesthetic reasons, are turned off in the display frame. Scripts connected to the simulated scrolling buttons can determine the `pageYOffset` value of the document, and then use the `window.scrollTo()` method to position the document precisely to the next logical division in the document for viewing.

IE has corresponding values as `body` object properties: `body.scrollLeft` and `body.scrollTop` (see Chapter 29).

Related Items: `window.innerHeight`, `window.innerWidth`, `body.scrollLeft`, `body.scrollTop` properties; `window.scrollBy()`, `window.scrollTo()` methods

parent

Value: Window object reference Read-Only

Compatibility: WinIE3+, MacIE3+, NN2+, Moz+, Safari+, Opera+,Chrome+

The `parent` property (and the `top` property, discussed later in this section) comes into play primarily when a document is to be displayed as part of a multiframe window. The HTML documents that users see in the frames of a multiframe browser window are distinct from the document that specifies the frameset for the entire window. That document, though still in the browser's memory (and appearing as the URL in the location field of the browser), is not otherwise visible to the user (except in source view).

If scripts in your visible documents need to reference objects or properties of the frameset window, you can reference those frameset window items with the `parent` property. (Do not, however, expand the reference by preceding it with the `window` object, as in `window.parent.propertyName`, because this causes problems in early browsers.) In a way, the `parent` property seems to violate the object hierarchy because, from a single frame's document, the property points to a level seemingly higher in precedence. If you didn't specify the `parent` property, or instead specified the `self` property from one of these framed documents, the object reference is to the frame only, rather than to the outermost framesetting `window` object.

A nontraditional but perfectly legal way to use the `parent` object is as a means of storing temporary variables. Thus, you could set up a holding area for individual variable values, or even an array of data. Then these values can be shared among all documents loaded into the frames, even when documents change inside the frames. You have to be careful, however, when storing data in the parent on the fly (that is, in response to user action in the frames). In early browsers, variables can revert to their default values (the values set by the parent's own script) if the user resizes the window.

A child window can also call a function defined in the parent window. The reference for such a function is

```
parent.functionName([parameters])
```

At first glance, it may seem as though the `parent` and `top` properties point to the same framesetting `window` object. In an environment consisting of one frameset window and its immediate children, that's true. But if one of the child windows was itself another framesetting window, you wind up with three generations of windows. From the point of view of the youngest child (for example, a window defined by the second frameset), the `parent` property points to its immediate parent, whereas the `top` property points to the first framesetting window in this chain.

On the other hand, a new window created via the `window.open()` method has no parent–child relationship to the original window. The new window's `top` and `parent` point to that new window. You can read more about these relationships in the "Frames" section earlier in this chapter.

Example

To demonstrate how various `window` object properties refer to window levels in a multiframe environment, use your browser to load the Listing 27-13 document. It in turn sets each of two equal-size frames to the same document: Listing 27-14. This document extracts the values of several window properties, plus the `document.title` properties of two different window references.

LISTING 27-13

Framesetting Document for Listing 27-14

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>The Parent Property Example</title>
      <script type="text/javascript">
         self.name = "Framesetter";
      </script>
   </head>
   <frameset cols="50%,50%" onunload="self.name = ''">
      <frame name="JustAKid1" src="jsb27-14.html" />
      <frame name="JustAKid2" src="jsb27-14.html" />
   </frameset>
</html>
```

LISTING 27-14

Revealing Various Window-Related Properties

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Window Revealer II</title>
      <script type="text/javascript">
         function gatherWindowData()
         {
            var msg = "";
            msg += "top name: "
                + top.name + "<br />";
            msg += "parent name: "
                + parent.name + "<br />";
            msg += "parent.document.title: "
                + parent.document.title + "<br />";
            msg += "window name: "
                + window.name + "<br />";
            msg += "self name: "
                + self.name + "<br />";
            msg += "self.document.title: "
                + self.document.title;
            return msg;
         }
      </script>
   </head>
```

continued

LISTING 27-14 *(continued)*

```
   <body>
      <script type="text/javascript">
         document.write(gatherWindowData());
      </script>
   </body>
</html>
```

In the two frames, the references to the `window` and `self` object names return the name assigned to the frame by the frameset definition (`JustAKid1` for the left frame, `JustAKid2` for the right frame). In other words, from each frame's point of view, the `window` object is its own frame. References to `self.document.title` refer only to the document loaded into that window frame. But references to the top and parent windows (which are one and the same in this example) show that those object properties are shared between both frames.

A couple of other fine points are worth highlighting. First, the name of the framesetting window is set as Listing 27-13 loads, rather than in response to an `onload` event handler in the `<frameset>` tag. The reason for this is that the name must be set in time for the documents loading in the frames to get that value. If we had waited until the frameset's `onload` event handler, the name wouldn't be set until after the frame documents had loaded. Second, we restore the parent window's name to an empty string when the framesetting document unloads. This is to prevent future pages from getting confused about the window name.

Related Items: `window.frames`, `window.self`, `window.top` properties

personalbar

(See `directories`)

returnValue

Value: Any data type — Read/Write

Compatibility: WinIE4+, MacIE5+, NN-, Moz+, Safari+, Opera-, Chrome-

Scripts use the `returnValue` property in a document that loads into the originally IE-specific modal dialog box. A modal dialog box is generated via the `showModalDialog()` method, which returns whatever data has been assigned to the `returnValue` property of the dialog-box window before it closes. This is possible because script processing in the main window freezes while the modal dialog box is visible. As the dialog box closes, a value can be returned to the main window's script right where the modal dialog box was invoked, and the main window's script resumes executing statements.

Example

See Listing 27-36 for the `showModalDialog()` method, for an example of how to get data back from a dialog box.

Related Item: `showModalDialog()` method

screen

Value: screen object Read-Only

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

Although the screen object appears as a property of the window object in modern browsers, the screen object is also available in NN4 (see Chapter 42), but as a stand-alone object. Because you can omit any reference to the window object for a window object's properties, the same windowless reference syntax can be used for compatibility with legacy browsers that support the screen object.

Example

See Chapter 42 for examples of using the screen object to determine the video-monitor characteristics of the computer running the browser.

Related Item: screen object

screenLeft
screenTop

Value: Integer Read-Only

Compatibility: WinIE5+, MacIE-, NN-, Moz-, Safari 1.2+, Opera+, Chrome+

WinIE5+ originally provided the screenLeft and screenTop properties of the window object to let you read the pixel position (relative to the top-left 0,0 coordinate of the video monitor) of what Microsoft calls the *client area* of the browser window. The client area excludes most window chrome, such as the title bar, address bar, and the window-sizing bar. Therefore, when the browser window is maximized (meaning that no sizing bars are exposed), the screenLeft property of the window is 0, whereas the screenTop property varies, depending on the combination of toolbars the user has elected to display. For nonmaximized windows, if the window has been positioned so that the top and/or left parts of the client area are out of view, their property values will be negative integers.

These two properties are read-only. You can position the browser window via the window.moveTo() and window.moveBy() methods, but these methods position the top-left corner of the entire browser window, not the client area. IE browsers through version 7 do not provide properties for the position of the entire browser window.

Example

Use The Evaluator (Chapter 4, "JavaScript Essentials") to experiment with the screenLeft and screenTop properties. Start with the browser window maximized (if you are using Windows). Enter the following property name in the top text box:

```
window.screenLeft
```

Click the Evaluate button to see the current setting. Unmaximize the window, and drag it around the screen. Each time you finish dragging, click the Evaluate button again to see the current value. Do the same for window.screenTop.

Related Items: window.moveTo(), window.moveBy() methods

screenX
screenY

Value: Integer Read/Write

Compatibility: WinIE-, MacIE-, NN6+, Moz+, Safari 1.2+, Opera+, Chrome+

NN6+/Moz/W3C provide the `screenX` and `screenY` properties to read the position of the outer boundary of the browser window relative to the top-left coordinates (`0,0`) of the video monitor. The browser window includes the 4-pixels-wide window-sizing bars that surround Win32 windows. Therefore, when the WinNN6+ browser window is maximized, the value for both `screenX` and `screenY` is `-4.` NN/Moz/W3C do not provide the equivalent measures of the browser window client area as found in the `screenLeft` and `screenTop` properties of IE5+. You can, however, find out whether various toolbars are visible in the browser window (see `window.directories`).

Although you can assign a value to either property, current versions of supporting browsers do not adjust the window position in response, if the user has set the preference that prevents window movement and resizing. Moving and resizing windows by script is considered by many web surfers to be unacceptable behavior.

Example

Use The Evaluator (Chapter 4, "JavaScript Essentials") to experiment with the `screenX` and `screenY` properties. Start with the browser window maximized (if you are using Windows). Enter the following property name in the top text box:

```
window.screenY
```

Click the Evaluate button to see the current setting. Unmaximize the window, and drag it around the screen. Each time you finish dragging, click the Evaluate button again to see the current value. Do the same for `window.screenY.`

Related Items: `window.moveTo()`, `window.moveBy()` methods

scrollbars

(See `directories`)

scrollMaxX
scrollMaxY

Value: Integer Read/Write

Compatibility: WinIE-, MacIE-, NN7.1+, Moz1.4+, Safari-, Opera-, Chrome-

The NN7.1+/Moz1.4+ `scrollMaxX` and `scrollMaxY` properties let you determine the maximum horizontal and vertical scrolling extents of a window. Scrolling is possible only if the window displays scroll bars along the desired axis. Values are pixel integers.

Related Items: `scrollX`, `scrollY` properties

scrollX
scrollY

Value: Integer Read-Only

Compatibility: WinIE-, MacIE-, NN6+, Moz+, Safari+, Opera+, Chrome+

The NN6+/Mozilla/Safari `scrollX` and `scrollY` properties let you determine the horizontal and vertical scrolling of a window. Scrolling is possible only if the window displays scroll bars along the desired axis. Values are pixel integers.

Although the IE DOM does not provide similar properties for the window, the same information can be derived from the `body.scrollLeft` and `body.scrollTop` properties.

Example

Use The Evaluator (Chapter 4, "JavaScript Essentials") to experiment with the `scrollX` and `scrollY` properties. Enter the following property in the top text box:

```
window.scrollY
```

Now manually scroll the page down so that you can still see the Evaluate button. Click the button to see how far the window has scrolled along the *y*-axis.

Related Items: `body.scrollLeft`, `body.scrollTop` properties

self

Value: Window object reference Read-Only

Compatibility: WinIE3+, MacIE3+, NN2+, Moz+, Safari+, Opera+, Chrome+

Just as the `window` object reference is optional, so is the `self` property when the object reference points to the same window as the one containing the reference. In what may seem to be an unusual construction, the `self` property represents the same object as the `window.` For instance, to obtain the title of the document in a single-frame window, you can use any of the following three constructions:

```
window.document.title
self.document.title
document.title
```

Although `self` is a property of a window, you should not combine the references within a single-frame window script (for example, don't begin a reference with `window.self`, which has been known to cause numerous scripting problems). Specifying the `self` property, though optional for single-frame windows, can help make an object reference crystal clear to someone reading your code (and to you, for that matter). Multiple-frame windows are where you need to pay particular attention to this property.

JavaScript is pretty smart about references to a statement's own window. Therefore, you can generally omit the `self` part of a reference to a same-window document element. But when you intend to display a document in a multiframe window, complete references (including the `self` prefix) to an object make it much easier on anyone who reads or debugs your code to track who is doing what to whom. You are free to retrieve the `self` property of any window. The value that comes back is a `window` object reference.

Example

Listing 27-15 uses the same operations as Listing 27-5 but substitutes the `self` property for all `window` object references. The application of this reference is entirely optional, but it can be helpful for reading and debugging scripts if the HTML document is to appear in one frame of a multiframe window — especially if other JavaScript code in this document refers to documents

in other frames. The `self` reference helps anyone reading the code know precisely which frame was being addressed. When testing, remember that even though modern browsers support the `status` property, some of them do not display scripted status-bar text associated with links, in order to prevent link spoofing.

LISTING 27-15

Using the self Property

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>self Property</title>
      <script type="text/javascript">
         self.defaultStatus = "Welcome to my Website.";
      </script>
   </head>
   <body>
      <h1>self Property</h1>
      <hr />
      <p><a href="http://www.microsoft.com"
            onmouseover="self.status = 'Visit Microsoft\'s Home page.';return true;"
            onmouseout="self.status = '';return true;">Microsoft</a>
      </p>
      <p><a href="http://mozilla.org"
            onmouseover="self.status = 'Visit Mozilla\'s Home page.';return true;"
            onmouseout="self.status = self.defaultStatus;
            return true;">Mozilla</a></p>
   </body>
</html>
```

Related Items: `window.frames`, `window.parent`, `window.top` properties

sidebar

(See `appCore`)

status

Value: String Read/Write

Compatibility: WinIE3+, MacIE3+, NN2+, Moz+, Safari+, Opera+, Chrome-

At the bottom of the browser window is a status bar. Part of that bar includes an area that normally discloses the document loading progress or the URL of a link that the mouse is pointing to at any given instant. You can control the temporary content of that field by assigning a text string to the `window` object's `status` property. You should adjust the `status` property only in response to events that have a temporary effect, such as a link or image map area object's `onmouseover` event handler. When the `status` property is set in this situation, it overrides any other setting in the status bar. If the user then moves the mouse pointer away from the object that changes the status bar, the bar returns to its default setting (which may be empty on some pages). To prevent link spoofing, however, not all modern browsers display scripted status-bar text associated with links.

Use this window property as a friendlier alternative to displaying the URL of a link as a user rolls the mouse around the page. For example, if you'd rather use the status bar to explain the nature of the destination of a link, put that text into the status bar in response to the `onmouseover` event handler. But be aware that experienced web surfers like to see URLs down there. Therefore, consider creating a hybrid message for the status bar that includes a friendly description followed by the URL in parentheses. In multiframe environments, you can set the `window.status` property without having to worry about referencing the individual frame.

Example

In Listing 27-16, the `status` property is set in a handler embedded in the `onmouseover` attribute of two HTML link tags. Notice that the handler requires a `return true` statement (or any expression that evaluates to `return true`) as the last statement of the handler. This statement is required; otherwise, the status message will not display in all browsers. When testing, remember that even though modern browsers support the `status` property, some of them do not display scripted status-bar text associated with links, in order to prevent link spoofing.

LISTING 27-16

Links with Custom Status-Bar Messages

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>window.status Property</title>
   </head>
   <body>
      <h1>window.status Property</h1>
      <hr />
      <a href="http://www.dannyg.com"
         onmouseover="window.status = 'Go to my Home page. (www.dannyg.com)'; ↩
         return true">
         Home</a>
      <p>
         <a href = "http://mozilla.org"
            onmouseover="window.status = 'Visit Mozilla Home page. (mozilla.org)'; ↩
            return true">
         Mozilla</a>
      </p>
   </body>
</html>
```

As a safeguard against platform-specific anomalies that affect the behavior of `onmouseover` event handlers and the `window.status` property, you should also include an `onmouseout` event handler for links and client-side image map area objects. Such `onmouseout` event handlers should set the `status` property to an empty string. This setting ensures that the status-bar message returns to the `defaultStatus` setting when the pointer rolls away from these objects. If you want to write a generalizable function that handles all window status changes, you can do so, but word the `onmouseover` attribute carefully so that the event handler evaluates to `return true`. Listing 27-17 shows such

an alternative. Again, when testing, remember that some modern browsers do not display scripted status-bar text associated with links, in order to prevent link spoofing.

LISTING 27-17

Handling Status Message Changes

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Generalizable window.status Property</title>
      <script type="text/javascript">
         function showStatus(msg)
         {
            window.status = msg;
            return true;
         }
      </script>
   </head>
   <body>
      <h1>Generalizable window.status Property</h1>
      <hr />
      <a href="http://www.dannyg.com"
         onmouseover="return showStatus('Go to my Home page.')"
         onmouseout="return showStatus('')">Home</a>
      <p>
         <a href="http://mozilla.org"
            onmouseover="return showStatus('Visit Mozilla Home page.')"
            onmouseout="return showStatus('')">Mozilla</a></p>
   </body>
</html>
```

Notice how the event handlers return the results of the `showStatus()` method to the event handler, allowing the entire handler to evaluate to `return true`.

One final example of setting the status bar (shown in Listing 27-18) also demonstrates how to create a simple scrolling banner in the status bar.

LISTING 27-18

Creating a Scrolling Banner

HTML: jsb27-18.html

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
```

```
      <title>Message Scroller</title>
      <script type="text/javascript" src="jsb27-18.js"></script>
   </head>
   <body onload="scrollMsg()">
      <h1>Message Scroller</h1>
      <hr />
   </body>
</html>
```

JavaScript: jsb27-18.js

```
var msg = "Welcome to my world...";
var delay = 150;
var timerId;
var maxCount = 0;
var currCount = 1;

function scrollMsg()
{
   // set the number of times scrolling message is to run
   if (maxCount == 0)
   {
      maxCount = 3 * msg.length;
   }
   window.status = msg;

   // keep track of how many characters have scrolled
   currCount++;

   // shift first character of msg to end of msg
   msg = msg.substring (1, msg.length) + msg.substring (0, 1);

   // test whether we've reached maximum character count
   if (currCount >= maxCount)
   {
      timerID = 0;   // zero out the timer
      window.status = "";   // clear the status bar
      return;   // break out of function
   }
   else
   {
      // recursive call to this function
      timerId = setTimeout("scrollMsg()", delay);
   }
}
```

Because the status bar is being set by a stand-alone function (rather than by an `onmouseover` event handler), you do not have to append a `return true` statement to set the `status` property. The `scrollMsg()` function uses more advanced JavaScript concepts, such as the `window.setTimeout()` method (covered later in this chapter) and string methods (covered in

Chapter 15, "The String Object"). To speed the pace at which the words scroll across the status bar, reduce the value of `delay`.

Many web surfers (including us) don't care for these scrollers that run forever in the status bar. Rolling the mouse over links disturbs the banner display. Use scrolling bars sparingly or design them to run only a few times after the document loads.

Tip

Setting the `status` property with `onmouseover` event handlers has had checkered results along various implementations in Navigator. A script that sets the status bar is always in competition against the browser itself, which uses the status bar to report loading progress. When a hot spot on a page is at the edge of a frame, many times the `onmouseout` event fails to fire, preventing the status bar from clearing itself. Be sure to torture-test any such implementations before declaring your page ready for public access. ■

Related Items: `window.defaultStatus` property; `onmouseover`, `onmouseout` event handlers; `link` object

statusbar
toolbar

(See `locationbar`)

top

Value: Window object reference Read-Only

Compatibility: WinIE3+, MacIE3+, NN2+, Moz+, Safari+, Opera+, Chrome+

The `window` object's `top` property refers to the topmost window in a frameset object hierarchy. For a single-frame window, the reference is to the same object as the window itself (including the `self` and `parent` properties), so do not include `window` as part of the reference. In a multiframe window, the top window is the one that defines the first frameset (in case of nested framesets). Users don't ever really see the top window in a multiframe environment, but the browser stores it as an object in its memory. The reason is that the top window has the road map to the other frames (in case one frame should need to reference an object in a different frame), and its child frames can call upon it. Such a reference looks like this:

```
top.functionName([parameters])
```

For more about the distinction between the `top` and `parent` properties, see the in-depth discussion about scripting frames at the beginning of this chapter. See also the example of the `parent` property for listings that demonstrate the values of the `top` property.

Related Items: `window.frames`, `window.self`, `window.parent` properties

window

Value: Window object Read-Only

Compatibility: WinIE3+, MacIE3+, NN2+, Moz+, Safari+, Opera+, Chrome+

Listing the `window` property as a separate property may be more confusing than helpful. The `window` property is the same object as the `window` object. You do not need to use a reference that begins with `window.window.` Although the `window` object is assumed for many references, you can use `window` as part of a reference to items in the same window or frame as the script statement that makes that reference. You should not, however, use `window` as a part of a reference involving items higher up in the hierarchy (`top` or `parent`).

Methods

alert("*message*")

Returns: Nothing

Compatibility: WinIE3+, MacIE3+, NN2+, Moz+, Safari+, Opera+, Chrome+

An *alert dialog box* is a modal window that presents a message to the user with a single OK button to dismiss the dialog box. As long as the alert dialog box is showing, no other application or window can be made active. The user must dismiss the dialog box before proceeding with any more work in the browser.

The single parameter to the `alert()` method can be a value of any data type, including representations of some unusual data types whose values you don't normally work with in JavaScript (such as complete objects). This makes the alert dialog box a handy tool for debugging JavaScript scripts. Any time you want to monitor the value of an expression, use that expression as the parameter to a temporary `alert()` method in your code. The script proceeds to that point and then stops to show you the value. (See Chapter 48 for more tips on debugging scripts.)

What is often disturbing to application designers is that all JavaScript-created modal dialog boxes (via the `alert()`, `confirm()`, and `prompt()` methods) identify themselves as being generated by JavaScript or the browser. The purpose of this identification is to act as a security precaution against unscrupulous scripters who might try to spoof system or browser alert dialog boxes, inviting a user to reveal passwords or other private information. These identifying words cannot be overwritten or eliminated by your scripts. You can simulate a modal dialog-box window in a cross-browser fashion with regular browser windows, but it is not as robust as a genuine modal window, which you can create in IE4+, and some of the other modern browsers, via the `window.showModalDialog()` method.

Because the `alert()` method is of a global nature (that is, no particular frame in a multiframe environment derives any benefit from laying claim to the alert dialog box), a common practice is to omit all `window` object references from the statement that calls the method. Restrict the use of alert dialog boxes in your HTML documents and site designs. The modality of the windows is disruptive to the flow of a user's navigation around your pages. Communicate with users via forms or by writing to separate document window frames. Of course, alert boxes can still be very handy as a quick debugging aid.

Example

The parameter for the example in Listing 27-19 is a concatenated string. It joins two fixed strings and the value of the browser's `navigator.appName` property. Loading this document causes the alert dialog box to appear, as shown in several configurations in Figure 27-5. The JavaScript Alert line cannot be deleted from the dialog box in earlier browsers, while the title bar can't be changed in later browsers.

LISTING 27-19

Displaying an Alert Dialog Box

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>window.alert() Method</title>
   </head>
   <body>
      <h1>window.alert() Method</h1>
      <hr />
      <script type="text/javascript">
         alert("You are running the " + navigator.appName + " browser.");
      </script>
   </body>
</html>
```

FIGURE 27-5

Results of the alert() method in Listing 27-19 in Firefox and Internet Explorer.

Related Items: `window.confirm()`, `window.prompt()` methods

back()
forward()

Returns: Nothing

Compatibility: WinIE+, MacIE-, NN4+, Moz+, Safari+, Opera+, Chrome+

The purpose of the `window.back()` and `window.forward()` methods that began in NN4 is to offer a scripted version of the global back and forward navigation buttons while allowing the `history` object to control navigation strictly within a particular window or frame — as it should. Even though IE now supports these window methods, they did not catch on (and the `window` object is out of the scope of the W3C DOM Level 2), so you are better off staying with the `history` object's

methods for navigating browser history. For more information about version compatibility and about back and forward navigation, see the `history` object in Chapter 28.

Related Items: `history.back()`, `history.forward()`, `history.go()` methods

clearInterval(*intervalIDnumber*)

Returns: Nothing

Compatibility: WinIE4+, MacIE4+, NN4+, Moz+, Safari+, Opera+, Chrome+

Use the `window.clearInterval()` method to turn off an interval loop action started with the `window.setInterval()` method. The parameter is the ID number returned by the `setInterval()` method. A common application for the JavaScript interval mechanism is animation of an object on a page. If you have multiple intervals running, each has its own ID value in memory. You can turn off any interval by its ID value. As soon as an interval loop stops, your script cannot resume that interval: It must start a new one, which generates a new ID value.

Example

See Listing 27-33 and Listing 27-34 later in this chapter for an example of how `setInterval()` and `clearInterval()` are used together on a page.

Related Items: `window.setInterval()`, `window.setTimeout()`, `window.clearTimeout()` methods

clearTimeout(*timeoutIDnumber*)

Returns: Nothing

Compatibility: WinIE3+, MacIE3+, NN2+, Moz+, Safari+, Opera+, Chrome+

Use the `window.clearTimeout()` method in concert with the `window.setTimeout()` method, as described later in this chapter, when you want your script to cancel a timer that is waiting to run its expression. The parameter for this method is the ID number that the `window.setTimeout()` method returns when the timer starts ticking. The `clearTimeout()` method cancels the specified timeout. A good practice is to check your code for instances where user action may negate the need for a running timer — and to stop that timer before it goes off.

Example

The page in Listing 27-20 features one text box and two buttons. One button starts a countdown timer coded to last 1 minute (easily modifiable for other durations); the other button interrupts the timer at any time while it is running. When the minute is up, an alert dialog box lets you know.

LISTING 27-20

A Countdown Timer

HTML: jsb27-20.html

```
<!DOCTYPE html>
<html>
   <head>
```

continued

LISTING 27-20 *(continued)*

```
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Count Down Timer</title>
      <script type="text/javascript" src="jsb27-20.js"></script>
   </head>
   <body>
      <form>
         <input type="button" name="startTime" value="Start 1 min. Timer"
                onclick="startTimer()" />
         <input type="button" name="clearTime"
                value="Clear Timer" onclick="stopTimer()" />
         <p><input type="text" name="timerDisplay" value="" /></p>
      </form>
   </body>
</html>
```

JavaScript: jsb27-20.js

```
var running = false;
var endTime = null;
var timerID = null;

function startTimer()
{
   running = true;
   now = new Date();
   now = now.getTime();
   // change last multiple for the number of minutes
   endTime = now + (1000 * 60 * 1);
   showCountDown();
}

function showCountDown()
{
   var now = new Date();
   now = now.getTime();
      if (endTime - now <= 0)
      {
         stopTimer();
         alert("Time is up.  Put down your pencils.");
      }
      else
      {
         var delta = new Date(endTime - now);
         var theMin = delta.getMinutes();
         var theSec = delta.getSeconds();
         var theTime = theMin;
         theTime += ((theSec < 10) ? ":0" : ":") + theSec;
         document.forms[0].timerDisplay.value = theTime;
         if (running)
```

```
        {
           timerID = setTimeout("showCountDown()",1000);
        }
     }
}

function stopTimer()
{
   clearTimeout(timerID);
   running = false;
   document.forms[0].timerDisplay.value = "0:00";
}
```

Notice that the script establishes three variables with global scope in the window: `running`, `endTime`, and `timerID`. These values are needed inside multiple functions, so they are initialized outside the functions.

In the `startTimer()` function, you switch the running flag on, meaning that the timer should be going. Using some date functions (see Chapter 17, "The Date Object"), you extract the current time in milliseconds and add the number of milliseconds for the next minute (the extra multiplication by 1 is the place where you can change the amount to the desired number of minutes). With the end time stored in a global variable, the function now calls another function that compares the current and end times, and displays the difference in the text box.

Early in the `showCountDown()` function, check to see whether the timer has wound down. If so, you stop the timer and alert the user. Otherwise, the function continues to calculate the difference between the two times, and formats the time in mm:ss format. As long as the `running` flag is set to `true`, the function sets the 1-second timeout timer before repeating itself. To stop the timer before it has run out (in the `stopTimer()` function), the most important step is to cancel the timeout running inside the browser. The `clearTimeout()` method uses the global `timerID` value to do that. Then the function turns off the running switch and zeros out the display.

When you run the timer, you may occasionally notice that the time skips a second. It's not cheating. It just takes slightly more than 1 second to wait for the timeout and then finish the calculations for the next second's display. What you're seeing is the display catching up with the real time left.

Related Item: `window.setTimeout()` method

close()

Returns: Nothing

Compatibility: WinIE3+, MacIE3+, NN2+, Moz+, Safari+, Opera+, Chrome+

The `window.close()` method closes the browser window referenced by the `window` object. Most likely, you will use this method to close subwindows created from a main document window. If the call to close the window comes from a window other than the new subwindow, the original `window` object must maintain a record of the subwindow object. You accomplish this by storing the value returned from the `window.open()` method in a global variable that will be available to other objects later (for example, a variable not initialized inside a function). If, on the other hand, an object inside the new subwindow calls the `window.close()` method, the `window` or `self` reference is sufficient.

Be sure to include a window as part of the reference to this method. Failure to do so may cause JavaScript to regard the statement as a `document.close()` method, which has a different behavior (see Chapter 29). Only the `window.close()` method can close the window via a script. Closing a window, of course, forces the window to trigger an `onunload` event handler before the window disappears from view, but after you've initiated the `window.close()` method, you cannot stop it from completing its task. Moreover, `onunload` event handlers that attempt to execute time-consuming processes (such as submitting a form in the closing window) may not complete because the window can easily close before the process completes — a behavior that has no workaround (with the exception of the `onbeforeunload` event handler in IE4+).

While we're on the subject of closing windows, a special case exists when a subwindow tries to close the main window (via a statement such as `self.opener.close()`) when the main window has more than one entry in its session history. As a safety precaution against scripts closing windows they did not create, modern browsers ask the user whether he or she wants the main window to close (via a browser-generated dialog box). This security precaution cannot be overridden except in NN4+/Moz via a signed script, when the user grants permission to control the browser (see Chapter 49).

Example

See Listing 27-4 (for the `window.closed` property), which provides a cross-platform example of applying the `window.close()` method across multiple windows.

Related Items: `window.open()`, `document.close()` methods

`confirm("message")`

Returns: Boolean

Compatibility: WinIE3+, MacIE3+, NN2+, Moz+, Safari+, Opera+, Chrome+

A confirm dialog box presents a message in a modal dialog box along with OK and Cancel buttons. Such a dialog box can be used to ask a question of the user, usually prior to a script's performing actions that will not be undoable. Querying a user about proceeding with typical web navigation in response to user interaction on a form element is generally a disruptive waste of the user's time and attention. But for operations that may reveal a user's identity or send form data to a server, a JavaScript confirm dialog box may make a great deal of sense. Users may also accidentally click buttons, so you should provide avenues for backing out of an operation before it executes.

Because this dialog box returns a Boolean value (OK = `true`; Cancel = `false`), you can use this method as a comparison expression or as an assignment expression. In a comparison expression, you nest the method within any other statement where a Boolean value is required. For example:

```
if (confirm("Are you sure?"))
{
   alert("OK");
}
else
{
   alert("Not OK");
}
```

Here, the returned value of the confirm dialog box provides the desired Boolean value type for the `if...else` construction (see Chapter 21).

This method can also appear on the right side of an assignment expression, as in:

```
var adult = confirm("Do you certify that you are over 18 years old?");
if (adult)
{
   //statements for adults
}
else
{
   //statements for children
}
```

You cannot specify other alert icons or labels for the two buttons in JavaScript confirm dialog-box windows.

The example in Listing 27-21 shows the user interface part of how you can use a confirm dialog box to query a user before clearing a table full of user-entered data. The text in the title bar, as shown in Figure 27-6, cannot be removed from the dialog box.

LISTING 27-21

The Confirm Dialog Box

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>window.confirm() Method</title>
      <script type="text/javascript">
         function clearTable()
         {
            if (confirm("Are you sure you want to empty the table?"))
            {
               alert("Emptying the table...");   // for demo purposes
               //statements that actually empty the fields
            }
         }
      </script>
   </head>
   <body>
      <h1>window.confirm() Method</h1>
      <hr />
      <form>
         <h3>A large table with data would be here.</h3>
         <input type="button" name="clear" value="Reset Table"
                onclick="clearTable()" />
      </form>
   </body>
</html>
```

FIGURE 27-6

A JavaScript confirm dialog box in Internet Explorer.

Related Items: `window.alert()`, `window.prompt()`, `form.submit()` methods

`createPopup()`

Returns: Pop-up object reference

Compatibility: WinIE5.5+, MacIE-, NN-, Moz-, Safari-, Opera-, Chrome-

An IE pop-up window is a chromeless rectangular space that overlaps the current window. Unlike the dialog boxes generated by the `showModalDialog()` and `showModelessDialog()` methods, the pop-up window's entire content must be explicitly controlled by script. That also goes for the size and location of the window. Generating the window via the `createPopup()` method simply creates the object in memory without displaying it. You can then use the reference to the pop-up window that is returned by the method to position the window, populate its content, and make it visible. See details in the description of the `popup` object later in this chapter.

Example

See Listing 27-46 later in this chapter for an example of the `createPopup()` method.

Related Item: `popup` object

`dump("`*`message`*`")`

Returns: Nothing

Compatibility: WinIE-, MacIE-, NN7.1+, Moz1.4+, Safari-, Opera-, Chrome-

The `window.dump()` method is a debugging/diagnostic method that you can use to output a string of text to *standard output,* which is typically the operating system's console window. The `dump()` method provides a less intrusive alternative to displaying debugging messages via the `alert()` method.

`execScript("`*`exprList`*`"[, `*`language`*`])`

Returns: Nothing

Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-, Opera-, Chrome-

The IE-specific `window.execScript()` method executes one or more script statements that are passed as string expressions. The first parameter is a string version of one or more script statements (multiple statements must be separated by semicolons). The second, optional, parameter is the language interpreter the browser should use to execute the script statement. Acceptable values for the

language are `JavaScript`, `JScript`, `VBS`, and `VBScript`. The default value is `JScript`, so you can omit the second parameter when supplying expressions in JavaScript.

Unlike the JavaScript core language `eval()` function (which also executes string versions of JavaScript statements), the `execScript()` method returns no values. Even so, the method operates within the global variable space of the window holding the current document. For example, if a document's script declares a global variable as follows

```
var myVar;
```

the `execScript()` method can read or write to that variable:

```
window.execScript("myVar = 10; myVar += 5");
```

After this statement runs, the global variable `myVar` has a value of `15`.

Example

Use The Evaluator (Chapter 4, "JavaScript Essentials") to experiment with the `execScript()` method. The Evaluator has predeclared global variables for the lowercase letters *a* through *z*. Enter each of the following statements in the top text box, and observe the results for each.

```
a
```

When first loaded, the variable is declared, but assigned no value, so it is undefined:

```
window.execScript("a = 5")
```

The method returns no value, so the mechanism inside The Evaluator says that the statement is undefined:

```
a
```

The variable is now 5.

```
window.execScript("b = a * 50")
b
```

The `b` global variable has a value of `250`. Continue exploring with additional script statements. Use semicolons to separate multiple statements within the string parameter.

Related Item: `eval()` function

find(["*searchString*" [, *matchCaseBoolean*, *searchUpBoolean*]])

Returns: Boolean value for non-dialog-box searches

Compatibility: WinIE-, MacIE-, NN4+, Moz1.0.1+, Safari+, Opera-, Chrome+

The `window.find()` method introduced in NN4 mimics the powers of the browser's Find dialog box, accessible from the Find button in the toolbar. This method was deactivated in NN6, but reactivated in NN7/Moz1.0.1.

If you specify no parameters, the browser's Find dialog box appears, just as though the user had clicked the Find button in the toolbar. With no parameters, this function does not return a value.

You can specify a search string as a parameter to the function. The search is based on simple string matching, and is not in any way connected with the regular-expression kind of search (see Chapter 45, "The Regular Expression and RegExp Objects"). If the search finds a match, the browser scrolls to that matching word and highlights the word, just as though it were using the browser's own Find dialog box. The function also returns a Boolean `true` after a match is found. If no match is found in the document, or no more matches occur in the current search direction (the default direction is from top to bottom), the function returns `false`.

Table 27-3 lists the optional parameters you can use; in all cases, the default value is `false`. These choices are identical to the ones that appear in NN4+'s Find dialog box.

TABLE 27-3

window.find() Method Attributes Controllable via Script

Attribute	Description
`caseSensitive`	Boolean; `true` specifies that the search is case sensitive
`backwards`	Boolean; `true` specifies that the search is backwards
`wraparound`	Boolean; `true` specifies that the search will wrap to the top of the document when the bottom is reached.
`wholeWord`	Boolean; `true` specifies that the search is a whole word search.
`searchInFrames`	Boolean; `true` specifies to search in frames.
`showDialog`	Boolean; `true` specifies to display the browser's find dialog.

Some modern browsers such as Firefox have evolved to forego the Find dialog box in favor of an integrated find feature that appears at the bottom of the browser window. This approach to find can be applied to the entire page at the same time, in which case all the text matches are highlighted.

IE4+ also has a scripted text search facility, but it is implemented in an entirely different way (using the `TextRange` object described in Chapter 33, "Body Text Objects"). The visual behavior also differs in that it does not highlight and scroll to a matching string in the text.

Example

A simple call to the `window.find()` method looks as follows:

```
var success = window.find("contract");
```

And if you want the search to be case sensitive, add at least one of the optional parameters:

```
success = window.find(matchString,caseSensitive);
```

In many ways, the `window.find()` method is a remnant of NN4. Refer to discussions of the `TextRange` and `Range` objects in Chapter 33 for more modern implementations of body-text searching.

Related Items: `TextRange`, `Range` objects (Chapter 33)

`forward()`

(See `window.back()`)

`geckoActiveXObject("`*progID*`")`

Returns: WMP control object

Compatibility: WinIE-, MacIE-, NN7.1+, Moz1.4+, Safari-, Opera-, Chrome-

One interesting result of the NN/IE browser wars is Microsoft's victory in establishing Windows Media Player as the PC media player of choice in the Windows operating system. Because WMP is implemented as an ActiveX control, NN/Moz browsers were somewhat left out in the cold in terms of scriptability. The `window.geckoActiveXObject()` method was added to Moz1.4 browsers to give them the capability of accessing WMP as an ActiveX control. Although the name of the method suggests generic support for ActiveX controls, it currently enables you to open only the WMP control.

The only parameter to `geckoActiveXObject()` is a programmatic ID, which for WMP is currently `MediaPlayer.MediaPlayer.1`. So, to grab a WMP control reference for media playback using the `geckoActiveXObject()` method, use code such as this:

```
var player = new GeckoActiveXObject("MediaPlayer.MediaPlayer.1");
```

`getComputedStyle(elementNodeRef, "pseudoElementName")`

Returns: CSS style object

Compatibility: WinIE-, MacIE-, NN-, Moz+, Safari+, Opera+, Chrome+

The `window.getComputedStyle()` method enables you to access the cascading style sheet (CSS) style object associated with a given element. You specify an element node reference as the first parameter to the method, along with the optional name of a specific pseudoelement to which the style applies. We say *optional* because you can pass an empty string as the second parameter to obtain a style object with no pseudoelement implications.

Note

Although various browsers support the `getComputedStyle()` method, the values returned could be in different formats, depending on the style specified for the element. For example, if you requested the style of an element that had a background color, Opera would return the color in hex format, and Firefox, Safari, and Chrome would return it in `rgb()` format. ■

Although the `getComputedStyle()` method is defined (and works) for the `window` object, the W3C DOM prefers `document.defaultView.getComputedStyle()` as the standard means of accessing a style object for an element. It's the same method that's ultimately being called; the way it's accessed is what differs.

`getSelection()`

Returns: Selection object

Compatibility: WinIE-, MacIE-, NN6+, Moz+, Safari+, Opera+, Chrome+

This method takes the place of the deprecated method of the same name that appeared in the `document` object. The method offers a scripted way of capturing the text selected by a user in a page, which is a common task involving the selection and copying of body text in a document for pasting into other application documents. The `window.getSelection()` method returns the string of text

selected by the user. If nothing is selected, an empty string is the result. Returned values consist only of the visible text on the page and not the underlying HTML or style of the text.

The WinIE4+ equivalent involves the `document.selection` property, which returns an IE `selection` object. To derive the text from this object, you must create a `TextRange` object from it and then inspect the `text` property:

```
var selectedText = document.selection.createRange().text;
```

Example

The document in Listing 27-22 provides a cross-browser (but not MacIE5) solution to capturing text that a user selects in the page. Selected text is displayed in the text area. The script uses browser detection and branching to account for the differences in event handling between Mozilla and Internet Explorer.

LISTING 27-22

Retrieving Selected Text

HTML: jsb27-22.html

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Getting Selected Text</title>
      <script type="text/javascript" src="../jsb-global.js"></script>
      <script type="text/javascript" src="jsb27-22.js"></script>
   </head>
   <body>
      <h1>Getting Selected Text</h1>
      <hr />
      <p>Select some text and see how JavaScript can capture the selection:</p>
      <h2>ARTICLE I</h2>
      <p>Congress shall make no law respecting an establishment of religion, or
         prohibiting the free exercise thereof; or abridging the freedom of
         speech, or of the press; or the right of the people peaceably to
         assemble, and to petition the government for a redress of grievances.
      </p>
      <form>
         <textarea name="selectedText" rows="3" cols="40" wrap="virtual">
         </textarea>
      </form>
   </body>
</html>
```

JavaScript: jsb27-22.js

```
addEvent(document, "mouseup", showSelection);
```

```
function showSelection()
{
   if (window.getSelection)
   {
      document.forms[0].selectedText.value = window.getSelection();
   }
   else if (document.selection)
   {
      document.forms[0].selectedText.value = document.selection.createRange().text;
      event.cancelBubble = true;
   }
}
```

Related Item: `document.selection` property

home()

Returns: Nothing

Compatibility: WinIE-, MacIE-, NN4+, Moz+, Safari-, Opera+, Chrome-

Like many of the window methods originally introduced in Navigator 4, the `window.home()` method provides a scripted way of replicating the action of a toolbar button: the Home button. The action navigates the browser to whatever URL is set in the browser preferences for home-page location. You cannot control the default home page of a visitor's browser.

Related Items: `window.back()`, `window.forward()` methods; `window.toolbar` property

moveBy(*deltaX*,*deltaY*)
moveTo(*x*,*y*)

Returns: Nothing

Compatibility: WinIE4+, MacIE4+, NN4+, Moz+, Safari+, Opera+, Chrome-

Version 4 browsers introduced the capability of allowing JavaScript to adjust the location of a browser window onscreen. This applies to the main window or any subwindow generated by script. NN/Moz regard the possibility of a window moved out of screen view as a potential security hole, so signed scripts are needed in NN4+/Moz to move a window offscreen. While these methods are compatible with Chrome, Chrome ignores them.

You can move a window to an absolute position onscreen or adjust it along the horizontal and/or vertical axis by any number of pixels, irrespective of the absolute pixel position. The coordinate space for the *x* (horizontal) and *y* (vertical) position is the entire screen, with the top-left corner representing `0,0`. The point of the window you set with the `moveBy()` and `moveTo()` methods is the top-left corner of the outer edge of the browser window. Therefore, when you move the window to point `0,0`, the window is set flush with the top-left corner of the screen. This may not be the equivalent of a truly maximized window for all browsers and operating systems, however, because a maximized window's coordinates may be negative by a handful of pixels.

The difference between the `moveTo()` and `moveBy()` methods is that one is an absolute move, whereas the other is relative with respect to the current window position. Parameters you specify for `moveTo()` are the precise horizontal and vertical pixel counts onscreen where you want the top-left

corner of the window to appear. By contrast, the parameters for `moveBy()` indicate how far to adjust the window location in either direction. If you want to move the window 25 pixels to the right, you must still include both parameters, but the *y* value will be zero:

```
window.moveBy(25,0);
```

To move to the left, the first parameter must be a negative number.

Example

Several examples of using the `window.moveTo()` and `window.moveBy()` methods are shown in Listing 27-23. The page presents four buttons, each of which performs a different kind of browser-window movement.

LISTING 27-23

Window Boogie

HTML: jsb27-23.html

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Window Gymnastics</title>
      <script type="text/javascript" src="jsb27-23.js"></script>
   </head>
   <body onload="init()">
      <h1>Window Gymnastics</h1>
      <hr />
      <form name="buttons">
         <ul>
            <li><input name="offscreen" type="button"
                       value="Disappear a Second"
                       onclick="moveOffScreen()" />
            </li>
            <li><input name="circles" type="button"
                       value="Swift and Tiny Circular Motion"
                       onclick="revolve()" />
            </li>
            <li><input name="bouncer" type="button"
                       value="Zig Zag"
                       onclick="zigzag()" />
            </li>
            <li><input name="expander" type="button"
                       value="Maximize"
                       onclick="maximize()" />
            </li>
         </ul>
      </form>
```

```
   </body>
</html>
```

JavaScript: jsb27-23.js

```
// wait in onload for page to load and settle in IE
function init()
{
   // fill missing IE properties
   if (!window.outerWidth)
   {
      window.outerWidth = document.body.clientWidth;
      window.outerHeight = document.body.clientHeight + 30;
   }
}

// function to run when window captures a click event
function moveOffScreen()
{
   // branch for NN security
   if (window.netscape)
   {
      // will also get here if in Opera, but Opera will throw error, so...
      try
      {
        netscape.security.PrivilegeManager.enablePrivilege("UniversalBrowserWrite");
      }
      catch (e)
      {
         null;
      }
   }
   var maxX = screen.width;
   var maxY = screen.height;
   window.moveTo(maxX+1, maxY+1);
   setTimeout("window.moveTo(0,0)",500);
   if (window.netscape)
   {
      // will also get here if in Opera, but Opera will throw error, so...
      try
      {
       netscape.security.PrivilegeManager.disablePrivilege("UniversalBrowserWrite");
      }
      catch (e)
      {
         null;
      }
   }
}

// moves window in a circular motion
function revolve()
```

continued

LISTING 27-23 *(continued)*

```
{
   var winX = (screen.availWidth - window.outerWidth) / 2;
   var winY = 50;
   window.resizeTo(400,300);
   window.moveTo(winX, winY);
   for (var i = 1; i < 36; i++)
   {
      winX += Math.cos(i * (Math.PI/18)) * 5;
      winY += Math.sin(i * (Math.PI/18)) * 5;
      window.moveTo(winX, winY);
   }
}

// moves window in a horizontal zig-zag pattern
function zigzag()
{
   window.resizeTo(400,300);
   window.moveTo(0,80);
   var incrementX = 2;
   var incrementY = 2;
   var floor = screen.availHeight - window.outerHeight;
   var rightEdge = screen.availWidth - window.outerWidth;
   for (var i = 0; i < rightEdge; i += 2)
   {
      window.moveBy(incrementX, incrementY);
      if (i%60 == 0)
      {
         incrementY = -incrementY;
      }
   }
}

// resizes window to occupy all available screen real estate
function maximize()
{
   window.moveTo(0,0);
   window.resizeTo(screen.availWidth, screen.availHeight);
}
```

To run successfully in NN/Moz, the first button requires that you have codebase principals turned on (see Chapter 49 to take advantage of what would normally be a signed script). The `moveOff-Screen()` function momentarily moves the window entirely out of view. Interestingly, Opera recognizes `window.netscape` but throws an error if an attempt is made to turn on the codebase principals. A `try catch` gets around that problem. Notice how the script determines the size of the screen before deciding where to move the window. After the journey offscreen, the window comes back into view at the top-left corner of the screen.

If using the web sometimes seems like going around in circles, the second function, `revolve()`, should feel just right. After reducing the size of the window and positioning it near the top center of the screen, the script uses a bit of math to position the window along 36 spots around a perfect circle (at 10-degree increments). This is an example of how to control a window's position dynamically based on math calculations. IE complicates the job a bit by not providing properties that reveal the outside dimensions of the browser window.

To demonstrate the `moveBy()` method, the third function, `zigzag()`, uses a `for` loop to increment the coordinate points to make the window travel in a sawtooth pattern across the screen. The *x* coordinate continues to increment linearly until the window is at the edge of the screen (also calculated on the fly to accommodate monitors of any size). The *y* coordinate must increase and decrease as that parameter changes direction as it moves around the screen.

In the fourth function, you see some practical code (finally) that demonstrates how best to simulate maximizing the browser window to fill the entire available screen space on the visitor's monitor.

Related Items: `window.outerHeight`, `window.outerWidth` properties; `window.resizeBy()`, `window.resizeTo()` methods

navigate(*"URL"*)

Returns: Nothing

Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-, Opera+, Chrome-

The `window.navigate()` method is an IE-original method that lets you load a new document into a window or frame. This method's action is the same as assigning a URL to the `location.href` property — a property that is available on all scriptable browsers. If your audience is entirely IE or Opera based, this method is safe. Otherwise, we recommend the `location.href` property as the best navigation approach.

Example

Supply any valid URL as the parameter to the method, as in:

```
window.navigate("http://www.dannyg.com");
```

Related Item: `location` object

open(*"URL"*, *"windowName"* [, *"windowFeatures"*] [,*replaceFlag*])

Returns: A window object representing the newly created window; `null` if method fails

Compatibility: WinIE3+, MacIE3+, NN2+, Moz+, Safari+, Opera+, Chrome+

With the `window.open()` method, a script provides a web-site designer an immense range of options for the way a second or third web browser window looks on the user's computer screen. Moreover, most of this control can work with all JavaScript-enabled browsers without the need for signed scripts. Because the interface elements of a new window are easier to envision, we cover those aspects of the `window.open()` method parameters first.

Setting new window features

The optional `windowFeatures` parameter is one string consisting of a comma-separated list of assignment expressions (behaving something like HTML tag attributes). *Important:* For the best browser compatibility, do not put spaces after the commas. If you omit the third parameter, JavaScript creates the same type of new window you get from the New Web Browser menu choice in the File menu. But you can control which window elements appear in the new window with the third parameter. Remember this important rule: If you specify even one of the method's original set of third parameter values, all other features are turned off unless the parameters specify the features to be switched on. Table 27-4 lists the attributes that you can control for a newly created window in all browsers. Except where noted, all Boolean values default to `yes` if you do not specify the third parameter.

TABLE 27-4

window.open() Method Attributes Controllable via Script

Attribute	Browsers	Description
`alwaysLowered`[3]	NN4+/Moz+	(Boolean) Always behind other browser windows
`alwaysRaised`[3]	NN4+/Moz+	(Boolean) Always in front of other browser windows
`channelMode`	IE4+	(Boolean) Theater mode with channel band (default is `no`)
`chrome`[3]	NN7.2+/Moz1.7+	(Boolean) Browser user interface features
`close`	NN4+/Moz+	(Boolean) System close command icon and menu item at top of dialog window
`dependent`[6]	NN4+/Moz+	(Boolean) Subwindow closes if the opener window closes
`directories`[7]	NN2+/Moz+, IE3–6	(Boolean) Displays personal/bookmarks/links toolbar
`fullscreen`[5]	IE4+	(Boolean) No title bar or menus (default is `no`)
`height`	NN2+/Moz+, IE3+, Safari+, Opera+, Chrome+	(Integer) Content region height in pixels
`innerHeight`[4]	NN4+/Moz+, Safari+, Chrome+	(Integer) Content region height; same as old height property
`innerWidth`[4]	NN4+/Moz+, Safari+, Chrome+	(Integer) Content region width; same as old width property
`left`	NN6+/Moz+, IE4+, Safari+, Opera+, Chrome+	(Integer) Horizontal position of top-left corner onscreen
`location`[8]	NN2+/Moz+, IE3+, Safari	(Boolean) Field displaying the current URL
`menubar`[1]	NN2+/Moz+, IE3+, Safari	(Boolean) Menu bar at top of window

Attribute	Browsers	Description
`minimizable`	NN7.1+/Moz1.2+	(Boolean) Minimize command icon at top of dialog windows only
`modal`[3]	NN7.1+/Moz1.2+	(Boolean) Modality of window, as in preventing access to the main window until the opened window is closed
`outerHeight`[4]	NN4+/Moz+	(Integer) Visible window height
`outerWidth`[4]	NN4+/Moz+	(Integer) Visible window width
`personalBar`[7]	NN4+/Moz+	(Boolean) Mozilla-specific version of the `directories` attribute
`resizable`[2,9]	NN2+/Moz0−1.4, IE3+	(Boolean) Interface elements that allow resizing by dragging
`screenX`[4]	NN4+/Moz+	(Integer) Horizontal position of top-left corner onscreen
`screenY`[4]	NN4+/Moz+	(Integer) Vertical position of top-left corner onscreen
`scrollbars`	NN2+/Moz+, IE3+, Opera+	(Boolean) Displays scroll bars if document is larger than window
`status`	NN2+/Moz+, IE3+, Safari	(Boolean) Status bar at bottom of window
`titlebar`[3]	NN4+/Moz+	(Boolean) Title bar and all other border elements
`toolbar`	NN2+/Moz+, IE3+, Safari	(Boolean) Back, Forward, and other buttons in the row
`top`	NN6+/Moz+, IE4+, Safari+, Opera+, Chrome+	(Integer) Vertical position of top-left corner onscreen
`width`	NN2+/Moz+, IE3+, Safari+, Opera+, Chrome+	(Integer) Content region width in pixels
`z-lock`[3]	NN4+/Moz+	(Boolean) Window layer is fixed below browser windows

[1]Not on Macintosh because the menu bar is not in the browser window; when off in MacNN4, displays an abbreviated Mac menu bar.
[2]Macintosh windows are always resizable.
[3]Requires a signed script.
[4]Requires a signed script to size or position a window beyond safe threshold. `Left` and `Top` have precedence over these two features, respectively.
[5]Behavior changed in IE6+. Its use is not recommended and, except in a kiosk type situation, we strongly concur as its use has always upset users.
[6]Behavior changed in Moz5. It was never implemented on Mac OS. It is currently under revision to be removed.
[7]In Mozilla based browsers, this feature only works if `toolbar` is also set to `yes`.
[8]In Safari, if `toolbar` is checked, then `location` always behaves as checked.
[9]In IE7+ this feature disables tabs in the new window. In FF3+ new windows are always resizable.

***windowObject*.open()**

Boolean values are handled a bit differently than you might expect. The value for `true` can be `yes`, `1`, or just the feature name by itself; for `false`, use a value of `no` or `0`. If you omit any Boolean attributes, they are rendered as `false`. Therefore, if you want to create a new window that shows only the toolbar and status bar and is resizable, the method looks like this:

```
window.open("newURL","NewWindow", "toolbar,status,resizable");
```

A new window whose height and width has not been specified is set to the default size of the browser window that the browser creates from a File menu's New Web Browser command. In other words, a new window does not automatically inherit the size of the window making the `window.open()` method call. A new window created via a script is positioned somewhat arbitrarily unless you use the window positioning attributes available in modern browsers. Notice that the position attributes are different for each browser (`screenX` and `screenY` for NN/Moz/W3C; `left` and `top` for IE). You can include both sets of attributes in a single parameter string because the browser ignores attributes that it doesn't recognize.

Note

The ability to invoke `window.open()` via a window's `onload` and `onunload` event handlers has led to severe abuse in the form of unwanted pop-up advertising windows. Browsers that include pop-up blockers (such as IE6+, and Mozilla/WebKit/Presto-based browsers) prevent the method from being invoked by these event handlers. With more browsers and users employing pop-up blockers every day, you should not even think about blasting pop-up ads to web surfers. ■

Netscape/Mozilla-only signed scripts

Many NN/Moz-specific attributes are deemed to be security risks, and thus require signed scripts and the user's permission before they are recognized. If the user fails to grant permission, the secure parameter is ignored.

To apply signed scripts to opening a new window with the secure window features, you must enable `UniversalBrowserWrite` privileges as you do for other signed scripts (see Chapter 49). A code fragment that generates an `alwaysRaised` style window follows:

```
<script type="text/javaScript" archive="myJar.jar" id="1">
function newRaisedWindow()
{
  netscape.security.PrivilegeManager.enablePrivilege("UniversalBrowserWrite");
  var newWindow = window.open("","","height=100,width=300,alwaysRaised=yes");
  netscape.security.PrivilegeManager.disablePrivilege("UniversalBrowserWrite");
  var newContent = "<html><body><b> "On top of spaghetti!"</b>";
  newContent += "<form><center><input type='button' value='OK'";
  newContent += "onclick='self.close()'></center></form></body></html>";
  newWindow.document.write(newContent);
  newWindow.document.close();
}
</script>
```

You can experiment with the look and behavior of new windows with any combination of attributes with the help of the script in Listing 27-24. This page presents a table of new window Boolean attributes and creates a new 300×300 pixel window based on your choices. This page assumes that if you are using NN/Moz, you have codebase principals turned on for signed scripts (see Chapter 49).

Be careful with turning off the title bar. With the title bar off, the content appears to float in space because absolutely no borders are displayed. However, hotkeys are always turned on, so you can use Ctrl+W to close this borderless window (except on the Mac, for which the hotkeys are always disabled with the title bar off). In browsers that support the `titlebar` feature, you can turn a computer into a kiosk by sizing a window to the screen's dimensions and setting the window options to `"titlebar=no"`. In IE you can also add `fullscreen=yes` to the feature list.

LISTING 27-24

New Window Laboratory

HTML: jsb27-24.html

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>window.open() Options</title>
      <style type="text/css">
          .colHeaders
          {
             background-color:yellow; text-align:center;
          }
      </style>
      <script type="text/javascript" src="jsb27-24.js"></script>
   </head>
   <body>
      <h1>Select new window options:</h1>
      <hr />
      <form>
         <table border="2">
            <tr>
               <td class="colHeaders" colspan="2">
                  Browser Features:
               </td>
            </tr>
            <tr>
               <td>
                  <input type="checkbox" name="toolbar" />
                  toolbar
               </td>
               <td>
                  <input type="checkbox" name="location" />
                  location
               </td>
            </tr>
            <tr>
               <td>
                  <input type="checkbox" name="directories" />
                  directories
               </td>
```

continued

LISTING 27-24 *(continued)*

```
               <td>
                  <input type="checkbox" name="status" />
                  status
               </td>
            </tr>
            <tr>
               <td>
                  <input type="checkbox" name="menubar" />
                  menubar
               </td>
               <td>
                  <input type="checkbox" name="scrollbars" />
                  scrollbars
               </td>
            </tr>
            <tr>
               <td>
                  <input type="checkbox" name="resizable" />
                  resizable
               </td>
               <td>
                  <input type="checkbox" name="titlebar" checked="checked" />
                  titlebar
               </td>
            </tr>
            <tr>
               <td colspan="2" align="middle">
                  <input type="button" name="forAll" value="Make New Window"
                         onclick="makeNewWind(this.form)" />
               </td>
            </tr>
         </table>
      </form>
      <br />
   </body>
</html>
```

JavaScript: jsb27-24.js

```
function makeNewWind(form)
{
   if (window.netscape)
   {
      // will also get here if in Opera, but Opera will throw error, so...
      try
      {
      netscape.security.PrivilegeManager.enablePrivilege("UniversalBrowserWrite");
      }
      catch(e)
```

```
            {
               null;
            }
         }
         var attr = "width=300,height=300";
         for (var i = 0; i < form.elements.length; i++)
         {
            if (form.elements[i].type == "checkbox")
            {
               attr += "," + form.elements[i].name + "=";
               attr += (form.elements[i].checked) ? "yes" : "no";
            }
         }
         var newWind = window.open("bofright.html","subwindow",attr);
         if (window.netscape)
         {
            try
            {
               netscape.security.PrivilegeManager.revertPrivilege("CanvasAccess");
            }
            catch(e)
            {
               null;
            }
         }
      }
```

Specifying a window name

Getting back to the other parameters of `window.open()`, the second parameter is the name for the new window. Don't confuse this parameter with the document's title, which normally would be set by whatever HTML text determines the content of the window. A window name must be the same style of the one-word identifier that you use for other object names and variables. This name is also an entirely different entity from the `window` object that the `open()` method returns. You don't use the name in your scripts. At most, the name can be used for `target` attributes of links and forms.

Loading content into a new window

A script generally populates a window with one of two kinds of information:

- An existing HTML document whose URL is known beforehand
- An HTML page created on the fly

To create a new window that displays an existing HTML document, supply the URL as the first parameter of the `window.open()` method. If your page is having difficulty loading a URL into a new page, try specifying the complete URL of the target document (instead of just the filename).

Leaving the first parameter as an empty string forces the window to open with a blank document, ready to have HTML written to it by your script (or loaded separately by another statement that sets that window's location to a specific URL). If you plan to write the content of the window on the fly, assemble your HTML content as one long string value and then use the `document.write()` method to post that content to the new window. If you plan to append

no further writing to the page, also include a `document.close()` method at the end to tell the browser that you're finished with the layout (so that the `Layout:Complete` or `Done` message appears in the status bar, if your new window has one).

A call to the `window.open()` method returns a reference to the new window's object if the window opens successfully. This value is vitally important if your script needs to address elements of that new window (such as when writing to its document).

To allow other functions in your script to reference the subwindow, you should assign the result of a `window.open()` method to a global variable. Before writing to the new window the first time, test the variable to make sure that it is not a `null` value; the window may have failed to open because of low memory, for instance. If everything is okay, you can use that variable as the beginning of a reference to any property or object within the new window. For example:

```
var newWindow
//...
function createNewWindow()
{
   newWindow = window.open("","");
   if (newWindow != null)
   {
      newWindow.document.write("<html><head><title>Hi!</title></head>");
   }
}
```

That global variable reference continues to be available for another function — perhaps one that closes the subwindow (via the `close()` method).

When scripts in the subwindow need to communicate with objects and scripts in the originating window, you must make sure that the subwindow has an `opener` property if the level of JavaScript in the visitor's browser doesn't automatically supply one. See the discussion about the `window.opener` property earlier in this chapter.

Invoking multiple `window.open()` methods with the same window name parameter (the second parameter) does not create additional copies of that window in Netscape browsers (although it does in Internet Explorer). JavaScript prevents you from creating two windows with the same name. Also be aware that a `window.open()` method does not bring an existing window of that name to the front of the window layers: Use `window.focus()` for that.

Internet Explorer idiosyncrasies

Creating subwindows in IE can sometimes be complicated by undesirable behavior by the browser. One of the most common problems occurs when you attempt to use `document.write()` to put content into a newly created window. IE, including some of the latest versions, fails to complete the window opening job before the script statement that uses `document.write()` executes. This causes a script error because the reference to the subwindow is not yet valid. To work around this, you should put the HTML assembly and `document.write()` statements in a separate function that gets invoked via a `setTimeout()` method after the window is created. You can see an example of this in Listing 27-25.

Another problem that affects IE is the occasional security violation ("access denied") warning when a script attempts to access a subwindow. This problem goes away when the page that includes the script for opening and accessing the subwindow is served from an HTTP server rather than accessed from a local hard disk.

Example

The page rendered by Listing 27-25 displays a single button that generates a new window of a specific size that has only the status bar turned on. The script here shows all the elements necessary to create a new window that has all the right stuff on most platforms. The new window object reference is assigned to a global variable, `newWindow`. Before a new window is generated, the script looks to see whether the window has never been generated before (in which case `newWindow` would be `null`) or, for newer browsers, the window is closed. If either condition is `true`, the window is created with the `open()` method; otherwise, the existing window is brought forward with the `focus()` method.

Due to the timing problem that afflicts all IE generations, the HTML assembly and writing to the new window are separated into their own function, which is invoked after a 50-millisecond delay (other browsers go along for the ride even if they could accommodate the assembly and writing without the delay). To build the string that is eventually written to the document, we use the += (add-by-value) operator, which appends the string on the right side of the operator to the string stored in the variable on the left side. In this example, the new window is handed an `<h1>`-level line of text to display.

LISTING 27-25

Creating a New Window

HTML: jsb27-25.html

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>New Window</title>
      <script type="text/javascript" src="jsb27-25.js"></script>
   </head>
   <body>
      <h1>Creating a New Window</h1>
      <hr />
      <form>
         <input type="button" name="newOne" value="Create New Window"
                onclick="makeNewWindow()" />
      </form>
   </body>
</html>
```

JavaScript: jsb27-25.js

```
var newWindow;

function makeNewWindow()
{
   if (!newWindow || newWindow.closed)
   {
      newWindow = window.open("","","status,height=200,width=300");
      // force small delay for IE to catch up
      setTimeout("writeToWindow()", 50);
```

continued

LISTING 27-25 *(continued)*

```
      }
      else
      {
         // window's already open; bring to front
         newWindow.focus();
      }
   }

   function writeToWindow()
   {
      // assemble content for new window
      var newContent = "<html><head><title>One Sub Window<\/title><\/head>";
      newContent += "<body><h1>This window is brand new.<\/h1>";
      newContent += "<\/body><\/html>";
      // write HTML to new window document
      newWindow.document.write(newContent);
      newWindow.document.close();   // close layout stream
   }
```

The `window.open()` method can potentially create problems in browsers such as IE7+ and FF2+ that support tabbed browsing where multiple pages are opened as different tabs within the same browser instance. The default response to `window.open()` is to open a new tab for the new window, as opposed to a new browser window, which can be a problem if the script is truly expecting a completely new browser window.

Related Items: `window.close()`, `window.blur()`, `window.focus()`, `window.openDialog()` methods; `window.closed` property

openDialog("*URL*", "*windowName*" [, "*windowFeatures*"] [, *arg1*][, *arg2*] ...)

Returns: A window object representing the newly created dialog box (window); `null` if method fails

Compatibility: WinIE-, MacIE-, NN7+, Moz1.0.1+, Safari-, Opera-, Chrome-

The `window.openDialog()` method is a Mozilla-specific method that serves as a XUL counterpart to the `window.open()` method. XUL is Mozilla's XML-based user interface description language that is used throughout the Mozilla application suite. The `openDialog()` method offers a few extra window features, along with the ability to pass a varying number of arguments, which can be handy when creating custom windows.

The parameters to the `openDialog()` method are similar to those found in the `open()` method, except for the optional `arg1`, `arg2`, and so on. One notable difference is the addition of the `all` window feature, which is used to show (`all=yes`) or hide (`all=no`) all window features except `chrome`, `dialog`, and `modal`; these excluded features can still be shown or hidden individually.

Related Item: `window.open()` method

print()

Returns: Nothing

Compatibility: WinIE5+, MacIE5+, NN4+, Moz+, Safari+, Opera+, Chrome+

The `print()` method provides a scripted way of sending the window or a frame from a frameset to the printer. In all cases, the Print dialog box appears for the user to make the printer choices typical of printing manually. This prevents a rogue `print()` command from tying up a printer without the user's permission. Rather disconcertingly, the user has to reload the page to see the page content after sending the page to the printer.

WinIE5 introduced some print-specific event handlers that are triggered by scripted printing as well as manual printing. The events begin to fire after the user has accepted the Print dialog box. An `onbeforeprint` event handler can be used to show content that might be hidden from view but should appear in the printout. After the content has been sent to the print spooler, the `onafterprint` event can restore the page.

Example

Listing 27-26 is a frameset that loads Listing 27-27 into the top frame, and a copy of the Bill of Rights into the bottom frame.

LISTING 27-26

Print Example Frameset

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>window.print() method</title>
   </head>
   <frameset rows="25%,75%">
      <frame name="controls" src="jsb27-27.html" />
      <frame name="display" src="bofright.html" />
   </frameset>
</html>
```

Two buttons in the top control panel (see Listing 27-27) let you print the whole frameset (in those browsers and OS's that support it) or just the bottom frame. To print the entire frameset, the reference includes the parent window; to print the bottom frame, the reference is directed at the `parent.display` frame.

LISTING 27-27

Printing Control

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
```

continued

LISTING 27-27 *(continued)*

```
      <title>Print()</title>
   </head>
   <body>
      <h1>Print()</h1>
      <hr />
      <form>
         <input type="button" name="printWhole"
                value="Print Entire Frameset"
                onclick="parent.print()" />
         <p><input type="button" name="printFrame"
                   value="Print Bottom Frame Only"
                   onclick="parent.display.print()" />
         </p>
      </form>
   </body>
</html>
```

If you don't like some facet of the printed output, blame the browser's print engine, not JavaScript. The `print()` method merely invokes the browser's regular printing routines.

Related Items: `window.back()`, `window.forward()`, `window.home()`, `window.find()` methods

prompt(*"message"*, *"defaultReply"*)

Returns: String of text entered by user, or `null`

Compatibility: WinIE3+, MacIE3+, NN2+, Moz+, Safari+, Opera+, Chrome+

The third kind of dialog box that JavaScript can display includes a message from the script author, a field for user entry, and two buttons (OK and Cancel). The script writer can supply a prewritten answer so that a user confronted with a prompt dialog box can click OK (or press Enter) to accept that answer without further typing. Supplying both parameters to the `window.prompt()` method is important. Even if you don't want to supply a default answer, enter an empty string as the second parameter:

```
prompt("What is your postal code?","");
```

If you omit the second parameter, JavaScript inserts the string `undefined` into the dialog box's field. This message is disconcerting to most web-page visitors.

The value returned by this method is a string in the dialog box's field when the user clicks the OK button. If you're asking the user to enter a number, remember that the value returned by this method is a string. You may need to perform data-type conversion with the `parseInt()` or `parseFloat()` function (see Chapter 24, "Global Functions and Statements") to use the returned values in math calculations.

When the user clicks the prompt dialog box's OK button without entering any text in a blank field, the returned value is an empty string (`""`). Clicking the Cancel button, however, makes the method

return a `null` value. Therefore, the scripter must test for the type of returned value to make sure that the user entered some data that can be processed later in the script, as in:

```
var entry = prompt("Enter a number between 1 and 10:","");
if (entry != null)
{
   //statements to execute with the value
}
```

This script excerpt assigns the results of the prompt dialog box to a variable and executes the nested statements if the returned value of the dialog box is not `null` (if the user clicked the OK button). The rest of the statements then include data validation to make sure that the entry is a number within the desired range (see Chapter 46, "Data-Entry Validation," on the CD-ROM).

It may be tempting to use the prompt dialog box as a handy user input device. But as with the other JavaScript dialog boxes, the modality of the prompt dialog box is disruptive to the user's flow through a document and can also trap automated macros that some users activate to capture web sites. In forms, HTML fields are better user interface elements for attracting user text entry. Perhaps the safest way to use a prompt dialog box is to have it appear when a user clicks a button element on a page — and then only if the information you require of the user can be provided in a single prompt dialog box. Presenting a sequence of prompt dialog boxes is downright annoying to users.

Example

The function that receives values from the prompt dialog box in Listing 27-28 (see the dialog box in Figure 27-7) does some data-entry validation (but certainly not enough for a commercial site). The function first checks to make sure that the returned value is neither `null` (Cancel) nor an empty string (the user clicked OK without entering any values). See Chapter 46 for more about data-entry validation.

LISTING 27-28

The Prompt Dialog Box

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>window.prompt() Method</title>
      <script type="text/javascript">
         function populateTable()
         {
            var howMany = prompt("Fill in table for how many factors?","");
            if (howMany != null && howMany != "")
            {
               alert("Filling the table for " + howMany);   // for demo
               //statements that validate the entry and
               //actually populate the fields of the table
            }
         }
      </script>
```

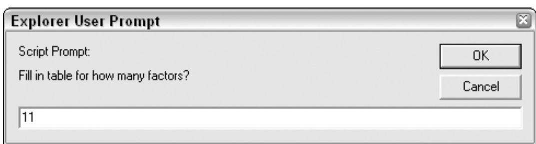
continued

LISTING 27-28 *(continued)*

```
   </head>
   <body>
      <h1>window.prompt() Method</h1>
      <hr />
      <form>
         <h3>Other statements that display and populate a large table
             would be here.</h3>
         <input type="button" name="fill" value="Fill Table..."
                onclick="populateTable()" />
      </form>
   </body>
</html>
```

FIGURE 27-7

The prompt dialog box from Listing 27-28 displayed in Internet Explorer.



Notice one important user interface element in Listing 27-28. Because clicking the button leads to a dialog box that requires more information from the user, the button's label ends in an ellipsis (or, rather, three periods acting as an ellipsis character). The ellipsis is a common courtesy to let users know that a user interface element leads to a dialog box of some sort. Consistent with stand-alone applications, the user should be able to cancel out of that dialog box and return to the same screen state that existed before the button was clicked.

Related Items: `window.alert()`, `window.confirm()` methods

resizeBy(*deltaX*,*deltaY*)
resizeTo(*outerwidth*,*outerheight*)

Returns: Nothing

Compatibility: WinIE4+, MacIE4+, NN4, Moz+, Safari 1+, Opera-, Chrome-

Starting with version 4 browsers, scripts can control the size of the current browser window on the fly (no longer available in Mozilla-based browsers). Although you can set the individual inner and (in NN4) outer width and height properties of a window, the `resizeBy()` and `resizeTo()` methods let you adjust both axis measurements in one statement. In both instances, all adjustments affect the bottom-right corner of the window. To move the top-left corner, use the `window.moveBy()` or `window.moveTo()` methods.

Each resize method requires a different kind of parameter. The `resizeBy()` method adjusts the window by a certain number of pixels along one or both axes. Therefore, it is not concerned with the specific size of the window beforehand — only by how much each axis is to change. For example, to increase the current window size by 100 pixels horizontally and 50 pixels vertically, the statement is

```
window.resizeBy(100, 50);
```

Both parameters are required, but if you want to adjust the size in only one direction, set the other to zero. You may also shrink the window by using negative values for either or both parameters.

You may find a greater need for the `resizeTo()` method, especially when you know that on a particular platform, the window needs adjustment to a specific width and height to best accommodate that platform's display of form elements. Parameters for the `resizeTo()` method are the actual pixel width and height of the outer dimension of the window — the same as NN4's `window.outerWidth` and `window.outerHeight` properties.

To resize the window so that it occupies all screen real estate (except for the Windows taskbar and Macintosh menu bar), use the `screen` object properties that calculate the available screen space:

```
window.resizeBy(screen.availWidth, screen.availHeight);
```

This action, however, is not precisely the same in Windows as maximizing the window. To achieve that same effect, you must move the window to coordinates -4, -4 and add 8 to the two parameters of `resizeBy()`:

```
window.moveTo(-4,-4);
window.resizeTo(screen.availWidth + 8, screen.availHeight + 8);
```

This hides the window's own 4-pixels-wide border, as occurs during OS-induced window maximizing. See also the `screen` object discussion (see Chapter 41, "Table and List Objects") for more OS-specific details.

On some platforms, the dimensions are applied to the inner width and height rather than outer. If a specific outer size is necessary, use the NN-specific `window.outerHeight` and `window.outerWidth` properties instead.

Navigator 4 imposes some security restrictions for maximum and minimum size for a window. For both methods, you are limited to the viewable area of the screen and visible minimums unless the page uses signed scripts (see Chapter 49). With signed scripts and the user's permission, for example, you can adjust windows beyond the available screen borders.

Example

You can experiment with the resize methods with the page in Listing 27-29. Two parts of a form let you enter values for each method. The one for `window.resize()` also lets you enter several repetitions to better see the impact of the values. Enter zero and negative values to see how those affect the method. Also test the limits of different browsers.

LISTING 27-29

Window Resize Methods

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Window Resize Methods</title>
      <style type="text/css">
         .instructions
         {
            font-weight:bold;
         }
      </style>
      <script type="text/javascript">
         function doResizeBy(form)
         {
            var x = parseInt(form.resizeByX.value);
            var y = parseInt(form.resizeByY.value);
            var count = parseInt(form.count.value);
            for (var i = 0; i < count; i++)
            {
               window.resizeBy(x, y);
            }
         }
         function doResizeTo(form)
         {
            var x = parseInt(form.resizeToX.value);
            var y = parseInt(form.resizeToY.value);
            window.resizeTo(x, y);
         }
      </script>
   </head>
   <body>
      <h1>Window Resize Methods</h1>
      <hr />
      <form>
         <div class="instructions">Enter the x and y increment,
            plus how many times the window should
            be resized by these increments:
         </div>
         Horiz:<input type="text" name="resizeByX" size="4" />
         Vert:<input type="text" name="resizeByY" size="4" />
         How Many:<input type="text" name="count" size="4" />
         <input type="button" name="ResizeBy" value="Show resizeBy()"
                onclick="doResizeBy(this.form)" />
         <hr />
         <div class="instructions">Enter the desired width
            and height of the current window:</div>
         Width:<input type="text" name="resizeToX" size="4" />
```

```
        Height:<input type="text" name="resizeToY" size="4" />
        <input type="button" name="ResizeTo" value="Show resizeTo()"
               onclick="doResizeTo(this.form)" />
      </form>
   </body>
</html>
```

Related Items: `window.outerHeight`, `window.outerWidth` properties; `window.moveTo()`, `window.sizeToContent()` methods

`scroll (`*`horizontalCoord, verticalCoord`*`)`

Returns: Nothing

Compatibility: WinIE4+, MacIE4+, NN3+, Moz+, Safari+, Opera+, Chrome+

The `window.scroll()` method was introduced in NN3 and has been implemented in all scriptable browsers since then. But in the meantime, the method has been replaced by the `window.scrollTo()` method, which is in more syntactic alliance with many other window methods.

The `window.scroll()` method takes two parameters: the horizontal (x) and vertical (y) coordinates of the document that is to be positioned at the top-left corner of the window or frame. You must realize that the window and document have two similar, but independent, coordinate schemes. From the window's point of view, the top-left pixel (of the content area) is point `0,0`. All documents also have a `0,0` point: the very top left of the document. The window's `0,0` point doesn't move, but the document's `0,0` point can move — via manual or scripted scrolling. Although `scroll()` is a window method, it seems to behave more like a document method, as the document appears to reposition itself within the window. Conversely, you can also think of the window moving to bring its `0,0` point to the designated coordinate of the document.

Although you can set values beyond the maximum size of the document or to negative values, the results vary from platform to platform. For the moment, the best usage of the `window.scroll()` method is as a means of adjusting the scroll to the very top of a document (`window.scroll(0,0)`) when you want the user to be at a base location in the document. For vertical scrolling within a text-heavy document, an HTML anchor may be a better alternative for now (though it doesn't readjust horizontal scrolling).

Related Items: `window.scrollBy()`, `window.scrollTo()` methods

Unwanted User Scrolling

Wheeled mice have become increasingly popular. These mice include a scroll wheel that is activated by pressing down on the wheel and spinning the wheel. Be aware that even if your page design loads into frames or new windows that intentionally lack scroll bars, the page will be scrollable via this wheel if the document or its background image is larger than the window or frame. Users may not even be aware that they have scrolled the page (because there are no scroll-bar visual clues). If this affects your design, you may need to build in a routine (via `setTimeout()`) that periodically sets the scroll of the window to `0,0`.

```
scrollBy(deltaX,deltaY)
scrollTo(x,y)
```

Returns: Nothing

Compatibility: WinIE4+, MacIE4+, NN4+, Moz+, Safari+, Opera+, Chrome+

Modern browsers provide a related pair of window scrolling methods. The `window.scrollTo()` method is the newer version of the `window.scroll()` method. The two work identically to position a specific coordinate point of a document at the top-left corner of the inner window region.

By contrast, the `window.scrollBy()` method allows for relative positioning of the document. Parameter values indicate by how many pixels the document should scroll in the window (horizontally and vertically). Negative numbers are allowed if you want to scroll to the left and/or up. The `scrollBy()` method comes in handy if you elect to hide the scroll bars of a window or frame and offer other types of scrolling controls for your users. For example, to scroll down one entire screen of a long document, you can use the `window.innerHeight` (in NN/Moz) or `document.body.clientHeight` (in IE) properties to determine what the offset from the current position would be:

```
// assign IE body clientHeight to window.innerHeight
if (document.body && document.body.clientHeight)
{
   window.innerHeight = document.body.clientHeight;
}
window.scrollBy(0, window.innerHeight);
```

To scroll up, use a negative value for the second parameter:

```
window.scrollBy(0, -window.innerHeight);
```

The window scroll methods are not the ones to use to produce the scrolling effect of a positioned element. That kind of animation is accomplished by adjusting `style` position properties (see Chapter 43, “Positioned Objects”).

Example

To work with the `scrollTo()` method, you can use Listing 27-30, Listing 27-31, and Listing 27-32. The code in Listing 27-34 includes a control panel frame (similar to Listing 27-32) that provides input to experiment with the `scrollBy()` method.

LISTING 27-30

Frameset for ScrollBy Controller

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>window.scrollBy() Method</title>
   </head>
```

```
   <frameset rows="50%,50%">
      <frame src="jsb27-31.html" name="display" />
      <frame src="jsb27-32.html" name="control" />
   </frameset>
</html>
```

LISTING 27-31

The Image to Be Scrolled

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Arch</title>
   </head>
   <body>
      <h1>
         A Picture is Worth...
      </h1>
      <hr />
      <center>
         <table border="3">
            <caption align="bottom">
               A Splendid Arch
            </caption>
            <tr>
               <td>
                  <img alt="image" src="arch.gif" />
               </td>
            </tr>
         </table>
      </center>
   </body>
</html>
```

Notice in Listing 27-32 that all references to window properties and methods are directed to the `display` frame. String values retrieved from text fields are converted to numbers with the `parseInt()` global function.

LISTING 27-32

ScrollBy Controller

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>ScrollBy Controller</title>
      <script type="text/javascript">
```

continued

LISTING 27-32 *(continued)*

```
         function page(direction)
         {
            var pixFrame = parent.display;
            var deltaY = (pixFrame.innerHeight) ? pixFrame.innerHeight :
               pixFrame.document.body.scrollHeight;
            if (direction == "up")
            {
               deltaY = -deltaY;
            }
            parent.display.scrollBy(0, deltaY);
         }
         function customScroll(form)
         {
            parent.display.scrollBy(parseInt(form.x.value), parseInt(form.y.value));
         }
      </script>
   </head>
   <body>
      <h1>ScrollBy Controller</h1>
      <hr />
      <form name="custom">
         Enter an Horizontal increment
         <input type="text" name="x" value="0" size="4" />
         and Vertical
         <input type="text" name="y" value="0" size="4" /> value.
         <br />
         Then
         <input type="button" value="click to scrollBy()"
                onclick="customScroll(this.form)" />
         <hr />
         <input type="button" value="PageDown" onclick="page('down')" />
         <input type="button" value="PageUp" onclick="page('up')" />
      </form>
   </body>
</html>
```

Related Items: `window.pageXOffset`, `window.pageYOffset` properties; `window.scroll()` method

`scrollByLines(`*`intervalCount`*`)`
`scrollByPages(`*`intervalCount`*`)`

Returns: Nothing

Compatibility: WinIE-, MacIE-, NN6+, Moz+, Safari-, Opera-, Chrome-

The `window.scrollByLines()` and `window.scrollByPages()` methods scroll the document by a specified number of lines or pages, respectively. You can think of these methods as the script equivalents to clicking the arrow (`scrollByLines()`) and page (`scrollByPages()`) regions of

the browser's vertical scroll bar. The argument to each method determines how many lines or pages to scroll, with positive values resulting in downward scrolling, and negative values resulting in upward scrolling.

setInterval("*expr*", *msecDelay* [, *language*])
setInterval(funcRef, msecDelay [, funcarg1, ... , funcargn])

Returns: Interval ID integer

Compatibility: WinIE4+, MacIE4+, NN4+, Moz+, Safari+, Opera+, Chrome+

It is important to understand the distinction between the `setInterval()` and `setTimeout()` methods. Before the `setInterval()` method was part of JavaScript, authors replicated the behavior with `setTimeout()`, but the task often required reworking scripts a bit.

Use `setInterval()` when your script needs to call a function or execute some expression repeatedly with a fixed time delay between calls to that function or expression. The delay is not at all like a wait state in some languages: Other processing does not halt while the delay is in effect. Typical applications include animation by moving an object around the page under controlled speed (instead of letting the JavaScript interpreter whiz the object through its path at CPU-dependent speeds). In a kiosk application, you can use `setInterval()` to advance slides that appear in other frames or as layers, perhaps changing the view every 10 seconds. Clock displays and countdown timers would also be suitable uses of this method (even though you see examples in this book that use the old-fashioned `setTimeout()` way to perform timer and clock functions).

By contrast, `setTimeout()` is best suited for those times when you need to carry out a function or expression once in the future — even if that future is only a second or two away. In other words, `setTimeout()` gives you a one-shot timer, whereas `setInterval()` gives you a recurring timer. See the discussion of the `setTimeout()` method later in this chapter for details on this application.

Although the primary functionality of the `setInterval()` method is the same in all browsers, NN/Moz and IE offer some extra possibilities depending on the way you use parameters to the method. For simple invocations of this method, the same parameters work in all browsers that support the method. First, we address the parameters that all browsers have in common.

The first parameter of the `setInterval()` method is the name of the function or expression to run after the interval elapses. This item must be a quoted string. If the parameter is a function, no function arguments are allowed inside the function's parentheses unless the arguments are literal strings (but see the next section, "Passing function parameters").

The second parameter of this method is the number of milliseconds (1,000 per second) that JavaScript should use as the interval between invocations of the function or expression. Even though the measure is in extremely small units, don't rely on 100 percent accuracy of the intervals. Various other internal processing delays may throw off the timing just a bit.

Just like `setTimeout()`, `setInterval()` returns an integer value that is the ID for the interval process. That ID value lets you turn off the process with the `clearInterval()` method, which takes the ID value as its sole parameter. This mechanism allows for the setting of multiple running interval processes and gives your scripts the power to stop individual processes at any time without interrupting the others.

IE4+ uses the optional third parameter to specify the scripting language of the statement or function being invoked in the first parameter. As long as you are scripting exclusively in JavaScript (the same as JScript), there is no need to include this parameter.

Passing function parameters

NN4+/Moz provides a mechanism for easily passing evaluated parameters to a function invoked by `setInterval()`. To use this mechanism, the first parameter of `setInterval()` must not be a string, but a reference to the function (no trailing parentheses). The second parameter remains the amount of delay. But beginning with the third parameter, you can include evaluated function arguments as a comma-delimited list:

```
intervalID = setInterval(cycleAnimation, 500, "figure1");
```

The function definition receives those parameters in the same form as any function:

```
function cycleAnimation(elemID) {...}
```

For use with a wider range of browsers, you can also cobble together the ability to pass parameters to a function invoked by `setInterval()`. Because the call to the other function is a string expression, you can use computed values as part of the strings, via string concatenation. For example, if a function uses event handling to find the element that a user clicked (to initiate some animation sequence), that element's ID, referenced by a variable, can be passed to the function invoked by `setInterval()`:

```
function findAndCycle()
{
   var elemID;
   // statements here that examine the event info
   // and extract the ID of the clicked element,
   // assigning that ID to the elemID variable
   intervalID = setInterval("cycleAnimation(" + elemID + ")", 500);
}
```

If you need to pass ever-changing parameters with each invocation of the function from `setInterval()`, look instead to using `setTimeout()` at the end of a function to invoke that same function again.

Example

The demonstration of the `setInterval()` method entails a two-framed environment. The frameset-ting document is shown in Listing 27-33.

LISTING 27-33

SetInterval() Demonstration Frameset

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
<title>setInterval() Method</title>
   </head>
   <frameset rows="50%,50%">
      <frame src="jsb27-34.html" name="control" />
      <frame src="bofright.html" name="display" />
   </frameset>
</html>
```

In the top frame is a control panel with several buttons that control the automatic scrolling of the Bill of Rights text document in the bottom frame. Listing 27-34 shows the control-panel document. Many functions here control the interval, scrolling jump size, and direction, and they demonstrate several aspects of applying `setInterval()`.

Notice that in the beginning the script establishes several global variables. Three of them are parameters that control the scrolling; the last one is for the ID value returned by the `setInterval()` method. The script needs that value to be a global value so that a separate function can halt the scrolling with the `clearInterval()` method.

All scrolling is performed by the `autoScroll()` function. For the sake of simplicity, all controlling parameters are global variables. In this application, placement of those values in global variables helps the page restart autoscrolling with the same parameters as it had when it last ran.

LISTING 27-34

SetInterval() Control Panel

HTML: jsb27-34.html

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>ScrollBy Controller</title>
      <script type="text/javascript" src="jsb27-34.js"></script>
   </head>
   <body onload="startScroll()">
      <h1>AutoScroll by setInterval() Controller</h1>
      <hr />
      <form name="custom">
         <input type="button" value="Start Scrolling"
                onclick="startScroll()" />
         <input type="button" value="Stop Scrolling"
                onclick="stopScroll()" />
         <p>
            <input type="button" value="Shorter Time Interval"
                   onclick="reduceInterval()" />
            <input type="button" value="Longer Time Interval"
                   onclick="increaseInterval()" />
         </p>
         <p><input type="button" value="Bigger Scroll Jumps"
                   onclick="increaseJump()" />
            <input type="button" value="Smaller Scroll Jumps"
                   onclick="reduceJump()" />
         </p>
         <p><input type="button" value="Change Direction"
                   onclick="swapDirection()" />
         </p>
      </form>
   </body>
</html>
```

continued

LISTING 27-34 *(continued)*

```
JavaScript: jsb27-34.js

var scrollSpeed = 500;
var scrollJump = 1;
var scrollDirection = "down";
var intervalID;

function autoScroll()
{
   if (scrollDirection == "down")
   {
      scrollJump = Math.abs(scrollJump);
   }
   else if (scrollDirection == "up" && scrollJump > 0)
   {
      scrollJump = -scrollJump;
   }
   parent.display.scrollBy(0, scrollJump);
   if (parent.display.pageYOffset <= 0)
   {
      clearInterval(intervalID);
   }
}

function reduceInterval()
{
   stopScroll();
   scrollSpeed -= 200;
   startScroll();
}
function increaseInterval()
{
   stopScroll();
   scrollSpeed += 200;
   startScroll();
}
function reduceJump()
{
   scrollJump -= 2;
}
function increaseJump()
{
   scrollJump += 2;
}
function swapDirection()
{
   scrollDirection = (scrollDirection == "down") ? "up" : "down";
}
function startScroll()
```

```
{
   parent.display.scrollBy(0, scrollJump);
   if (intervalID)
   {
      clearInterval(intervalID);
   }
   intervalID = setInterval("autoScroll()",scrollSpeed);
}
function stopScroll()
{
   clearInterval(intervalID);
}
```

The `setInterval()` method is invoked inside the `startScroll()` function. This function initially burps the page by one `scrollJump` interval so that the test in `autoScroll()` for the page being scrolled all the way to the top doesn't halt a page from scrolling before it gets started. Notice, too, that the function checks for the existence of an interval ID. If one is there, it is cleared before the new one is set. This is crucial within the design of the example page, because repeated clicking of the Start Scrolling button triggers multiple interval timers inside the browser. Only the most recent one's ID would be stored in `intervalID`, allowing no way to clear the older ones. But this little side trip makes sure that only one interval timer is running. One of the global variables, `scrollSpeed`, is used to fill the delay parameter for `setInterval()`. To change this value on the fly, the script must stop the current interval process, change the `scrollSpeed` value, and start a new process.

The intensely repetitive nature of this application is nicely handled by the `setInterval()` method.

Related Items: `window.clearInterval()`, `window.setTimeout()` methods

setTimeout("*expr*", *msecDelay* [, *language*])
setTimeout(*functionRef*, *msecDelay* [, *funcarg1*, ... , *funcargn*])

Returns: ID value for use with `window.clearTimeout()` method

Compatibility: WinIE3+, MacIE3+, NN2+, Moz+, Safari+, Opera+, Chrome+

The name of this method may be misleading, especially if you have done other kinds of programming involving timeouts. In JavaScript, a *timeout* is an amount of time (in milliseconds) before a stated expression evaluates. A timeout is not a wait or script delay, but a way to tell JavaScript to hold off executing a statement or function for a desired amount of time. Other statements following the one containing `setTimeout()` execute immediately.

Suppose that you have a web page designed to enable users to interact with a variety of buttons or fields within a time limit (this is a web page running at a freestanding kiosk). You can turn on the timeout of the window so that if no interaction occurs with specific buttons or fields lower in the document after, say, 2 minutes (120,000 milliseconds), the window reverts to the top of the document or to a help screen. To tell the window to switch off the timeout after a user does navigate within the allotted time, you need to have any button that the user interacts with call the other side of a `setTimeout()` method — the `clearTimeout()` method — to cancel the current timer. (The `clearTimeout()` method is explained earlier in this chapter.) Multiple timers can run concurrently and are completely independent of one another.

Although the primary functionality of the `setTimeout()` method is the same in all browsers, NN/Moz and IE offer some extra possibilities depending on the way you use parameters to the method. For simple invocations of this method, the same parameters work in all browsers that support the method. We first address the parameters that all browsers have in common.

The expression that comprises the first parameter of the method `window.setTimeout()` is a quoted string that can contain either a call to any function or method, or a stand-alone JavaScript statement. The expression evaluates after the time limit expires.

Understanding that this timeout does not halt script execution is very important. In fact, if you use a `setTimeout()` method in the middle of a script, the succeeding statements in the script execute immediately; after the delay time, the expression in the `setTimeout()` method executes. Therefore, we've found that the best way to design a timeout in a script is to plug it in as the last statement of a function: Let all other statements execute and then let the `setTimeout()` method appear to halt further execution until the timer goes off. In truth, however, although the timeout is holding, the user is not prevented from performing other tasks. And after a timeout timer is ticking, you cannot adjust its time. Instead, clear the timeout and start a new one.

If you need to use `setTimeout()` as a delay inside a function, break the function into two parts, using the `setTimeout()` method as a bridge between the two functions. You can see an example of this in Listing 27-25, where IE needs a little delay to finish opening a new window before content can be written for it. If it weren't for the required delay, the HTML assembly and writing would have been accomplished in the same function that opens the new window.

It is not uncommon for a `setTimeout()` method to invoke the very function in which it lives. For example, if you have written a Java applet to perform some extra work for your page, and you need to connect to it via the NPAPI, your scripts must wait for the applet to load and carry out its initializations. Although an `onload` event handler in the document ensures that the applet object is visible to scripts, it doesn't know whether the applet has finished its initializations. A JavaScript function that inspects the applet for a clue might need to poll the applet every 500 milliseconds until the applet sets some internal value indicating that all is ready, as shown here:

```
var t;
function autoReport()
{
   if (!document.myApplet.done)
   {
      t = setTimeout("autoReport()",500);
   }
   else
   {
      clearTimeout(t);
      // more statements using applet data //
   }
}
```

JavaScript provides no built-in equivalent for a `wait` command. The worst alternative is to devise a looping function of your own to trap script execution for a fixed amount of time. Unfortunately, this approach prevents other processes from being carried out, so you should consider reworking your code to rely on a `setTimeout()` method instead.

NN4+/Moz provides a mechanism for passing parameters to functions invoked by `setTimeout()`. See the section "Passing function parameters" in the discussion of `window.setInterval()` for details on this topic, and on passing parameters in other browser versions.

As a note to experienced programmers, neither `setInterval()` nor `setTimeout()` spawns new threads in which to run its invoked scripts. When the timer expires and invokes a function, the process gets at the end of the pending script processing queue in the JavaScript execution thread.

Example

When you load the HTML page in Listing 27-35, it triggers the `updateTime()` function, which displays the time (in hh:mm am/pm format) in the status bar. Instead of showing the seconds incrementing one by one (which may be distracting to someone trying to read the page), this function alternates the last character of the display between an asterisk and nothing, like a visual heartbeat.

LISTING 27-35

Display the Current Time

```
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Status Bar Clock</title>
      <script type="text/javascript" src="../jsb-global.js"></script>
      <script type="text/javascript">
         // initialize when the page has loaded
         addEvent(window, "load", updateTime);

         var flasher = false;
         // calculate current time, determine flasher state,
         // and insert time into status bar every second
         function updateTime()
         {
            var now = new Date();
            var theHour = now.getHours();
            var theMin = now.getMinutes();
            var theTime = "" + ((theHour > 12) ? theHour - 12 : theHour);
            theTime += ((theMin < 10) ? ":0" : ":") + theMin;
            theTime  += (theHour >= 12) ? " pm" : " am";
            theTime += ((flasher) ? " " : "*");
            flasher = !flasher;
            window.status = theTime;
            // recursively call this function every second to keep timer going
            timerID = setTimeout("updateTime()",1000);
         }
      </script>
   </head>
   <body>
      <h1>Status Bar Clock</h1>
      <hr />
   </body>
</html>
```

In this function, the `setTimeout()` method works in the following way: When the current time (including the flasher status) appears in the status bar, the function waits approximately 1 second

(1,000 milliseconds) before calling the same function again. You don't have to clear the `timerID` value in this application because JavaScript does it for you every time the 1,000 milliseconds elapse.

A logical question to ask is whether this application should be using `setInterval()` instead of `setTimeout()`. This is a case in which either one does the job. Using `setInterval()` here would require that the interval process start outside the `updateTime()` function, because you need only one process running that repeatedly calls `updateTime()`. It would be a cleaner implementation instead of the numerous timeout processes spawned by Listing 27-35. On the other hand, the application would not run in legacy browsers, as Listing 27-35 does. That's likely not a problem at this point in time, but this application remains a decent example of the `setTimeout()` function.

To demonstrate passing parameters, you can modify the `updateTime()` function to add the number of times it gets invoked to the display in the status bar. For that to work, the function must have a parameter variable so that it can catch a new value each time it is invoked by `setTimeout()`'s expression. For all browsers, the function would be modified as follows (unchanged lines are represented by the ellipsis):

```
function updateTime(i)
{
   //...
   window.status = theTime + "  (" + i + ")";
   // pass updated counter value with next call to this function
   timerID = setTimeout("updateTime(" + i+1 + ")",1000);
}
```

If you were running this exclusively in NN4+/Moz, you could use its more convenient way of passing parameters to the function:

```
timerID = setTimeout(updateTime, 1000, i+1);
```

In either case, the `onload` event handler would also have to be modified to get the ball rolling with an initial parameter:

```
onload = "updateTime(0)";
```

Related Items: `window.clearTimeout()`, `window.setInterval()`, `window.clearInterval()` methods

showHelp("*URL*",["*contextID*"])

Returns: Nothing

Compatibility: WinIE4+, MacIE-, NN-, Moz-, Safari-, Opera-, Chrome-

The IE-specific `showHelp()` method lets a script open a Winhelp window with a particular `.hlp` file. This method is specific to Win32 operating systems.

If your Winhelp file has context identifiers specified in various places, you can pass the ID as an optional second parameter. This lets the call to `showHelp()` navigate to the particular area of the `.hlp` file that applies to a specific element on the page.

See the Microsoft Visual Studio authoring environment for details on building Winhelp files.

showModalDialog("*URL*"[, *arguments*][, *features*])
showModelessDialog("*URL*"[, *arguments*][, *features*])

Returns: `returnValue` (modal) or `window` object (modeless)

Compatibility: WinIE4+, MacIE4+, NN-, Moz+, Safari 2.01+, Opera-, Chrome+

IE, Safari, Chrome, and Mozilla-based browsers such as Firefox, provide methods for opening a modal dialog-box window, which always stays in front of the main browser window, while making the main window inaccessible to the user. In WinIE5, Microsoft added the modeless type of dialog box, which is not supported by Safari, Chrome and Mozilla-based browsers such as Firefox. The modeless type of dialog box also stays in front but allows user access to whatever can be seen in the main window. You can load any HTML page or image that you like into the dialog-box window by providing a URL as the first parameter. Optional parameters let you pass data to a dialog box and give you considerable control over the look of the window. A similar type of dialog-box window is available in NN/Moz via the `window.openDialog()` method.

The windows generated by both methods are (almost) full-fledged `window` objects, with some extra properties that are useful for what these windows are intended to do. Perhaps the most important property is the `window.dialogArgument` property. This property lets a script read the data that is passed to the window via the second parameter of both `showModalDialog()` and `showModelessDialog()`. Passed data can be in any valid JavaScript data type, including objects and arrays.

Displaying a modal dialog box has some ramifications for scripts. In particular, script execution in the main window halts at the statement that invokes the `showModalDialog()` method as long as the modal dialog box remains visible. Scripts are free to run in the dialog-box window during this time. The instant the user closes the dialog box, execution resumes in the main window. A call to show a modeless dialog box, on the other hand, does not halt processing because scripts in the main page or dialog-box window are allowed to communicate live with the other window.

Retrieving dialog-box data

To send data back to the main window's script from a modal dialog-box window, a script in the dialog-box window can set the `window.returnValue` property to any JavaScript value. It is this value that gets assigned to the variable receiving the returned value from the `setModelDialog()` method, as shown in the following example:

```
var specifications = window.showModalDialog("preferences.html");
```

The makeup and content of the returned data are in the hands of your scripts. No data is automatically returned for you.

Because a modeless dialog box coexists with your live main page window, returning data is not as straightforward as for a modal dialog box. The second parameter of the `showModelessDialog()` method takes on a special task that isn't exactly the same as passing parameters to the dialog box. Instead, if you define a global variable or a function in the main window's script, pass a reference to that variable or function as the second parameter to display the modeless dialog box. A script in the modeless dialog box can then point to that reference as the way to send data back to the main window before the dialog box closes (or when a user clicks something, such as an Apply button). This

mechanism even allows for passing data back to a function in the main window. For example, say that the main window has a function defined as the following:

```
function receivePrefsDialogData(a, b, c)
{
   // statements to process incoming values //
}
```

Then pass a reference to this function when opening the window:

```
dlog = showModelessDialog("prefs.html", receivePrefsDialogData);
```

A script statement in the dialog-box window's document can pick up that reference so that other statements, such as a function for an Apply button's `onclick` event handler, can use it:

```
var returnFunc = window.dialogArguments;
//...
function apply(form)
{
   returnFunc(form.color.value, form.style.value, form.size.value);
}
```

Although this approach seems to block ways of getting parameters to the dialog box when it opens, you can always reference the dialog box in the main window's script and set form or variable values directly:

```
dlog = showModelessDialog("prefs.html", receivePrefsDialogData);
dlog.document.forms[0].userName.value = GetCookie("userName");
```

Be aware that a dialog-box window opened with either of these methods does not maintain a connection to the originating window via the `opener` property. The `opener` property for both dialog-box types is undefined.

Dialog-box window features

Both methods provide an optional third property that lets you specify visible features of the dialog-box window. Omitting the property sets all features to their default values. All parameters are to be contained by a single string, and each parameter's name–value pair is in the form of CSS `attribute:value` syntax. Table 27-5 lists all the window features available for the two window styles. If you are designing for backward compatibility with IE4, you are restricted to the modal dialog box and a subset of features, as noted in the table. All values listed as Boolean take only the following four values: `yes`, `no`, `1`, and `0`.

The CSS-type of syntax for these features lets you thread multiple features together by separating each pair with a semicolon within the string. For example:

```
var dlogData = showModalDialog("prefs.html", defaultData,
   "dialogHeight:300px; dialogWidth:460px; help:no");
```

Although they are not explicitly listed among the window features, scroll bars are normally displayed in the window if the content exceeds the size assigned or available to the dialog box. If you don't want scroll bars to appear, have your dialog-box document's script set the `document.body.scroll` property to `false` as the page opens.

TABLE 27-5

IE Dialog-Box Window Features

Feature	Type	Default	Description
`center`	Boolean	`yes`	Whether to center dialog box (overridden by `dialogLeft` and/or `dialogTop`).
`dialogHeight`	Length	Varies	Outer height of the dialog-box window. IE4 default length unit is `em`; IE5+/Moz/Safari/Chrome is pixel (`px`).
`dialogLeft`	Integer	Varies	Pixel offset of dialog box from left edge of screen.
`dialogTop`	Integer	Varies	Pixel offset of dialog box from top edge of screen.
`dialogWidth`	Length	Varies	Outer width of the dialog-box window. IE4 default length unit is `em`; IE5+/Moz/Safari/Chrome is pixel (`px`).
`edge`	String	`raised` | `sunken`	Border style (IE only).
`resizable`	Boolean	`no`	Dialog box is resizable.
`scroll`	Boolean	`yes`	Display scrollbars
`status`	Boolean	Varies	Display status bar at window bottom (IE5+/Safari/Chrome only). Default is `yes` for untrusted dialog box, `no` for trusted dialog box.

Dialog-box cautions

A potential user problem to watch for is that typically, a dialog-box window does not open until the HTML file for the dialog box has loaded. Therefore, if there is substantial delay before a complex document loads, the user does not see any action indicating that something is happening. You may want to experiment with setting the `cursor` style sheet property and restoring it when the dialog box's document loads.

Example

To demonstrate the two styles of dialog boxes, we have implemented the same functionality (setting some session visual preferences) for both modal and modeless dialog boxes. This tactic shows you how to pass data back and forth between the main page and both styles of dialog-box windows.

The first example demonstrates how to use a modal dialog box. In the process, data is passed into the dialog-box window, and values are returned. Listing 27-36 is the HTML and scripting for the main page. A button's `onclick` event handler invokes a function that opens the modal dialog box. The dialog box's document (see Listing 27-37) contains several form elements for entering a user name and selecting a few color styles for the main page. Data from the dialog box is fashioned into an array to be sent back to the main window. That array is initially assigned to a local variable, `prefs`, as the dialog box closes. If the user cancels the dialog box, the returned value is an empty string, so nothing more in `getPrefsData()` executes. But when

the user clicks OK, the array comes back. Each of the array items is read and assigned to its respective form value or style property. These values are also preserved in the global `currPrefs` array. This allows the settings to be sent to the modal dialog box (as the second parameter to `showModalDialog()`) the next time the dialog box is opened.

LISTING 27-36

Main Page for showModalDialog()

HTML: jsb27-36.html

```
<html>
   <head>
      <title>window.setModalDialog() Method</title>
      <script type="text/javascript" src="jsb27-36.js"></script>
   </head>
   <body bgcolor="#EEEEEE" style="margin:20px" onload="init()">
      <h1>window.setModalDialog() Method</h1>
      <hr />
      <h2 id="welcomeHeader">Welcome, <span id="firstName">.</span>!</h2>
      <hr />
      <p>Use this button to set style preferences for this page:
         <button id="prefsButton" onclick="getPrefsData()">Preferences</button>
      </p>
   </body>
</html>
```

JavaScript: jsb27-36.js

```
var currPrefs = new Array();
var nameHolder;

function getPrefsData()
{
   var prefs = showModalDialog("jsb27-37.html", currPrefs,
               "dialogWidth:400px; dialogHeight:300px;");
   if (prefs)
   {
      if (prefs["name"])
      {
         if (nameHolder.textContent == "friend")
         {
            nameHolder.textContent = prefs["name"];
         }
         else
         {
            //IE
            nameHolder.innerText = prefs["name"];
         }
         currPrefs["name"] = prefs["name"];
      }
```

```
      if (prefs["bgColor"])
      {
         document.body.style.backgroundColor = prefs["bgColor"];
         currPrefs["bgColor"] = prefs["bgColor"];
      }
      if (prefs["textColor"])
      {
         document.body.style.color = prefs["textColor"];
         currPrefs["textColor"] = prefs["textColor"];
      }
      if (prefs["h1Size"])
      {
         if (document.all)
         {
            document.all.welcomeHeader.style.fontSize = prefs["h1Size"];
         }
         else
         {
            document.getElementById("welcomeHeader").style.fontSize
            = prefs["h1Size"];
         }
         currPrefs["h1Size"] = prefs["h1Size"];
      }
   }
}
function init()
{
   nameHolder = document.getElementById("firstName");
   if (nameHolder.textContent == ".")
   {
      nameHolder.textContent = "friend";
   }
   else
   {
      //IE
      nameHolder.innerText = "friend";
   }
}
```

The dialog box's document, shown in Listing 27-37, is responsible for reading the incoming data (and setting the form elements accordingly) and assembling form data for return to the main window's script. Notice when you load the example that the `title` element of the dialog box's document appears in the dialog-box window's title bar.

When the page loads into the dialog-box window, the `init()` function examines the `window.dialogArguments` property. If it has any data, the data is used to preset the form elements to mirror the current settings of the main page. A utility function, `setSelected()`, preselects the option of a `select` element to match the current settings.

Buttons at the bottom of the page are explicitly positioned to be at the bottom-right corner of the window. Each button invokes a function to do what is needed to close the dialog box. In the case of the OK button, the `handleOK()` function sets the `window.returnValue` property to the data

that comes back from the `getFormData()` function. This latter function reads the form element values and packages them in an array using the form elements' names as array indices. This helps keep everything straight back in the main window's script, which uses the index names, and therefore is not dependent upon the precise sequence of the form elements in the dialog-box window.

LISTING 27-37

Document for the Modal Dialog Box

HTML: jsb27-37.html

```
<html>
   <head>
      <title>User Preferences</title>
      <script type="text/javascript" src="jsb27-37.js"></script>
   </head>
   <body bgcolor="#EEEEEE" onload="init()">
      <h2>Web Site Preferences</h2>
      <hr />
      <form name="prefs" onsubmit="return false">
         <table>
            <tr>
               <td>Enter your first name:
                  <input name="name" type="text" value=""
                         size="20" onkeydown="checkEnter()" />
               </td>
            </tr>
            <tr>
               <td>Select a background color:
                  <select name="bgColor">
                     <option value="beige">Beige</option>
                     <option value="antiquewhite">Antique White</option>
                     <option value="goldenrod">Goldenrod</option>
                     <option value="lime">Lime</option>
                     <option value="powderblue">Powder Blue</option>
                     <option value="slategray">Slate Gray</option>
                  </select>
               </td>
            </tr>
            <tr>
               <td>Select a text color:
                  <select name="textColor">
                     <option value="black">Black</option>
                     <option value="white">White</option>
                     <option value="navy">Navy Blue</option>
                     <option value="darkorange">Dark Orange</option>
                     <option value="seagreen">Sea Green</option>
                     <option value="teal">Teal</option>
                  </select>
               </td>
            </tr>
            <tr>
               <td>Select "Welcome" heading font point size:
```

```
                <select name="h1Size">
                   <option value="12">12</option>
                   <option value="14">14</option>
                   <option value="18">18</option>
                   <option value="24">24</option>
                   <option value="32">32</option>
                   <option value="48">48</option>
                </select>
             </td>
          </tr>
       </table>
    </form>
    <div style="position:absolute; left:200px; top:220px">
       <button style="width:80px"
               onclick="handleOK()">OK</button>
         
       <button style="width:80px"
               onclick="handleCancel()">Cancel</button>
    </div>
  </body>
</html>
```

JavaScript: jsb27-37.js

```
// Close the dialog
function closeme()
{
   window.close();
}

// Handle click of OK button
function handleOK()
{
   window.returnValue = getFormData();
   closeme();
}

// Handle click of Cancel button
function handleCancel()
{
   window.returnValue = "";
   closeme();
}

// Generic function converts form element name-value pairs
// into an array
function getFormData()
{
   var form = document.prefs;
   var returnedData = new Array();
   // Harvest values for each type of form element
   for (var i = 0; i < form.elements.length; i++)
```

continued

LISTING 27-37 *(continued)*

```
    {
       if (form.elements[i].type == "text")
       {
          returnedData[form.elements[i].name] = form.elements[i].value;
       }
       else if (form.elements[i].type.indexOf("select") != -1)
       {
          returnedData[form.elements[i].name] =
            form.elements[i].options[form.elements[i].selectedIndex].value;
       }
       else if (form.elements[i].type == "radio")
       {
          returnedData[form.elements[i].name] = form.elements[i].value;
       }
       else if (form.elements[i].type == "checkbox")
       {
          returnedData[form.elements[i].name] = form.elements[i].value;
       }
       else
       {
          continue;
       }
    }
    return returnedData;
}

// Initialize by setting form elements from passed data
function init()
{
    if (window.dialogArguments)
    {
       var args = window.dialogArguments;
       var form = document.prefs;
       if (args["name"])
       {
          form.name.value = args["name"];
       }
       if (args["bgColor"])
       {
          setSelected(form.bgColor, args["bgColor"]);
       }
       if (args["textColor"])
       {
          setSelected(form.textColor, args["textColor"]);
       }
       if (args["h1Size"])
       {
          setSelected(form.h1Size, args["h1Size"]);
       }
```

```
   }
}

// Utility function to set a SELECT element to one value
function setSelected(select, value)
{
   for (var i = 0; i < select.options.length; i++)
   {
      if (select.options[i].value == value)
      {
         select.selectedIndex = i;
         break;
      }
   }
   return;
}

// Utility function to accept a press of the
// Enter key in the text field as a click of OK
function checkEnter()
{
   if (window.event.keyCode == 13)
   {
      handleOK();
   }
}
```

One last convenience feature of the dialog-box window is the `onkeypress` event handler in the text box. The function it invokes looks for the Enter key. If that key is pressed while the box has focus, the same `handleOK()` function is invoked, as though the user had clicked the OK button. This feature makes the dialog box behave as though the OK button is an automatic default, just as in real dialog boxes.

You should observe several important structural changes that were made to turn the modal approach into a modeless one. Listing 27-38 shows a version of the main window modified for use with a modeless dialog box. Another global variable, `prefsDlog`, is initialized to eventually store the reference to the modeless window returned by the `showModelessWindow()` method. The variable gets used to invoke the `init()` function inside the modeless dialog box, but also as a condition in an `if` construction surrounding the generation of the dialog box. The reason this is needed is to prevent multiple instances of the dialog box from being created (the button is still alive while the modeless window is showing). The dialog box won't be created again as long as there is a value in `prefsDlog` and the dialog-box window has not been closed (picking up the `window.closed` property of the dialog-box window).

The `showModelessDialog()` method's second parameter is a reference to the function in the main window that updates the main document. As you see in a moment, that function is invoked from the dialog box when the user clicks the OK or Apply button.

LISTING 27-38

Main Page for showModelessDialog()

HTML: jsb27-38.html

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>window.setModelessDialog() Method</title>
      <script type="text/javascript" src="jsb27-38.js"></script>
   </head>
   <body bgcolor="#EEEEEE" style="margin:20px" onload="init()">
      <h1>window.setModelessDialog() Method</h1>
      <hr />
      <h2 id="welcomeHeader">Welcome, <span id="firstName"> </span>!</h2>
      <hr />
      <p>Use this button to set style preferences for this page:
      <button id="prefsButton"
              onclick="getPrefsData()">Preferences</button>
      </p>
   </body>
</html>
```

JavaScript: jsb27-38.js

```
var currPrefs = new Array();
var prefsDlog;
function getPrefsData()
{
   if (!prefsDlog || prefsDlog.closed)
   {
      prefsDlog = showModelessDialog("jsb27-39.html", setPrefs,
                  "dialogWidth:400px; dialogHeight:300px");
      prefsDlog.init(currPrefs);
   }
}

function setPrefs(prefs)
{
   if (prefs["bgColor"])
   {
      document.body.style.backgroundColor = prefs["bgColor"];
      currPrefs["bgColor"] = prefs["bgColor"];
   }
   if (prefs["textColor"])
   {
      document.body.style.color = prefs["textColor"];
      currPrefs["textColor"] = prefs["textColor"];
   }
   if (prefs["h1Size"])
```

```
   {
      document.all.welcomeHeader.style.fontSize = prefs["h1Size"];
      currPrefs["h1Size"] = prefs["h1Size"];
   }
   if (prefs["name"])
   {
      document.all.firstName.innerText = prefs["name"];
      currPrefs["name"] = prefs["name"];
   }
}

function init()
{
   document.all.firstName.innerText = "friend";
}
```

Changes to the dialog-box window document for a modeless version (see Listing 27-39) are rather limited. A new button is added to the bottom of the screen for an Apply button. As in many dialog-box windows you see in Microsoft products, the Apply button lets current settings in dialog boxes be applied to the current document, but without closing the dialog box. This approach makes experimenting with settings easier.

The Apply button invokes a `handleApply()` function, which works the same as `handleOK()`, except that the dialog box is not closed. But these two functions communicate back to the main window differently from a modal dialog box. The main window's processing function is passed as the second parameter of `showModelessDialog()` and is available as the `window.dialogArguments` property in the dialog-box window's script. That function reference is assigned to a local variable in both functions, and the remote function is invoked, passing the results of the `getFormData()` function as parameter values back to the main window.

LISTING 27-39

Document for the Modeless Dialog Box

HTML: jsb27-39.html

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>User Preferences</title>
      <script type="text/javascript" src="jsb27-39.js"></script>
   </head>
   <body bgcolor="#EEEEEE" onload="init()">
      <h2>Web Site Preferences</h2>
      <hr />
      <form name="prefs" onsubmit="return false">
         <table>
            <tr>
               <td>Enter your first name:
```

continued

LISTING 27-39 *(continued)*

```
            <input name="name" type="text" value=""
                   size="20" onkeydown="checkEnter()" />
         </td>
      </tr>
      <tr>
         <td>Select a background color:
            <select name="bgColor">
               <option value="beige">Beige</option>
               <option value="antiquewhite">Antique White</option>
               <option value="goldenrod">Goldenrod</option>
               <option value="lime">Lime</option>
               <option value="powderblue">Powder Blue</option>
               <option value="slategray">Slate Gray</option>
            </select>
         </td>
      </tr>
      <tr>
         <td>Select a text color:
            <select name="textColor">
               <option value="black">Black</option>
               <option value="white">White</option>
               <option value="navy">Navy Blue</option>
               <option value="darkorange">Dark Orange</option>
               <option value="seagreen">Sea Green</option>
               <option value="teal">Teal</option>
            </select>
         </td>
      </tr>
      <tr>
         <td>Select "Welcome" heading font point size:
            <select name="h1Size">
               <option value="12">12</option>
               <option value="14">14</option>
               <option value="18">18</option>
               <option value="24">24</option>
               <option value="32">32</option>
               <option value="48">48</option>
            </select>
         </td>
      </tr>
   </table>
</form>
<div style="position:absolute; left:120px; top:220px">
   <button style="width:80px"
           onclick="handleOK()">OK</button>
     
   <button style="width:80px"
           onclick="handleCancel()">Cancel</button>
     
   <button style="width:80px"
```

```
                 onclick="handleApply()">Apply</button>
      </div>
   </body>
</html>
```

JavaScript: jsb27-39.js

```
// Close the dialog
function closeme()
{
   window.close();
}

// Handle click of OK button
function handleOK()
{
   var returnFunc = window.dialogArguments;
   returnFunc(getFormData());
   closeme();
}

// Handle click of Apply button
function handleApply()
{
   var returnFunc = window.dialogArguments;
   returnFunc(getFormData());
}

// Handle click of Cancel button
function handleCancel()
{
   window.returnValue = "";
   closeme();
}

// Generic function converts form element name-value pairs
// into an array
function getFormData()
{
   var form = document.prefs;
   var returnedData = new Array();
   // Harvest values for each type of form element
   for (var i = 0; i < form.elements.length; i++)
   {
      if (form.elements[i].type == "text")
      {
         returnedData[form.elements[i].name] = form.elements[i].value;
      }
      else if (form.elements[i].type.indexOf("select") != -1)
      {
         returnedData[form.elements[i].name] =
            form.elements[i].options[form.elements[i].selectedIndex].value;
```

continued

LISTING 27-39 *(continued)*

```
      }
      else if (form.elements[i].type == "radio")
      {
         returnedData[form.elements[i].name] = form.elements[i].value;
      }
      else if (form.elements[i].type == "checkbox")
      {
         returnedData[form.elements[i].name] = form.elements[i].value;
      }
      else
      {
         continue;
      }
   }
   return returnedData;
}

// Initialize by setting form elements from passed data
function init(currPrefs)
{
   if (currPrefs)
   {
      var form = document.prefs;
      if (currPrefs["name"])
      {
         form.name.value = currPrefs["name"];
      }
      if (currPrefs["bgColor"])
      {
         setSelected(form.bgColor, currPrefs["bgColor"]);
      }
      if (currPrefs["textColor"])
      {
         setSelected(form.textColor, currPrefs["textColor"]);
      }
      if (currPrefs["h1Size"])
      {
         setSelected(form.h1Size, currPrefs["h1Size"]);
      }
   }
}

// Utility function to set a SELECT element to one value
function setSelected(select, value)
{
   for (var i = 0; i < select.options.length; i++)
   {
      if (select.options[i].value == value)
      {
         select.selectedIndex = i;
```

```
         break;
      }
   }
   return;
}

// Utility function to accept a press of the
// Enter key in the text field as a click of OK
function checkEnter()
{
   if (window.event.keyCode == 13)
   {
      handleOK();
   }
}
```

The biggest design challenge you probably face with respect to these windows is deciding between a modal and modeless dialog-box style. Some designers insist that modality has no place in a graphical user interface; others say that there are times when you need to focus the user on a very specific task before any further processing can take place. That's where a modal dialog box makes perfect sense.

Related Item: `window.open()` method

sizeToContent()

Returns: Nothing

Compatibility: WinIE-, MacIE-, NN6+, Moz+, Safari-, Opera-, Chrome-

The NN6+/Moz `window.sizeToContent()` method can be a valuable aid in making sure that a window (especially a subwindow) is sized for the optimum display of the window's content. But you must be cautious with this method, or it will do more harm than good.

Invoking the `sizeToContent()` method resizes the window so that all content is visible. Concerns about variations in OS-specific rendering become a thing of the past. Naturally, you should perform this action only on a window whose content occupies, at the most, a space smaller than the smallest video monitor running your code (typically 640×480 pixels, but conceivably much smaller for future versions of the browser used on handheld computers).

You can get the user in trouble, however, if you invoke the method twice on the same window that contains the resizing script. This action can cause the window to expand to a size that may exceed the pixel size of the user's video monitor. Successive invocations fail to cinch up the window's size to fit its content again. Multiple invocations are safe, however, on subwindows when the resizing script statement is in the main window.

Example

Use The Evaluator (Chapter 4, "JavaScript Essentials") in NN6+/Moz to try the `sizeToContent()` method. Assuming that you are running The Evaluator from the `Chap13` directory on the CD-ROM (or the directory copied as is to your hard disk), you can open a subwindow with one of the other files in the directory and then size the subwindow. Enter the following statement in the top text box:

```
a = window.open("jsb13-02.html","")
```

You might, depending on where you installed the CD-ROM, have to alter the path in the above statement to something like this:

```
a = window.open("../../Chap13/lst13-02.htm","")
```

The file will open, depending on the browser version and your browser configuration settings, in either a new tab or a new subwindow. Enter the following statement in the top text box:

```
a.sizeToContent()
```

The resized window (if a new tab was opened) or subwindow is at the minimum recommended width for a browser window and at a height tall enough to display the little bit of content in the document.

As with any method that changes the size of the browser window, problems arise in browsers such as Firefox 2 that support tabbed browsing where multiple pages are opened as different tabs within the same browser instance. In a tabbed browser, it is impossible to resize the window of one page without altering all others. It's certainly still possible to open multiple browser instances to resolve this problem, but the default response to `window.open()` is to open a new tab, not a new browser window.

Related Item: `window.resizeTo()` method

stop()

Returns: Nothing

Compatibility: WinIE-, MacIE-, NN4+, Moz+, Safari+, Opera+, Chrome-

The NN/Moz original `stop()` method offers a scripted equivalent of clicking the Stop button in the toolbar. The availability of this method allows you to create your own toolbar on your page and hide the toolbar (in the main window with signed scripts, or in a subwindow). For example, if you have an image representing the Stop button in your page, you can surround it with a link whose action stops loading, as in the following:

```
<a href="javascript: void stop()"><img src="myStop.gif" border="0"></a>
```

A script cannot stop its own document from loading, but it can stop the loading of another frame or window. Similarly, if the current document dynamically loads a new image or a multimedia MIME type file as a separate action, the `stop()` method can halt that process. Even though the `stop()` method is a window method, it is not tied to any specific window or frame: Stop means stop.

Related Items: `window.back()`, `window.find()`, `window.forward()`, `window.home()`, `window.print()` methods

Event handlers

onafterprint
onbeforeprint

Compatibility: WinIE5+, MacIE-, NN-, Moz-, Safari-, Opera-, Chrome-

Each of these event handlers fires after the user has clicked the OK button in IE's Print dialog box. This goes for printing that is invoked manually (via menus and browser shortcut buttons) and the `window.print()` method.

Although printing is usually WYSIWYG, it is conceivable that you may want the printed version of a document to display more, or less, of the document than is showing at that instant. For example, you may have a special copyright notice that you want appended to the end of a page whenever it goes to the printer. In that case, the element with that content can have its `display` style sheet property set to `none` when the page loads. Before the document is sent to the printer, a script needs to adjust that style property to display the element as a block item; after printing, have your script revert the setting to `none`.

Immediately after the user clicks the OK button in the Print dialog box, the `onbeforeprint` event handler fires. As soon as the page is sent to the printer or spooler, the `onafterprint` event handler fires.

Example

The following script fragment assumes that the page includes a `div` element whose style sheet includes a setting of `display:none` as the page loads. Somewhere in the `head`, the print-related event handlers are set as properties:

```
function showPrintCopyright()
{
   document.all.printCopyright.style.display = "block";
}
function hidePrintCopyright()
{
   document.all.printCopyright.style.display = "none";
}
window.onbeforeprint = showPrintCopyright;
window.onafterprint = hidePrintCopyright;
```

onbeforeunload

Compatibility: WinIE4+, MacIE5+, NN-, Moz+, Safari+, Opera-, Chrome+

Any user or scripted action that normally forces the current page to be unloaded or replaced causes the `onbeforeunload` event handler to fire. Unlike the `onunload` event handler, however, `onbeforeunload` is a bit better behaved when it comes to allowing complex scripts to finish before the actual unloading takes place. Moreover, you can assign a string value to the `event.returnValue` property in the event handler function. That string becomes part of a message in an alert window that gives the user a chance to stay on the page. If the user agrees to stay, the page does not unload, and any action that caused the potential replacement is canceled.

Example

The simple page in Listing 27-40 shows you how to give the user a chance to stay on the page.

LISTING 27-40

Using the onbeforeunload Event Handler

```
<html>
   <head>
      <title>onbeforeunload Event Handler</title>
```

continued

LISTING 27-40 *(continued)*

```
      <script type="text/javascript">
         function verifyClose()
         {
            return "We really like you and hope you will stay longer.";
         }
      </script>
   </head>
   <body onbeforeunload="return verifyClose()">
      <h1>onbeforeunload Event Handler</h1>
      <hr />
      <p>Use this button to navigate to the previous page:
         <button id="go"
                 onclick="history.back()">Go Back</button>
      </p>
   </body>
</html>
```

Related Item: `onunload` event handler

onerror

Compatibility: WinIE4+, MacIE4+, NN3+, Moz+, Safari-, Opera-, Chrome-

(See the discussion of the `window.onerror` property earlier in this chapter.)

onhelp

Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-, Opera-, Chrome-

The generic `onhelp` event handler, discussed in Chapter 26, also fires when the user activates the context-sensitive help within a modal or modeless dialog box. In the latter case, a user can click the Help icon in the dialog box's title bar, at which time the cursor changes to a question mark. The user can then click any element in the window. At that second click, the `onhelp` event handler fires, and the `event` object contains information about the element clicked (the `event.srcElement` is a reference to the specific element), allowing a script to supply help about that element.

To prevent the browser's built-in help window from appearing, the event handler must evaluate to `return false` (IE4+) or set the `event.returnValue` property to `false` (IE5+).

Example

The following script fragment can be used to provide context-sensitive help within a dialog box. Help messages for only two form elements are shown here, but in a real application, you could easily add more messages.

```
function showHelp()
{
   switch (event.srcElement.name)
```

```
      {
         case "bgColor" :
            alert("Choose a color for the main window\'s background.");
            break;
         case "name" :
            alert("Enter your first name for a friendly greeting.");
            break;
         default :
            alert("Make preference settings for the main page styles.");
      }
      event.returnValue = false;
   }
   window.onhelp = showHelp;
```

Because this page's help focuses on form elements, the `switch` construction cases are based on the `name` properties of the form elements. For other kinds of pages, the `id` properties may be more appropriate.

Related Items: `event` object (Chapter 32, "Event Objects"); `switch` construction (Chapter 21, "Control Structures and Exception Handling")

onload

Compatibility: WinIE3+, MacIE3+, NN2+, Moz+, Safari+, Opera+, Chrome+

The `onload` event handler fires in the current window at the end of the document loading process (after all text and image elements have been transferred from the source file server to the browser, and after all plug-ins and Java applets have loaded and started running). At that point, the browser's memory contains all the objects and script components in the document that the browser can possibly know about.

The `onload` handler is an attribute of a `<body>` tag for a single-frame document, or of the `<frameset>` tag for the top window of a multiple-frame document. When the handler is an attribute of a `<frameset>` tag, the event triggers only after all frames defined by that frameset have completely loaded.

Use either of the following scenarios to insert an `onload` handler into a document the old way:

```
<html>
   <head>
   </head>
   <body [other attributes] onload="statementOrFunction">
      [body content]
   </body>
</html>

<html>
   <head>
   </head>
   <frameset [other attributes] onload="statementOrFunction">
      <frame [frame specification attributes] />
   </frameset>
</html>
```

Alternatively, you can bind an event listener to the `window` object using the W3C `addEventListener()` method. Chapter 32 explains the modern cross-browser event-handling technique.

This handler has a special capability when it's part of a frameset definition: The handler won't fire until the `onload` event handlers of all child frames in the frameset have fired. Therefore, if some initialization scripts depend on components existing in other frames, trigger them from the frameset's `onload` event handler. This brings up a good general rule of thumb for writing JavaScript: Scripts that execute during a document's loading should contribute to the process of generating the document and its objects. To act immediately on those objects, design additional functions that are called by the `onload` event handler for that window.

The type of operations suited for an `onload` event handler are those that can run quickly and without user intervention. Users shouldn't be penalized by having to wait for considerable postloading activity to finish, before they can interact with your pages. At no time should you present a modal dialog box as part of an `onload` handler. Users who design macros on their machines to visit sites unattended may get hung up on a page that automatically displays an alert, confirm, or prompt dialog box. On the other hand, an operation such as setting the `window.defaultStatus` property is a perfect candidate for an `onload` event handler, as is initializing event handlers as properties of element objects in the page.

Note

Browsers equipped with pop-up window blockers ignore all `window.open()` method calls in `onload` event handler functions. ■

Related Items: `onunload` event handler; `window.defaultStatus` property

`onresize`

Compatibility: WinIE4+, MacIE4+, NN4+, Moz+, Safari+, Opera+, Chrome+

If a user resizes a window, the action causes the `onresize` event handler to fire for the `window` object. When you assign a function to the event (for example, `window.onresize = handleResizes`), the NN/Moz event object conveys `width` and `height` properties that reveal the outer width and height of the entire window. A window resize should not reload the document such that an `onload` event handler fires (although some early Navigator versions did fire the extra event).

Related Item: `event` object (Chapter 32, "Event Objects")

`onscroll`

Compatibility: WinIE4+, MacIE4+, NN7+, Moz+, Safari 1.3+, Opera+, Chrome+

The `onscroll` event handler fires for the `body` element object as the result of manual scrolling of the document (via scroll bars or navigation keyboard keys) and scripted scrolling via the `doScroll()` method, via the `scrollIntoView()` method, or by adjusting the `scrollTop` and/or `scrollLeft` properties of the `body` element object. For manual scrolling and scrolling by `doScroll()`, the event seems to fire twice in succession. Moreover, the `event.srcElement` property is `null` even when the `body` element is handling the `onscroll` event handler.

Example

Listing 27-41 is a highly artificial demonstration of what can be a useful tool for some page designs. Consider a document that occupies a window or frame but that you don't want scrolled, even by accident with a mouse wheel. (Be aware that the `onscroll` event is not fired if the user scrolls the page

with a mouse wheel in Safari and Chrome in some versions of Windows, but is fired on the Mac.) If scrolling of the content would destroy the appearance or value of the content, you want to make sure that the page always zips back to the top. The `onscroll` event handler in Listing 27-41 does just that. Notice that the event handler is set as a property of the `window` object.

LISTING 27-41

Preventing a Page from Scrolling

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>onscroll Event Handler</title>
      <script type="text/javascript" src="../jsb-global.js"></script>
      <script type="text/javascript">
         addEvent(window, "scroll", zipBack);
         //window.onscroll = zipBack;

         function zipBack()
         {
            window.scroll(0,0);
         }
      </script>
   </head>
   <body>
      <h1>onscroll Event Handler</h1>
      <hr />
      <div>This page always zips back to the top if you try to scroll it.</div>
      <p>
         <iframe frameborder="0" scrolling="no" height="1000"
                 src="bofright.html"></iframe>
      </p>
   </body>
</html>
```

Related Items: `scrollBy()`, `scrollByLines()`, `scrollByPages()` methods

onunload

Compatibility: WinIE3+, MacIE3+, NN2+, Moz+, Safari+, Opera-, Chrome+

An `unload` event reaches the current window just before a document is cleared from view. The most common ways windows are cleared are when new HTML documents are loaded into them, when the user clicks the `back` button or the `reload` button, or when the user does the reload keyboard shortcut, when a script begins writing new HTML on the fly for the window or frame. Be aware that in Safari the unload event is not fired when closing the tab/browser in some versions of Windows.

Limit your use of the `onunload` event handler to quick operations that do not inhibit the transition from one document to another. Do not invoke any methods that display dialog boxes. You specify `onunload` event handlers in the same places in an HTML document as the `onload` handlers:

as a `<body>` tag attribute for a single-frame window or as a `<frameset>` tag attribute for a multiframe window. Both `onload` and `onunload` event handlers can appear in the same `<body>` or `<frameset>` tag without causing problems. The `onunload` event handler merely stays safely tucked away in the browser's memory, waiting for the `unload` event to arrive for processing as the document gets ready to clear the window.

One caution about the `onunload` event handler: Even though the event fires before the document goes away, don't burden the event handler with time-consuming tasks, such as generating new objects or submitting a form. The document will probably go away before the function completes, leaving the function looking for objects and values that no longer exist. The best defense is to keep your `onunload` event handler processing to a minimum.

Note

Browsers equipped with pop-up window blockers ignore all `window.open()` method calls in `onunload` event handler functions. ■

Related Item: `onload` event handler

frame Element Object

For HTML element properties, methods, and event handlers, see Chapter 26, "Generic HTML Element Objects."

Properties	Methods	Event Handlers
allowTransparency		
borderColor		
contentDocument		
contentWindow		
frameBorder		
height		
longDesc		
marginHeight		
marginWidth		
name		
noResize		
scrolling		
src		
width		

Syntax

Accessing properties or methods of a `frame` element object from a `frameset`:

```
(IE4+)     document.all.frameID. property | method([parameters])
(IE5+/W3C) document.getElementById("frameID"). property |
           method([parameters])
```

Accessing properties or methods of a `frame` element from a frame document:

```
(IE4+)     parent.document.all.frameID. property | method([parameters])
(IE5+/W3C) parent.document.getElementById("frameID"). property |
           method([parameters])
```

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

About this object

As noted in the opening section of this chapter, a `frame` element object is distinct from the frame object that acts as a `window` object in a document hierarchy. The `frame` element object is available to scripts only when all HTML elements are exposed in the object model.

Because the `frame` element object is an HTML element, it shares the properties, methods, and event handlers of all HTML elements, as described in Chapter 26. By and large, you access the `frame` element object to set or modify an attribute value in the `<frame>` tag. If so, you simplify matters if you assign an identifier to the `id` attribute of the tag. Your tag still needs a `name` attribute if your scripts refer to frames through the original object model (a `parent.frameName` reference). Although there is no law against using the same identifier for both `name` and `id` attributes, using different names to prevent potential conflict with references in browsers that recognize both attributes is best.

To modify the dimensions of a frame, you must go to the `frameset` element object that defines the `cols` and `rows` attributes for the frameset. These properties can be modified on the fly in modern browsers.

Properties

allowTransparency

Value: Boolean Read/Write

Compatibility: WinIE6+, MacIE-, NN-, Moz-, Safari-, Opera-, Chrome-

The `allowTransparency` property indicates whether the frame's background is transparent. This property applies primarily to the `iframe` object, because framesets don't have background colors or images to show through a transparent frame.

borderColor

Value: Hexadecimal triplet or color name string Read/Write

Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-, Opera-, Chrome-

If a frame displays a border (as determined by the `frameborder` attribute of the `frame` element, or `border` attribute of the `frameset` element), it can have a color set separately from the rest of the frames. The initial color (if different from the rest of the frameset) is usually set by the `bordercolor` attribute of the `<frame>` tag. After that, scripts can modify settings as needed.

Modifying a single frame's border can be risky at times, depending on your color combinations. In practice, different browsers appear to follow different rules when it comes to negotiating conflicts or defining just how far a single frame's border extends into the border space. Color changes to individual frame borders do not always render. Verify your designs on as many browsers and operating system variations as you can to test your combinations.

Example

Although you may experience problems changing the color of a single frame border, the W3C DOM syntax would look like the following if the script were inside the framesetting document:

```
document.getElementById("contentsFrame").borderColor = "red";
```

The IE-only version would be

```
document.all["contentsFrame"].borderColor = "red";
```

These examples assume that the frame name arrives to a script function as a string. If the script is executing in one of the frames of the frameset, add a reference to `parent` in the preceding statements.

Related Items: `frame.frameBorder`, `frameset.frameBorder` properties

contentDocument

Value: `document` object reference — Read-Only

Compatibility: WinIE8+, MacIE-, NN6+, Moz+, Safari+, Opera+, Chrome+

The `contentDocument` property of a `frame` element object is nothing more than a reference to the document contained by that frame. This property bridges the gap between the `frame` element object and the `frame` object. Both of these objects contain the same `document` object, but from a scripting point of view, references most typically use the frame object to reach the document inside a frame, whereas the `frame` element is used to access properties equated with the `frame` tag's attributes. But if your script finds that it has a reference to the `frame` element object, you can use the `contentDocument` property to get a valid reference to the document and, therefore, any other content of the frame.

Example

A framesetting document script might be using the ID of a `frame` element to read or adjust one of the element properties, and then need to perform some action on the content of the page through its `document` object. You can get the reference to the `document` object via a statement such as the following:

```
var doc = document.getElementById("Frame3").contentDocument;
```

Then your script can, for example, dive into a form in the document:

```
var val = doc.mainForm.entry.value;
```

Related Items: contentWindow property; document object

contentWindow

Value: document object reference Read-Only

Compatibility: WinIE5.5+, MacIE-, NN7+, Moz1.0.1+, Safari+, Opera+, Chrome+

The contentWindow property of a frame element object is simply a reference to the window generated by that frame. This property provides access to the frame's window, which can then be used to reach the document inside the frame.

Example

You can get the reference to the window object associated with a frame via a statement such as the following:

```
var win = document.getElementById("Frame3").contentWindow;
```

Related Item: window object

frameBorder

Value: yes | no | 1 | 0 as strings Read/Write

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

The frameBorder property offers scripted access to a frame element object's frameborder attribute setting. Except for IE, none of the browsers we tested responded well to modifying this property after the page has loaded.

Values for the frameBorder property are strings that substitute for Boolean values. Value yes or 1 means that the border is (supposed to be) turned on; no or 0 (supposedly) turns off the border. There is no consistent default behavior among the browsers.

Example

The default value for the frameBorder property is yes. You can use this setting to create a toggle script (which, unfortunately, does not change the border's appearance in IE). The W3C-compatible version looks like the following:

```
function toggleFrameScroll(frameID)
{
   var theFrame = document.getElementById(frameID);
   if (theFrame.frameBorder == "yes")
   {
      theFrame.frameBorder = "no";
   }
   else
   {
      theFrame.frameBorder = "yes";
   }
}
```

Related Items: frameset.frameBorder properties

height
width

Value: Integer Read-Only

Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari+, Opera-, Chrome+

These two properties enable you to retrieve the height and width of a `frame` element object. These values are not necessarily the same as `document.body.clientHeight` and `document.body.clientWidth`, because the frame dimensions include chrome associated with the frame, such as scroll bars. These values are read-only. If you need to modify the dimensions of a frame, do so via the `frameset` element object's `rows` and/or `cols` properties. Reading integer values for a frame's height and width properties is much easier than trying to parse the `rows` and `cols` string properties.

Example

The following fragment assumes a frameset defined with two frames set up as two columns within the frameset. The statements here live in the framesetting document. They retrieve the current width of the left frame and increase the width of that frame by 10 percent. Syntax shown here is for the W3C DOM, but can easily be adapted to IE-only terminology.

```
var frameWidth = document.getElementById("leftFrame").width;
document.getElementById("mainFrameset").cols =
   (Math.round(frameWidth * 1.1)) + ",*";
```

Notice that the numeric value of the existing frame width is first increased by 10 percent and then concatenated to the rest of the string property assigned to the frameset's `cols` property. The asterisk after the comma means that the browser should figure out the remaining width and assign it to the right frame.

Related Item: `frameset` object

longDesc

Value: URL string Read/Write

Compatibility: WinIE6+, MacIE5+, NN6+, Moz+, Safari+, Opera+, Chrome+

The `longDesc` property is the scripted equivalent of the `longdesc` attribute of the `<frame>` tag. This HTML 4 attribute is intended to provide browsers a URL to a document that contains a long description of the element. Future browsers can use this feature to provide information about the frame for visually impaired site visitors.

marginHeight
marginWidth

Value: Integer Read/Write

Compatibility: WinIE6+, MacIE5+, NN6+, Moz+, Safari+, Opera+, Chrome+

Browsers tend to insert content within a frame automatically by adding a margin between the content and the edge of the frame. These values are represented by the `marginHeight` (top and bottom edges) and `marginWidth` (left and right edges) properties. Although the properties are not read-only, changing the values after the frameset has loaded does not alter the appearance of the

document in the frame. If you need to alter the margin(s) of a document inside a frame, adjust the `document.body.style` margin properties.

Also be aware that although the default values of these properties are empty (meaning when no `marginheight` or `marginwidth` attributes are set for the `<frame>` tag), margins are built into the page. The precise pixel count of those margins varies with operating system.

Related Item: `style` object (Chapter 38, "Style Sheet and Style Objects")

name

Value: String Read/Write

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

The `name` property is the identifier associated with the frame for use as a reference. Scripts can reference the frame through the `name` property (for example, `top.frames["myFrame"]`), which is typically assigned via the `name` attribute.

noResize

Value: Boolean Read/Write

Compatibility: WinIE6+, MacIE5+, NN6+, Moz+, Safari+, Opera+, Chrome+

Web designers commonly fix their framesets so that users cannot resize the frames (by dragging any divider border between frames). The `noResize` property lets you read and adjust that behavior of a frame after the page has loaded. For example, during some part of the interaction with a user on a page, you may allow the user to modify the frame size manually while in a certain mode. Or you may grant the user one chance to resize the frame. When the `onresize` event handler fires, a script sets the `noResize` property of the `frame` element to `false.` If you turn off resizing for a frame, all edges of the frame become nonresizable, regardless of the `noResize` value setting of adjacent frames. Turning off resizability has no effect on the ability of scripts to alter the sizes of frames via the `frameset` element object's `cols` or `rows` properties.

Example

The following statement turns off the ability for a frame to be resized:

```
parent.document.getElementById("myFrame1").noResize = true;
```

Because of the negative nature of the property name, it may be difficult to keep the logic straight (setting `noResize` to `true` means that resizability is turned off). Keep a watchful eye on your Boolean values.

Related Items: `frameset.cols`, `frameset.rows` properties

scrolling

Value: `yes | no | 1 | 0` as strings Read/Write

Compatibility: WinIE6+, MacIE5+, NN6+, Moz+, Safari+, Opera+, Chrome+

The `scrolling` property lets scripts turn scroll bars on and off inside a single frame of a frameset. By default, scrolling is turned on unless overridden by the `scroll` attribute of the `<frame>` tag.

Values for the `scrolling` property are strings that substitute for Boolean values. Value `yes` or `1` means that scroll bars are visible (provided that there is more content than can be viewed without

scrolling); `no` or `0` hides scroll bars in the frame. IE also recognizes (and sets as default) the `auto` value.

Note

Although this property is read/write, changing its value by script does not alter a frame's appearance in WinIE, Mozilla browsers, or Safari. ■

Example

Listing 27-42 produces a frameset consisting of eight frames. The content for the frames is generated by a script within the frameset (via the `fillFrame()` function). Event handlers (`onclick`) in the `body` of each frame invoke the `toggleFrameScroll()` function. Both ways of referencing the `frame` element object are shown, with the IE-only version commented out.

In the `toggleFrameScroll()` function, the `if` condition checks whether the property is set to something other than `no`. This allows the condition to evaluate to `true` if the property is set to either `auto` (the first time) or `yes` (as set by the function). Note that the scroll bars don't disappear from the frames in IE, NN6+, Safari, Opera, or Chrome.

LISTING 27-42

Controlling the frame.scrolling Property

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>frame.scrolling Property</title>
      <script type="text/javascript">
         function toggleFrameScroll(frameID)
         {
            // IE5+/W3C version
            var theFrame = document.getElementById(frameID);

            if (theFrame.scrolling != "no")
            {
               theFrame.scrolling = "no";
            }
            else
            {
               theFrame.scrolling = "yes";
            }
         }

         // generate content for each frame
         function fillFrame(frameID)
         {
            var page = "<html><body onclick='parent.toggleFrameScroll(\""
                     +  frameID
```

```
                 + "\")'><span style='font-size:24pt'>";
         page += "<p>This frame has the ID of:<\/p><p>"
              + frameID
              + ".<\/p>";
         page += "<\/span><\/body><\/html>";
         return page;
      }
   </script>
</head>
<frameset id="outerFrameset" cols="50%,50%">
   <frameset id="innerFrameset1" rows="25%,25%,25%,25%">
      <frame id="myFrame1" src="javascript:parent.fillFrame('myFrame1')" />
      <frame id="myFrame2" src="javascript:parent.fillFrame('myFrame2')" />
      <frame id="myFrame3" src="javascript:parent.fillFrame('myFrame3')" />
      <frame id="myFrame4" src="javascript:parent.fillFrame('myFrame4')" />
   </frameset>
   <frameset id="innerFrameset2" rows="25%,25%,25%,25%">
      <frame id="myFrame5" src="javascript:parent.fillFrame('myFrame5')" />
      <frame id="myFrame6" src="javascript:parent.fillFrame('myFrame6')" />
      <frame id="myFrame7" src="javascript:parent.fillFrame('myFrame7')" />
      <frame id="myFrame8" src="javascript:parent.fillFrame('myFrame8')" />
   </frameset>
</frameset>
</html>
```

src

Value: URL string Read/Write

Compatibility: WinIE6+, MacIE5+, NN6+, Moz+, Safari+, Opera+, Chrome+

The `src` property of a `frame` element object offers an additional way of navigating to a different page within a frame (meaning other than assigning a new URL to the `location.href` property of the frame object). For backward compatibility with older browsers, however, continue using `location.href` for scripted navigation. Remember that the `src` property belongs to the `frame` element object, not the `window` object it represents. Therefore, references to the `src` property must be via the element's ID and/or node hierarchy.

Example

For best results, use fully formed URLs as values for the `src` property, as shown here:

```
parent.document.getElementById("mainFrame").src = "http://www.dannyg.com";
```

Relative URLs and `javascript:` pseudo-URLs will also work most of the time.

Related Item: `location.href` property

frameset Element Object

For HTML element properties, methods, and event handlers, see Chapter 26, "Generic HTML Element Objects."

Properties	Methods	Event Handlers
border		
borderColor		
cols		
frameBorder		
frameSpacing		
rows		

Syntax

Accessing properties or methods of a `frameset` element object from a `frameset`:

```
(IE4+)     document.all.framesetID. property | method([parameters])
(IE5+/W3C) document.getElementById("framesetID"). property |
           method([parameters])
```

Accessing properties or methods of a `frameset` element from a frame document:

```
(IE4+)     parent.document.all.framesetID. property | method([parameters])
(IE5+/W3C) parent.document.getElementById("framesetID"). property |
           method([parameters])
```

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

About this object

The `frameset` element object is the script-accessible equivalent of the element generated via the `<frameset>` tag. This element is different from the parent (window-type) object of the original object model. A `frameset` element object has properties and methods that impact the HTML element; by contrast, the `window` object referenced from documents inside frames via the `parent` or `top` window references, contains a document and all the content that goes along with it.

When framesets are nested in side one another, a node parent–child relationship exists between containing and contained framesets. For example, consider the following skeletal nested frameset structure:

```
<frameset id="outerFrameset" cols="30%, 70%">
   <frame id="frame1">
   <frameset id="innerFrameset" rows="50%,50%">
```

```
        <frame id="frame2">
        <frame id="frame3">
    </frameset>
</frameset>
```

When writing scripts for documents that go inside any of the frames of this structure, references to the framesetting window and frames are a flatter hierarchy than the HTML signifies. A script in any frame references the framesetting window via the `parent` reference; a script in any frame references another frame via the `parent.frameName` reference. In other words, the `window` objects of the frameset defined in a document are all siblings, and have the same parent.

Such is not the case when viewing the preceding structure from the perspective of W3C node terminology. Parent–child relationships are governed by the nesting of HTML elements, irrespective of whatever windows get generated by the browser. Therefore, `frame2` has only one sibling: `frame3`. Both of those share one parent: `innerFrameset`. Both `innerFrameset` and `frame1` are children of `outerFrameset`. If your script were sitting on a reference to `frame2`, and you wanted to change the `cols` property of `outerFrameset`, you would have to traverse two generations of nodes:

```
frame2Ref.parentNode.parentNode.cols = "40%,60%";
```

What might confuse matters ever more in practice is that a script belonging to one of the frames must use window object terminology to jump out of the current `window` object to the frameset that generated the frame window for the document. In other words, there is no immediate way to jump directly from a document to the `frame` element object that defines the frame in which the document resides. The document's script accesses the node hierarchy of its frameset via the `parent.document` reference. But this reference is to the `document` object that contains the entire frameset structure. Fortunately, the W3C DOM provides the `getElementById()` method to extract a reference to any node nested within the document. Thus, a document inside one of the frames can access the `frame` element object just as though it were any element in a typical document (which it is):

```
parent.document.getElementById("frame2")
```

No reference to the containing `frameset` element object is necessary. To make that column width change from a script inside one of the frame windows, the statement would be

```
parent.document.getElementById("outerFrame").cols = "40%,60%";
```

The inner frameset is equally accessible by the same syntax.

Properties

`border`

Value: Integer Read/Write

Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-, Opera-, Chrome-

The `border` property of a `frameset` element object lets you read the thickness (in pixels) of the borders between frames of a frameset. If you do not specify a `border` attribute in the frameset's tag, the property is empty, rather than reflecting the actual border thickness applied by default.

Example

Even though the property is read/write, changing the value does not change the thickness of the border you see in the browser. If you need to find the thickness of the border, a script reference from one of the frame's documents would look like the following:

```
var thickness = parent.document.all.outerFrameset.border;
```

Related Item: `frameset.frameBorder` property

borderColor

Value: Hexadecimal triplet or color name string   Read/Write

Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-, Opera-, Chrome-

The `borderColor` property lets you read the value of the color assigned to the `bordercolor` attribute of the frameset's tag. Although the property is read/write, changing the color by script does not alter the border colors rendered in the browser window. Attribute values set as color names are returned as hexadecimal triplets when you read the property value.

Example

To retrieve the current color setting in a frameset, a script reference from one of the frame's documents would look like the following:

```
var borderColor = parent.document.all.outerFrameset.borderColor;
```

Related Items: `frame.borderColor`, `frameset.frameBorder` properties

cols
rows

Value: String   Read/Write

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

The `cols` and `rows` properties of a `frameset` element object let you read and modify the sizes of frames after the frameset has loaded. These two properties are defined in the W3C DOM. Values for both properties are strings, which may include percent symbols or asterisks. Therefore, if you are trying to increase or decrease the size of a frame column or row gradually, you must parse the string for the necessary original values before performing any math on them (or, in IE4+, use the `frame` element object's `height` and `width` properties to gauge the current frame size in pixels).

Adjusting these two properties lets you modify the frameset completely, including adding or removing columns or rows in the frameset grid. Because a change in the frameset structure could impact scripts by changing the size of the frames array associated with the parent window or unloading documents that contain needed data, be sure to test your scripts with both states of your frameset. If you want to remove a frame from a frameset view, you might be safer to specify the size of zero for that particular row or column in the frameset. Of course, a size of zero still leaves a 1-pixel frame, but it is essentially invisible if borders are not turned on and the 1-pixel frame has the same background color as the other frames. Another positive byproduct of this technique is that you can restore the other frame with its document state identical from when it was hidden.

When you have nested framesets defined in a single document, be sure to reference the desired `frameset` element object. One object may be specifying the columns, and another (nested) one

specifies the rows for the grid. Assign a unique ID to each `frameset` element so that references can be reliably directed to the proper object.

Example

Listings 27-43, 27-44, and 27-45 show the HTML for a frameset and two of the three documents that go into the frameset. The final document is an HTML version of the U.S. Bill of Rights, which is serving here as a content frame for the demonstration.

The frameset listing (see Listing 27-43) shows a three-frame setup. Down the left column is a table of contents (see Listing 27-44). The right column is divided into two rows. In the top row is a simple control (see Listing 27-45) that hides and shows the table-of-contents frame. As the user clicks the hot spot of the control (located inside a `span` element), the `onclick` event handler invokes the `toggleTOC()` function in the frameset.

LISTING 27-43

Frameset and Script for Hiding/Showing a Frame

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Hide/Show Frame Example</title>
      <script type="text/javascript">
         var origCols;
         function toggleTOC()
         {
            if (origCols)
            {
               showTOC();
            }
            else
            {
               hideTOC();
            }
         }
         function hideTOC()
         {
            var frameset = document.getElementById("outerFrameset");
            origCols = frameset.cols;
            frameset.cols = "0,*";
         }
         function showTOC()
         {
            if (origCols)
            {
               document.getElementById("outerFrameset").cols = origCols;
               origCols = null;
            }
         }
      </script>
```

continued

LISTING 27-43 *(continued)*

```
  </head>
  <frameset id="outerFrameset" frameborder="no" cols="150,*">
     <frame id="TOC" name="TOCFrame" src="jsb27-44.html" />
     <frameset id="innerFrameset1" rows="80,*">
        <frame id="controls" name="controlsFrame" src="jsb27-45.html" />
        <frame id="content" name="contentFrame" src="bofright.html" />
     </frameset>
  </frameset>
</html>
```

When a user clicks the hot spot to hide the frame, the script copies the original `cols` property settings to a global variable. The variable is used in `showTOC()` to restore the frameset to its original proportions. This allows a designer to modify the HTML for the frameset without also having to dig into scripts to hard-wire the restored size.

LISTING 27-44

Table of Contents Frame Content

```
<!DOCTYPE html>
<html>
  <head>
     <meta http-equiv="content-type" content="text/html;charset=utf-8">
     <title>Table of Contents</title>
     <style type="text/css">
        body
        {
         background-color:#EEEEEE;
        }
     </style>
  </head>
  <body>
     <h3>
        Table of Contents
     </h3>
     <hr />
     <ul style="font-size:10pt">
        <li><a href="bofright.htm#article1"
               target="contentFrame">Article I</a></li>
        <li><a href="bofright.htm#article2"
               target="contentFrame">Article II</a></li>
        <li><a href="bofright.htm#article3"
               target="contentFrame">Article III</a></li>
        <li><a href="bofright.htm#article4"
               target="contentFrame">Article IV</a></li>
        <li><a href="bofright.htm#article5"
               target="contentFrame">Article V</a></li>
        <li><a href="bofright.htm#article6"
```

```
                target="contentFrame">Article VI</a></li>
         <li><a href="bofright.htm#article7"
                target="contentFrame">Article VII</a></li>
         <li><a href="bofright.htm#article8"
                target="contentFrame">Article VIII</a></li>
         <li><a href="bofright.htm#article9"
                target="contentFrame">Article IX</a></li>
         <li><a href="bofright.htm#article10"
                target="contentFrame">Article X</a></li>
      </ul>
   </body>
</html>
```

LISTING 27-45

Control Panel Frame

```
<html>
   <head>
      <title>Control Panel</title>
      <style type="text/css">
         span
         {
            text-decoration:underline; cursor:pointer;
         }
      </style>
   </head>
   <body>
      <p><span id="tocToggle"
               onclick="parent.toggleTOC()">
            &lt;&lt;Hide/Show&gt;&gt;</span> Table of Contents
      </p>
   </body>
</html>
```

Related Item: `frame` object

frameBorder

Value: `yes` | `no` | `1` | `0` as strings — Read/Write

Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-, Opera-, Chrome-

The `frameBorder` property offers scripted access to a `frameset` element object's `frameborder` attribute setting. IE4+ does not respond well to modifying this property after the page has loaded.

Values for the `frameBorder` property are strings that substitute for Boolean values. Value `yes` or `1` means that the border is (supposed to be) turned on; `no` or `0` turns off the border.

Example

The default value for the `frameBorder` property is `yes`. You can use this setting to create a toggle script (which, unfortunately, does not change the appearance in IE). The IE5+ version looks like the following:

```
function toggleFrameScroll(framesetID)
{
   var theFrameset = document.getElementById(framesetID);
   if (theFrameset.frameBorder == "yes")
   {
      theFrameset.frameBorder = "no";
   }
   else
   {
      theFrameset.frameBorder = "yes";
   }
}
```

Related Item: `frame.frameBorder` property

frameSpacing

Value: Integer Read/Write

Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-, Opera-, Chrome-

The `frameSpacing` property of a `frameset` element object lets you read the spacing (in pixels) between frames of a frameset. If you do not specify a `framespacing` attribute in the frameset's tag, the property is empty, rather than reflecting the actual border thickness applied by default (usually 2).

Example

If you need to change the spacing, a script reference would look like the following:

```
document.getElementById("outerFrameset").frameSpacing = "10";
```

Related Item: `frameset.border` property

iframe Element Object

For HTML element properties, methods, and event handlers, see Chapter 26, "Generic HTML Element Objects."

Properties	Methods	Event Handlers
align		
allowTransparency		
contentDocument		
contentWindow		

Properties	Methods	Event Handlers
frameBorder		
frameSpacing		
height		
hspace		
longDesc		
marginHeight		
marginWidth		
name		
noResize		
scrolling		
src		
vspace		
width		

Syntax

Accessing properties or methods of an `iframe` element object from a containing document:

```
(IE4+)     document.all.iframeID. property | method([parameters])
(IE4+/NN6) window.frames["iframeName"]. property | method([parameters])
(IE5+/W3C) document.getElementById("iframeID"). property |
           method([parameters])
```

Accessing properties of methods of an `iframe` element from a document inside the `iframe` element:

```
(IE4+)     parent.document.all.iframeID. property | method([parameters])
(IE5+/W3C) parent.document.getElementById("iframeID"). property |
           method([parameters])
```

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

About this object

An `iframe` element allows HTML content from a separate source to be loaded within the body of another document. In some respects, the NN4 `layer` element was a precursor to the `iframe` concept, but unlike the `layer`, an `iframe` element is not inherently positionable. It is positionable the same way as any other HTML element: by assigning positioning attributes to a style sheet associated with the `iframe`. Without explicit positioning, an `iframe` element appears in the body of a

document in the normal source-code order of elements. Unlike a frame of a frameset, an `iframe` can be placed arbitrarily in the middle of any document. If the `iframe` changes size under script control, the surrounding content moves out of the way or cinches up.

What truly sets the `iframe` apart from other HTML elements is its ability to load and display external HTML files and, with the help of scripts, have different pages loaded into the `iframe` without disturbing the rest of the content of the main document. Pages loaded into the `iframe` can also have scripts and any other features that you may like to put into an HTML document (including XML, in IE for Windows).

The `iframe` element has a rich set of attributes that lets the HTML author control the look, size (`height` and `width`), and, to some degree, behavior of the frame. Most of these attributes are accessible to scripts as properties of an `iframe` element object.

It is important to bear in mind that an `iframe` element is in many respects like a `frame` element, especially when it comes to window kinds of relationships. If you plant an `iframe` element in a document of the main window, that element shows up in the main window's object model as a frame, accessible via common frames terminology:

```
window.frames[i]
window.frames[frameName]
```

Within that `iframe` frame object is a document and all its contents. All references to the document objects inside the `iframe` must flow through the portal of the `iframe` frame.

Conversely, scripts in the document living inside an `iframe` can communicate with the main document via the `parent` reference. Of course, you cannot replace the content of the main window with another HTML document (using `location.href`, for instance) without destroying the `iframe` that was in the original document.

Properties

`align`

Value: String Read/Write

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

The `align` property governs how an `iframe` element aligns itself with respect to surrounding content on the page. Two of the possible values (`left` and `right`) position the `iframe` along the left and right edge (respectively) of the `iframe`'s containing element (usually the `body`). Just as with an image, when an `iframe` is floated along the left and right edges of a container, other content wraps around the element. Table 27-6 shows all possible values and their meanings.

As your script changes the value of the `align` property, the page automatically reflows the content to suit the new alignment.

Example

The default setting for an `iframe` alignment is `baseline.` A script can shift the `iframe` to be flush with the right edge of the containing element as follows:

```
document.getElementById("iframe1").align = "right";
```

TABLE 27-6

Values of the align Property

Value	Description
`absbottom`	Aligns the bottom of the `iframe` with the imaginary line that extends along character descenders of surrounding text
`absmiddle`	Aligns the middle of the `iframe` with the center point between the surrounding text's `top` and `absbottom`
`baseline`	Aligns the bottom of the `iframe` with the baseline of surrounding text
`bottom`	Aligns the bottom of the `iframe` with the bottom of the surrounding text
`left`	Aligns the `iframe` flush with left edge of the containing element
`middle`	Aligns the imaginary vertical center line of surrounding text with the same for the `iframe` element
`right`	Aligns the `iframe` flush with the right edge of the containing element
`texttop`	Aligns the top of the `iframe` element with the imaginary line that extends along the tallest ascender of surrounding text
`top`	Aligns the top of the `iframe` element with the surrounding element's `top`

Related Items: `iframe.hspace`, `iframe.vspace` properties

allowTransparency

Value: Boolean Read/Write

Compatibility: WinIE6+, MacIE-, NN-, Moz-, Safari-, Opera-, Chrome-

The `allowTransparency` property indicates whether the frame's background is transparent. By setting this property to `true`, you allow a background color or image to show through the transparent frame.

contentDocument

Value: `document` object reference Read-Only

Compatibility: WinIE8+, MacIE-, NN6+, Moz+, Safari+, Opera+, Chrome+

The `contentDocument` property of an `iframe` element object is nothing more than a reference to the document contained by that frame. If your script finds that it has a reference to an `iframe` element object, you can use the `contentDocument` property to get a valid reference to the document and, therefore, any other content of the frame.

Example

A document script might be using the ID of an `iframe` element to read or adjust one of the element properties; then it needs to perform some action on the content of the page through its

document object. You can get the reference to the document object via a statement such as the following:

```
var doc = document.getElementById("Frame3").contentDocument;
```

Then your script can, for example, dive into a form in the document:

```
var val = doc.mainForm.entry.value;
```

Related Items: contentWindow property; document object

contentWindow

Value: document object reference Read-Only

Compatibility: WinIE5.5+, MacIE-, NN7+, Moz1.0.1+, Safari+, Opera+, Chrome+

The contentWindow property of an iframe element object serves as a reference to the window object generated by the frame. You can then use this window object as a means of accessing the document object and any document elements.

Related Items: contentDocument property; window object

frameBorder

(See frame.frameBorder() and frameset.frameBorder())

frameSpacing

(See frameset.frameSpacing())

height
width

Value: Integer Read/Write

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

The height and width properties provide access to the height and width of the iframe object, and allow you to alter the size of the frame. Both properties are specified in pixels.

hspace
vspace

Value: Integer Read/Write

Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-, Opera-, Chrome-

These IE-specific properties allow for margins to be set around an iframe element. In general, hspace and vspace properties (and their HTML attributes) have been replaced by CSS margins and padding. These properties and their attributes are not recognized by any W3C standard (including HTML 4).

Values for these properties are integers representing the number of pixels of padding between the element and surrounding content. The hspace value assigns the same number of pixels to the left and

right sides of the element; the `vspace` value is applied to both the top and bottom edges. Scripted changes to these values have no effect in WinIE5+.

Related Item: `style.padding` property

longDesc

Value: URL string Read/Write

Compatibility: WinIE6+, MacIE5+, NN6+, Moz+, Safari+, Opera+, Chrome+

The `longDesc` property is the scripted equivalent of the `longdesc` attribute of the `<iframe>` tag. This HTML 4 attribute is intended to provide browsers a URL to a document that contains a long description of the element. Future browsers can use this feature to provide information about the frame for visually impaired site visitors.

marginHeight
marginWidth

Value: Integer Read/Write

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

Browsers tend to insert content within a frame automatically by adding a margin between the content and the edge of the frame. These values are represented by the `marginHeight` (top and bottom edges) and `marginWidth` (left and right edges) properties. Although the properties are not read-only, changing the values after the frameset has loaded does not alter the appearance of the document in the frame. If you need to alter the margin(s) of a document inside a frame, adjust the `document.body.style` margin properties.

Also be aware that although the default values of these properties are empty (that is, when no `marginheight` or `marginwidth` attributes are set for the `<iframe>` tag), margins are built into the page. The precise pixel count of those margins varies with different operating systems.

Related Item: `style` object (Chapter 38, "Style Sheet and Style Objects")

name

Value: String Read/Write

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

The `name` property is the identifier associated with the frame for use as a reference. Scripts can reference the frame through the `name` property (for example, `window.frames["myIframe"]`), which is typically assigned via the `name` attribute.

noResize

(See `frame.noResize()`)

scrolling

Value: `yes | no | 1 | 0` as strings Read/Write

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

The `scrolling` property lets scripts turn scroll bars on and off inside an `iframe` element. By default, scrolling is turned on unless overridden by the `scroll` attribute of the `<iframe>` tag.

Values for the `scrolling` property are strings that substitute for Boolean values. Value `yes` or `1` means that scroll bars are visible (provided that there is more content than can be viewed without scrolling); `no` or `0` hides scroll bars in the frame. IE4+ also recognizes (and sets as default) the `auto` value.

Example

The following `toggleIFrameScroll()` function accepts a string of the `iframe` element's ID as a parameter, and switches between on and off scroll bars in the `iframe`. The `if` condition checks whether the property is set to something other than `no`. This test allows the condition to evaluate to `true` if the property is set to either `auto` (the first time) or `yes` (as set by the function).

```
function toggleFrameScroll(frameID)
{
   // IE5 & NN6 version
   var theFrame = document.getElementById(frameID);

   if (theFrame.scrolling != "no")
   {
      theFrame.scrolling = "no";
   }
   else
   {
      theFrame.scrolling = "yes";
   }
}
```

Related Item: `frame.scrolling` property

src

Value: URL string Read/Write

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

The `src` property of an `iframe` element object offers an additional way of navigating to a different page within an inline frame (that is, other than assigning a new URL to the `location.href` property of the frame object). Remember that the `src` property belongs to the `iframe` element object, not the `window` object it represents. Therefore, references to the `src` property must be via the element's ID and/or node hierarchy.

Example

For best results, use fully formed URLs as values for the `src` property, as shown here:

```
document.getElementById("myIframe").src = "http://www.dannyg.com";
```

Relative URLs and `javascript:` pseudo-URLs also work most of the time.

Related Item: `location.href` property

popup Object

Properties	Methods	Event Handlers
document	hide()	
isOpen	show()	

Syntax

Creating a popup object:

```
var popupObj = window.createPopup()
```

Accessing properties or methods of a popup object from a document in the window that created the pop-up:

```
popupObj.property | method([parameters])
```

Compatibility: WinIE5.5+, MacIE-, NN-, Moz-, Safari-, Opera-, Chrome-

About this object

A popup object is a chromeless window space that overlaps the window whose document generates the pop-up. A pop-up also appears in front of any dialog boxes. Unlike the dialog-box windows generated via showModalDialog() and showModelessDialog() methods, your scripts must not only create the window, but also put content in it, and then define where on the screen, and how big, it will be. Because the pop-up window IE creates is a window and not a dialog box, there is no exact W3C equivalent. However, with the ability to add and delete elements to the node tree using the W3C DOM, you can achieve the same type of functionality. See Chapter 25, "Document Object Model Essentials," for more information.

Because the pop-up window has no chrome (title bar, resize handles, and so on), you should populate its content with a border and/or background color so that it stands out from the main window's content. The following statements reflect a typical sequence of creating, populating, and showing a popup object:

```
var popup = window.createPopup();
var popupBody = popup.document.body;
popupBody.style.border = "solid 2px black";
popupBody.style.padding = "5px";
popupBody.innerHTML = "<p>Here is some text in a popup window</p>";
popup.show(200,100, 200, 50, document.body);
```

Note

As handy as it may be, in a strict W3C approach to JavaScript, you wouldn't use the innerHTML property, because it isn't officially part of the W3C standard. However, it is often too powerful a convenience property to ignore, as much of the code throughout this book is a testament. The book does show the W3C node manipulation alternative to innerHTML in some examples. Refer to Chapter 29 for a thorough explanation and examples of the W3C alternative to innerHTML. ■

The pop-up window that IE creates is, in fact, a window, but only from the point of view of the document that it contains. In other words, although the number of properties and methods for the `popup` object is small, the `parentWindow` property of the document inside the pop-up points to a genuine `window` property. Even so, be aware that this pop-up does not appear as a distinct window among those listed in the Windows Taskbar. If a user clicks outside the pop-up or switches to another application, the pop-up disappears, and you must reinvoke the `show()` method by script (complete with dimension and position parameters) to force the pop-up to reappear.

When you assign content to a pop-up, you are also responsible for making sure that the content fits the size of the pop-up you specify. If the content runs past the rectangular space (body text word wraps within the pop-up's rectangle), no scroll bars appear.

Properties

document

Value: `document` object reference Read-Only

Compatibility: WinIE5.5+, MacIE-, NN-, Moz-, Safari-, Opera-, Chrome-

Use the `document` property as a gateway to the content of a pop-up window. This property is the only access point available from the script that creates the pop-up, to the pop-up itself. The most common application of this property is to set document properties governing the content of the pop-up window. For example, to give the pop-up a border (because the pop-up itself has no window chrome), the script that creates the window can assign values to the `style` property of the document in the pop-up window, as follows:

```
myPopup.document.body.style.border = "solid 3px gray";
```

Be aware that the `document` object of a pop-up window may not implement the full flexibility you're used to with primary window `document` objects. For example, you are not allowed to assign a URL to the `document.URL` property in a pop-up window.

Example

Use The Evaluator (Chapter 4, "JavaScript") to experiment with the `popup` object and its properties. Enter the following statements in the top text box. The first statement creates a pop-up window whose reference is assigned to the `a` global variable. Next, a reference to the body of the pop-up's document is preserved in the `b` variable for the sake of convenience. Further statements work with these two variables.

```
a = window.createPopup()
b = a.document.body
b.style.border = "solid 2px black"
b.style.padding = "5px"
b.innerHTML = "<p>Here is some text in a popup window</p>"
a.show(200,100, 200, 50, document.body)
```

See the description of the `show()` method for details on the parameters.

Related Item: `document` object

isOpen

Value: Boolean Read-Only

Compatibility: WinIE5.5+, MacIE-, NN-, Moz-, Safari-, Opera-, Chrome-

While a pop-up window is visible, its `isOpen` property returns `true`; otherwise, the property returns `false`. Because any user action in the browser causes the pop-up to hide itself, the property is useful only for script statements that are running on their own after the pop-up is made visible.

Example

Use The Evaluator (Chapter 4, "JavaScript Essentials") to experiment with the `isOpen` property. Enter the following statements in the top text box. The sequence begins with a creation of a simple pop-up window, whose reference is assigned to the `a` global variable. Note that the final statement is actually two statements, designed so that the second statement executes while the pop-up window is still open.

```
a = window.createPopup();
a.document.body.innerHTML = "<p>Here is a popup window</p>";
a.show(200,100, 200, 50, document.body); alert("Popup is open:" + a.isOpen);
```

If you then click in the main window to hide the pop-up, you will see a different result if you enter the following statement in the top text box by itself:

```
alert("Popup is open:" + a.isOpen);
```

Related Item: `popup.show()` method

Methods

hide()
show(left, top, width, height[, positioningElementRef])

Returns: Nothing

Compatibility: WinIE5.5+, MacIE-, NN-, Moz-, Safari-, Opera-, Chrome-

After you have created a `popup` object with the `window.createPopup()` method and populated it with content, you must explicitly show the window via the `show()` method. If the window is hidden because a user clicked the main browser window somewhere, the `show()` method (and all its parameters) must be invoked again. To have a script hide the window, invoke the `hide()` method for the `popup` object.

The first four parameters of the `show()` method are required; they define the pixel location and size of the pop-up window. By default, the coordinate space for the `left` and `top` parameters is the video display. Thus, a `left` and `top` setting of zero places the pop-up in the top-left corner of the video screen. But you can define a different coordinate space by adding an optional fifth parameter. This parameter must be a reference to an element on the page. To confine the coordinate space to the content region of the browser window, specify the `document.body` object as the positioning element reference.

***popupObject*.show()**

Example

Listing 27-46 demonstrates the `show()` and `hide()` methods for a `popup` object. A click of the button on the page invokes the `selfTimer()` function, which acts as the main routine for this page. The goal is to produce a pop-up window that self-destructs 5 seconds after it appears. Along the way, a message in the pop-up counts down the seconds.

A reference to the pop-up window is preserved as a global variable called `popup`. After the `popup` object is created, the `initContent()` function stuffs the content into the pop-up by way of assigning `style` properties, and some `innerHTML`, for the body of the document that is automatically created when the pop-up is generated. A `span` element is defined so that another function later on can modify the content of just that segment of text in the pop-up. Notice that the assignment of content to the pop-up is predicated on the pop-up window's having been initialized (by virtue of the `popup` variable's having a value assigned to it) and that the pop-up window is not showing. Although invoking `initContent()` under any other circumstances is probably impossible, the validation of the desired conditions is good programming practice.

Back in `selfTimer()`, the `popup` object is displayed. Defining the desired size requires some trial and error to make sure that the pop-up window comfortably accommodates the text that is put into the pop-up in the `initContent()` function.

With the pop-up window showing, now is the time to invoke the `countDown()` function. Before the function performs any action, it validates that the pop-up has been initialized and is still visible. If a user clicks the main window while the counter is counting down, this changes the value of the `isOpen` property to `false`, and nothing inside the `if` condition executes.

This `countDown()` function grabs the inner text of the `span` and uses `paresInt()` to extract just the integer number (using base 10 numbering, because we're dealing with zero-leading numbers that can potentially be regarded as octal values). The condition of the `if` construction decreases the retrieved integer by one. If the decremented value is zero, the time is up, and the pop-up window is hidden with the `popup` global variable returned to its original, `null` value. But if the value is other than zero, the inner text of the `span` is set to the decremented value (with a leading zero), and the `setTimeout()` method is called upon to reinvoke the `countDown()` function in 1 second (1,000 milliseconds).

LISTING 27-46

Hiding and Showing a Pop-Up

HTML: jsb27-46.html

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>popup Object</title>
      <script type="text/javascript" src="jsb27-46.js"></script>
   </head>
   <body>
      <form>
```

```
        <input type="button" value="Impossible Mission"
               onclick="selfTimer()" />
      </form>
   </body>
</html>
```

JavaScript: jsb27-46.js

```
var popup;
function initContent()
{
   if (popup && !popup.isOpen)
   {
      var popBody = popup.document.body;
      popBody.style.border = "solid 3px red";
      popBody.style.padding = "10px";
      popBody.style.fontSize = "24pt";
      popBody.style.textAlign = "center";
      var bodyText = "<P>This popup will self-destruct in ";
      bodyText += "<span id='counter'>05<\/span>";
      bodyText += " seconds...<\/P>";
      popBody.innerHTML = bodyText;
   }
}
function countDown()
{
   if (popup && popup.isOpen)
   {
      var currCount = parseInt(popup.document.all.counter.innerText, 10);
      if (--currCount == 0)
      {
         popup.hide();
         popup = null;
      }
      else
      {
         popup.document.all.counter.innerText = "0" + currCount;
         setTimeout("countDown()", 1000);
      }
   }
}
function selfTimer()
{
   popup = window.createPopup();
   initContent();
   popup.show(200,200,400,100,document.body);
   setTimeout("countDown()", 1000);
}
```

***popupObject*.show()**

The `hide()` method here is invoked by a script that is running while the pop-up window is showing. Because a pop-up window automatically goes away if a user clicks the main window, it is highly unlikely that the `hide()` method would ever be invoked by itself in response to user action in the main window. If you want a script in the pop-up window to close the pop-up, use `parentWindow.close()`.

Related Items: `popup.isOpen` property; `window.createPopup()` method

CHAPTER 28

Location and History Objects

Not all objects in the document object model (DOM) are things you can see in the content area of the browser window. Each browser window or frame maintains a bunch of other information about the page you are currently visiting and where you have been. The URL of the page you see in the window is called the *location,* and browsers store this information in the `location` object. As you surf the Web, the browser stores the URLs of your past pages in the `history` object. You can manually view what that object contains by looking in the browser menu for the item that enables you to jump back to a previously visited page. This chapter is all about these two nearly invisible, but important, objects.

These objects are not only valuable to your browser, but also valuable to snoopers who might want to write scripts to see what URLs you're viewing in another frame, or the URLs of other sites you've visited in the past dozen mouse clicks. As a result, security restrictions built into browsers limit access to some of these objects' properties (unless you use signed scripts in NN4+/Moz). For older browsers, these properties simply are not available from a script.

IN THIS CHAPTER

Loading new pages and other media types via the location object

Security restrictions across frames

Navigating through the browser history under script control

location Object

Properties	Methods	Event Handlers
`hash`	`assign()`	None
`host`	`reload()`	
`hostname`	`replace()`	
`href`		
`pathname`		

Properties	Methods	Event Handlers
port		
protocol		
search		

Syntax

Loading a new document into the current window:

```
[window.]location.href = "URL";
```

Accessing location object properties or methods:

```
[window.]location.property | method([parameters])
```

About this object

In its place one level below window-style objects in the original document object hierarchy, the location object represents information about the URL of any currently open window, or of a specific frame. To display the URL of the current web page, you can reference the location object like this:

```
document.write(location.href);
```

In this example, the href property evaluates to the URL, which is written to the current page in its entirety. The location object also allows you to access individual parts of the URL, as you will see in a moment.

When you reference the location object in the framesetting document of a multiple-frame window, the location is given as the parent window's URL, which appears in the Location (or Address) field of the browser. Each frame also has a location associated with it, although you may not see any overt reference to the frame's URL in the browser. To get URL information about a document located in another frame, the reference to the location object must include the window frame reference. For example, if you have a window consisting of two frames, Table 28-1 shows the possible references to the location objects for all frames comprising the web presentation.

Note

Scripts cannot alter the URL displayed in the browser's Location/Address box. For security and privacy reasons, that text box cannot display anything other than the URL of a current page or URL in transit. ■

Most properties of a location object deal with network-oriented information. This information involves various data about the physical location of the document on the network, including the host server, the protocol being used, and other components of the URL. Given a complete URL for a typical World Wide Web page, the window.location object assigns property names to various segments of the URL, as shown here:

```
http://www.example.com:80/promos/newproducts.html#giantGizmo
```

Property	Value
`protocol`	`"http:"`
`hostname`	`"www.example.com"`
`port`	`"80"`
`host`	`"www.example.com:80"`
`pathname`	`"/promos/newproducts.html"`
`hash`	`"#giantGizmo"`
`href`	`"http://www.example.com:80/promos newproducts.html#giantGizmo"`

TABLE 28-1

Location Object References in a Two-Frame Browser Window

Reference	Description
`location` (or `window.location`)	URL of frame displaying the document that runs the script statement containing this reference
`parent.location`	URL information for parent window that defines the `<frameset>`
`parent.frames[0].location`	URL information for first visible frame
`parent.frames[1].location`	URL information for second visible frame
`parent.otherFrameName.location`	URL information for another named frame in the same frameset

The `window.location` object is handy when a script needs to extract information about the URL, perhaps to obtain a base reference on which to build URLs for other documents to be fetched as the result of user action. This object can eliminate a nuisance for web authors who develop sites on one machine and then upload them to a server (perhaps at an Internet service provider) with an entirely different directory structure. By building scripts to construct base references from the directory location of the current document, you can construct the complete URLs for loading documents. You don't have to change the base reference data manually in your documents as you shift the files from computer to computer or from directory to directory. To extract the segment of the URL and place it in the enclosing directory, use the following:

```
var baseRef = location.href.substring(0,location.href.lastIndexOf("/") + 1);
```

Caution

Security alert: To allay fears of Internet security breaches and privacy invasions, scriptable browsers prevent your script in one frame from retrieving `location` object properties from other frames whose domain and server are not your own (unless you use signed scripts in NN4+/Moz or the user has set the IE browser to trust your site). This restriction puts a damper on many scripters' well-meaning designs and aids for web watchers and visitors. If you attempt such property accesses, however, you receive an "access denied" (or similar) security warning dialog box. ■

Setting the value of some `location` properties is the preferred way to control which document gets loaded into a window or frame. Though you may expect to find a method somewhere in JavaScript that contains a plain-language *Go* or *Open* word (to simulate what you see in the browser menu bar), you point your browser to another URL by setting the `window.location.href` property to that URL, as in:

```
window.location.href = "http://www.dannyg.com/";
```

The equals assignment operator (=) in this kind of statement is a powerful weapon. In fact, setting the `location.href` object to a URL of a different MIME type, such as one of the variety of sound and video formats, causes the browser to load those files into the plug-in or helper application designated in your browser's settings. The `location.assign()` method was originally intended for internal use by the browser, but it is available for scripters (although I don't recommend using it for navigation). Internet Explorer's object model includes a `window.navigate()` method that also loads a document into a window, but you can't use it for cross-browser applications.

Two other methods complement the `location` object's capability to control navigation. One method is the script equivalent of clicking Reload; the other method enables you to replace the current document's entry in the history with that of the next URL of your script's choice.

Properties

hash

Value: String Read/Write

Compatibility: WinIE3+, MacIE3+, NN2+, Moz+, Safari+, Opera+, Chrome+

The *hash mark* (#) is a URL convention that directs the browser to an anchor located in the document. Any name you assign to an anchor (with the `<a name="..."> ...</a>` tag pair) becomes part of the URL after the hash mark. A `location` object's `hash` property is the name of the anchor part of the current URL (which consists of the hash mark and the name).

If you have written HTML documents with anchors and directed links to navigate to those anchors, you have probably noticed that although the destination location shows the anchor as part of the URL (for example, in the Location field), the window's anchor value does not change as the user manually scrolls to positions in the document where other anchors are defined. An anchor appears in the URL only when the window has navigated there as part of a link or in response to a script that adjusts the URL.

Just as you can navigate to any URL by setting the `window.location.href` property, you can navigate to another hash in the same document by adjusting only the `hash` property of the location without the hash mark (as shown in the following example).

Listing 28-1 demonstrates how to use the `hash` property to access the anchor part of a URL. When you load the script in Listing 28-1, adjust the height of the browser window so that only one section is visible at a time. When you click a button, the script navigates to the next logical section in the progression and eventually takes you back to the top. The page won't scroll any farther than the bottom

of the document. Therefore, an anchor near the bottom of the page may not appear at the top of the browser window.

LISTING 28-1

A Document with Anchors

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>location.hash Property</title>
      <script type="text/javascript">
      function goNextAnchor(where)
      {
         window.location.hash = where;
      }
      </script>
   </head>
   <body>
      <h1><a id="start" name="start">Top</a></h1>
      <form>
         <input type="button" name="next" value="NEXT"
         onclick="goNextAnchor('sec1')" />
      </form>
      <hr /><br /><br /><br /><br /><br />
      <h1><a id="sec1" name="sec1">Section 1</a></h1>
      <form>
         <input type="button" name="next" value="NEXT"
         onclick="goNextAnchor('sec2')" />
      </form>
      <hr /><br /><br /><br /><br /><br />
      <h1><a id="sec2" name="sec2">Section 2</a></h1>
      <form>
         <input type="button" name="next" value="NEXT"
         onclick="goNextAnchor('sec3')" />
      </form>
      <hr /><br /><br /><br /><br /><br />
      <h1><a id="sec3" name="sec3">Section 3</a></h1>
      <form>
         <input type="button" name="next" value="BACK TO TOP"
         onclick="goNextAnchor('start')" />
      </form>
   </body>
</html>
```

Note

The property assignment event handling technique used in the previous example and throughout the chapter is a deliberate simplification to make the code more readable. It is generally better to use the more modern approach of binding events using the `addEventListener()` (NN6+/Moz/W3C) or `attachEvent()` (IE5+) methods. A modern cross-browser event handling technique is explained in detail in Chapter 32, "Event Objects." Several other chapters use the modern technique extensively. ■

Anchor names are passed as parameters with each button's `onclick` event handler. Instead of going through the work of assembling a `window.location` value in the function by appending a literal hash mark and the value for the anchor, here we simply modify the `hash` property of the current window's location. This is the preferred, cleaner method.

If you attempt to read back the `window.location.hash` property in an added line of script, however, the window's actual URL probably will not have been updated yet, and the browser will appear to be giving your script false information. To prevent this problem in subsequent statements of the same function, construct the URLs of those statements from the same variable values you use to set the `window.location.hash` property; don't rely on the browser to give you the values you expect.

Related Item: `location.href` property

host

Value: String Read/Write

Compatibility: WinIE3+, MacIE3+, NN2+, Moz+, Safari+, Opera+, Chrome+

The `location.host` property describes both the hostname and port of a URL. The port is included in the value only when the port is an explicit part of the URL. If you navigate to a URL that does not display the port number in the Location field of the browser, the `location.host` property returns the same value as the `location.hostname` property. As with the `location.hostname` property, the `location.host` property will likely come up blank for pages that you open from your local hard drive (localhost) in some browsers — for example, IE and WebKit browsers such as Chrome.

Use the `location.host` property to extract the `hostname:port` part of the URL of any document loaded in the browser. This capability may be helpful for building a URL to a specific document that you want your script to access on-the-fly.

Use the documents in Listing 28-2, Listing 28-3, and Listing 28-4 as tools to help you learn the values that the various `window.location` properties return. In the browser, open the file for Listing 28-2. This file creates a two-frame window. The left frame contains a temporary placeholder (see Listing 28-4) that displays some instructions. The right frame has a document (see Listing 28-3) that enables you to load URLs into the left frame and get readings on three different windows available: the parent window (which creates the multiframe window), the left frame, and the right frame.

LISTING 28-2

Frameset for the Property Picker

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
```

```
    <title>window.location Properties</title>
    </head>
    <frameset cols="50%,50%" border="1" bordercolor="black">
       <frame name="Frame1" src="jsb28-04.html" />
       <frame name="Frame2" src="jsb28-03.html" />
    </frameset>
</html>
```

LISTING 28-3

Property Picker

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Property Picker</title>
      <script type="text/javascript">
      var isNav = (typeof netscape != "undefined") ? true : false;

      function fillLeftFrame()
      {
         newURL = prompt("Enter the URL of a document to show in the left
            frame:","");
         if (newURL != null && newURL != "")
         {
            parent.frames[0].location = newURL;
         }
      }
      function showLocationData(form)
      {
         for (var i = 0; i <3; i++)
         {
            if (form.whichFrame[i].checked)
            {
               var windName = form.whichFrame[i].value;
               break;
            }
         }
         var theWind = "" + windName + ".location";
         if (isNav)
         {
            netscape.security.PrivilegeManager.enablePrivilege(
               "UniversalBrowserRead");
         }
         var theObj = eval(theWind);
         form.windName.value = windName;
         form.windHash.value = theObj.hash;
         form.windHost.value = theObj.host;
         form.windHostname.value = theObj.hostname;
```

continued

LISTING 28-3 *(continued)*

```
         form.windHref.value = theObj.href;
         form.windPath.value = theObj.pathname;
         form.windPort.value = theObj.port;
         form.windProtocol.value = theObj.protocol;
         form.windSearch.value = theObj.search;
         if (isNav)
         {
            netscape.security.PrivilegeManager.disablePrivilege(
               "UniversalBrowserRead");
         }
      }
      </script>
   </head>
   <body>
      Click the "Open URL" button to enter the location of an HTML document to
      display in the left frame of this window.
      <form>
         <input type="button" name="opener" value="Open URL..."
         onclick="fillLeftFrame()" />
         <hr />
         <center>
            Select a window/frame. Then click the "Show Location Properties"
            button to view each window.location property value for the desired
            window.
            <p><input type="radio" name="whichFrame" value="parent"
               checked="checked" />Parent window <input type="radio"
               name="whichFrame" value="parent.frames[0]" />Left frame <input
               type="radio" name="whichFrame" value="parent.frames[1]" />This
               frame</p>
            <p><input type="button" name="getProperties"
               value="Show Location Properties"
               onclick="showLocationData(this.form)" /> <input type="reset"
               value="Clear" /></p>
            <table border="2">
               <tr>
                  <td align="right">Window:</td>
                  <td><input type="text" name="windName" size="30" /></td>
               </tr>
               <tr>
                  <td align="right">hash:</td>
                  <td><input type="text" name="windHash" size="30" /></td>
               </tr>
               <tr>
                  <td align="right">host:</td>
                  <td><input type="text" name="windHost" size="30" /></td>
               </tr>
               <tr>
                  <td align="right">hostname:</td>
                  <td><input type="text" name="windHostname" size="30" /></td>
               </tr>
```

```
            <tr>
               <td align="right">href:</td>
               <td><textarea name="windHref" rows="3" cols="30" wrap="soft">
                  </textarea></td>
            </tr>
            <tr>
               <td align="right">pathname:</td>
               <td><textarea name="windPath" rows="3" cols="30" wrap="soft">
                  </textarea></td>
            </tr>
            <tr>
               <td align="right">port:</td>
               <td><input type="text" name="windPort" size="30" /></td>
            </tr>
            <tr>
               <td align="right">protocol:</td>
               <td><input type="text" name="windProtocol" size="30" /></td>
            </tr>
            <tr>
               <td align="right">search:</td>
               <td><textarea name="windSearch" rows="3" cols="30"
                  wrap="soft"></textarea></td>
            </tr>
         </table>
      </center>
   </form>
</body>
</html>
```

LISTING 28-4

Placeholder Document for Listing 28-2

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
   <title>Opening Placeholder</title>
   </head>
   <body>
      Initial placeholder. Experiment with other URLs for this frame (see
      right).
   </body>
</html>
```

For the best results, open a URL to a web document on the network from the same domain and server from which you load the listings (perhaps your local hard disk). If you open a URL to a web document that is not from the same domain and server, be sure to specify the fully qualified URL. If possible, load a document that includes anchor points to navigate through a long document. Click

the Left frame radio button and then click the button that shows all properties. This action fills the table in the right frame with all the available `location` properties for the selected window.

Related Items: `location.port`, `location.hostname` properties

hostname

Value: String Read/Write

Compatibility: WinIE3+, MacIE3+, NN2+, Moz+, Safari+, Opera+, Chrome+

The hostname of a typical URL is the name of the server on the network that stores the document you view in the browser. For most web sites, the server name includes not only the domain name, but also the `www.` prefix. The hostname does not, however, include the port number if the URL specifies such a number. Keep in mind that the `hostname` property will likely come up blank for pages that you open from your local hard drive (localhost) in some browsers — for example, IE and WebKit browsers such as Chrome.

See Listing 28-2, Listing 28-3, and Listing 28-4 for a set of related pages to help you view the hostname data for a variety of other pages.

Related Items: `location.host`, `location.port` properties

href

Value: String Read/Write

Compatibility: WinIE3+, MacIE3+, NN2+, Moz+, Safari+, Opera+, Chrome+

Of all the `location` object properties, `href` (hypertext reference) is probably the one most often called upon in scripting. The `location.href` property supplies a string of the entire URL of the specified `window` object.

Using this property on the left side of an assignment statement is the JavaScript way of opening a URL for display in a window. Any of the following statements can load my web site's index page into a single-frame browser window:

```
window.location = "http://www.dannyg.com";
window.location.href = "http://www.dannyg.com";
```

At times, you may encounter difficulty by omitting a reference to a window. JavaScript may get confused and reference the `document.location` property. To prevent this confusion, the `document.location` property was deprecated (put on the no-no list) and replaced by the `document.URL` property. In the meantime, you can't go wrong by always specifying a window in the reference.

Note

You should be able to omit the `href` property name when assigning a new URL to the `location` object (for example, `location = "http://www.dannyg.com"`). Although this works in most browsers most of the time, some early browsers behave more reliably if you assign a URL explicitly to the `location.href` property. If you want to play it safe, use `location.href` at all times. ■

Sometimes, you must extract the name of the current directory in a script so that another statement can append a known document to the URL before loading it into the window. Although the other `location` object properties yield an assortment of a URL's segments, none of them provides the

full URL to the current URL's directory. But you can use JavaScript string manipulation techniques to accomplish this task. Listing 28-5 shows such a possibility.

Depending on your browser, the values for the `location.href` property may be encoded with ASCII equivalents of nonalphanumeric characters. Such an ASCII value includes the `%` symbol and the ASCII numeric value. The most common encoded character in a URL is the space: `%20`. If you need to extract a URL and display that value as a string in your documents, you can safely pass all such potentially encoded strings through the JavaScript `unescape()` function. For example, if a URL is `http://www.example.com/product%20list`, you can convert it by passing it through the `unescape()` function, as in the following example.

```
var plainURL = unescape(window.location.href);
// result = "http://www.example.com/product list";
```

The inverse function, `escape()`, is available for sending encoded strings to server applications, such as CGI scripts. See Chapter 24, "Global Functions and Statements," for more details on these functions.

Listing 28-5 shows how the `href` property can be used to view the directory URL of the current page. This example includes the `unescape()` function in front of the part of the script that captures the URL. This function serves cosmetic purposes by displaying the pathname in alert dialog boxes for browsers that normally display the ASCII-encoded version.

Note

Although Listing 28-5 uses the `unescape()` global function for backward compatibility, that function (and its partner, `escape()`) have been removed from the ECMAScript standard as of version 3. These functions have been replaced by more modern versions, `decodeURI()` and `encodeURI()`. See Chapter 24 for details. ■

LISTING 28-5

Extracting the Directory of the Current Document

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Extract pathname</title>
      <script type="text/javascript">
      // general purpose function to extract URL of current directory
      function getDirPath(URL)
      {
         var result = unescape(URL.substring(0,(URL.lastIndexOf("/")) + 1));
         return result;
      }

      // handle button event, passing work onto general purpose function
      function showDirPath(URL)
      {
         alert(getDirPath(URL));
      }
      </script>
   </head>
```

continued

LISTING 28-5 *(continued)*

```
   <body>
      <form>
         <input type="button" value="View directory URL"
         onclick="showDirPath(window.location.href)" />
      </form>
   </body>
</html>
```

Related Items: `location.pathname`, `document.location` properties; `String` object (Chapter 15, "The String Object")

pathname

Value: String   Read/Write

Compatibility: WinIE3+, MacIE3+, NN2+, Moz+, Safari+, Opera+, Chrome+

The pathname component of a URL consists of the directory structure relative to the server's root volume. In other words, the root (the server name in an `http:` connection) is not part of the pathname. If the URL's path is to a file in the root directory, the `location.pathname` property is a single slash (`/`) character. Any other pathname starts with a slash character, indicating a directory nested within the root. The value of the `location.pathname` property also includes the document name.

See Listing 28-2, Listing 28-3, and Listing 28-4 earlier in this chapter for a multiple-frame example you can use to view the `location.pathname` property for a variety of URLs of your choice.

Related Item: `location.href` property

port

Value: String   Read/Write

Compatibility: WinIE3+, MacIE3+, NN2+, Moz+, Safari+, Opera+, Chrome+

These days, few consumer-friendly web sites need to include the port number as part of their URLs. You see port numbers mostly in the less-popular protocols, in URLs to sites used for private development purposes, or in URLs to sites that have no assigned domain names. You can retrieve the value with the `location.port` property. If you extract the value from one URL and intend to build another URL with that component, be sure to include the colon delimiter between the server's IP address and port number.

If you have access to URLs containing port numbers, use the documents in Listing 28-2, Listing 28-3, and Listing 28-4 to experiment with the output of the `location.port` property.

Related Item: `location.host` property

protocol

Value: String   Read/Write

Compatibility: WinIE3+, MacIE3+, NN2+, Moz+, Safari+, Opera+, Chrome+

The first component of any URL is the protocol used for the particular type of communication. For World Wide Web pages, the Hypertext Transfer Protocol (`http`) is the standard. Other common protocols you may see in your browser include HTTP-Secure (`https`), File Transfer Protocol (`ftp`), File (`file`), and Mail (`mailto`); web pages opened from your local hard drive use the `file` protocol. Values for the `location.protocol` property include not only the name of the protocol, but also the trailing colon delimiter. Thus, for a typical web-page URL, the `location.protocol` property is

```
http:
```

Notice that the usual slashes after the protocol in the URL are not part of the `location.protocol` value. Of all the `location` object properties, only the full URL (`location.href`) reveals the slash delimiters between the protocol and other components.

See Listing 28-2, Listing 28-3, and Listing 28-4 for a multiple-frame example you can use to view the `location.protocol` property for a variety of URLs. Notice that the protocol shows up initially as `file:` to indicate that the first page in the left frame is stored locally and accessed via the File protocol. Also try loading an FTP site to see the `location.protocol` value for that type of URL.

Related Item: `location.href` property

search

Value: String Read/Write

Compatibility: WinIE3+, MacIE3+, NN2+, Moz+, Safari+, Opera+, Chrome+

Perhaps you've noticed the long, cryptic URL that appears in the Location/Address field of your browser whenever you ask one of the World Wide Web search services to look up matches for items you enter in the keyword field. The URL starts the regular way — with protocol, host, and pathname values. But following the more traditional URL are search commands that are submitted to the search engine (typically, a CGI program running on the server). You can retrieve or set that trailing search query by using the `location.search` property.

Each search engine has its own formula for query submissions based on the designs of the HTML forms that obtain details from users. These search queries come in an encoded format that appears in anything but plain language. If you plan to script a search query, be sure you fully understand the search engine's format before you start assembling a string to assign to the `location.search` property of a window.

The most common format for search data is a series of name/value pairs. An equal symbol (=) separates a name and its value. Multiple name/value pairs have ampersands (&) between them. You should use the `escape()` function to convert the data to URL-friendly format, especially when the content includes spaces.

The `location.search` property also applies to any part of a URL after the filename, including parameters being sent to CGI programs on the server.

Passing data between pages via URLs

It is not uncommon to want to preserve some pieces of data that exist in one page so that a script in another page can pick up where the script processing left off in the first page. You can achieve persistence across page loads without any server programming through one of three techniques: the `document.cookie` (see Chapter 29, "Document and Body Objects"), variables in framesetting documents, and the search string of a URL. That's really what happens when you visit search and e-commerce sites that return information to your browser. Rather than store, say, your search criteria on the server, they

spit the criteria back to the browser as part of the URL. The next time you activate that URL, the values are sent to the server for processing (for example, to send you the next page of search results for a particular query).

Passing data between pages is not limited to client/server communication. You can use the search string strictly on the client side to pass data from one page to another. Unless some CGI process on the server is programmed to do something with the search string, a web server regurgitates the search string as part of the location data that comes back with a page. A script in the newly loaded page can inspect the search string (via the `location.search` property) and tear it apart to gather the data and put it into script variables. Take a look at Listing 28-6, Listing 28-7, and Listing 28-8 to see a powerful application of this technique.

As mentioned in Chapter 27, "Window and Frame Objects," you can force a particular HTML page to open inside the frameset for which it is designed. But with the help of the search string, you can reuse the same framesetting document to accommodate any number of content pages that go into one of the frames (rather than specifying a separate frameset for each possible combination of pages in the frameset). The listings in this section create a simple example of how to force a page to load in a frameset by passing some information about the page to the frameset. Thus, if a user has a URL to one of the content frames (perhaps it has been bookmarked by right-clicking the frame, or it comes up as a search-engine result), the page appears in its designated frameset the next time the user visits the page.

The fundamental task in this scheme has two parts. The first is in each of the content pages, where a script checks whether the page is loaded inside a frameset. If the frameset is missing, a search string is composed and appended to the URL for the framesetting document. The framesetting document has its own short script that looks for the presence of the search string. If the string is there, the script extracts the search string data and uses it to load that specific page into the content frame of the frameset.

Listing 28-6 is the framesetting document. The `getSearchAsArray()` function is more complete than necessary for this simple example, but you can use it in other instances to convert any number of name/value pairs passed in the search string (in traditional format of `name1=value1&name2=value2&etc.`) into an array whose indexes are the names (making it easier for scripts to extract a specific piece of passed data).

LISTING 28-6

A Smart Frameset

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Example Frameset</title>
      <script type="text/javascript">
      // Convert location.search into an array of values
      // indexed by name.
      function getSearchAsArray()
      {
         var results = new Array();
         var input = unescape(location.search.substr(1));
```

```
        if (input)
        {
           var srchArray = input.split("&");
           var tempArray = new Array();
           for (var i = 0; i < srchArray.length; i++)
           {
              tempArray = srchArray[i].split("=");
              results[tempArray[0]] = tempArray[1];
           }
        }
        return results;
     }

     function loadFrame()
     {
        if (location.search)
        {
           var srchArray = getSearchAsArray();
           if (srchArray["content"])
           {
              self.content.location.href = srchArray["content"];
           }
        }
     }
     </script>
   </head>
   <frameset cols="250,*" onload="loadFrame()">
      <frame name="toc" src="jsb28-07.html" />
      <frame name="content" src="jsb28-08.html" />
   </frameset>
</html>
```

Listing 28-7 is the HTML for the table-of-contents frame. Nothing elaborate goes on here, but you can see how normal navigation works for this simplified frameset. You can also see how this example could be easily built upon to provide a handy table-of-contents feature to a site with multiple sections or pages.

LISTING 28-7

The Table of Contents

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
   <title>Table of Contents</title>
   </head>
   <body bgcolor="#EEEEEE">
      <h3>Table of Contents</h3>
```

continued

LISTING 28-7 *(continued)*

```
      <hr />
      <ul>
         <li><a href="jsb28-08.html" target="content">Page 1</a></li>
         <li><a href="jsb28-08a.html" target="content">Page 2</a></li>
         <li><a href="jsb28-08b.html" target="content">Page 3</a></li>
      </ul>
   </body>
</html>
```

Listing 28-8 shows one of the content pages. As the page loads, the `checkFrameset()` function is invoked. If the window does not load inside a frameset, the script navigates to the framesetting page, passing the current content URL as a search string. Notice that the loading of this page on its own does not get recorded to the browser's history and isn't accessed if the user clicks the Back button.

LISTING 28-8

A Content Page

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Page 1</title>
      <script type="text/javascript">
      function checkFrameset()
      {
         if (parent == window)
         {
            // Use replace() to keep current page out of history
            location.replace("jsb28-06.html?content=" + escape(location.href));
         }
      }

      // Invoke the function
      checkFrameset();
      </script>
   </head>
   <body>
      <h1>Page 1</h1>
      <hr />
   </body>
</html>
```

In practice, we recommend placing the code for the `checkFrameset()` function and the call to it inside an external `.js` library, and linking that library to each content document of the frameset. That's why the function assigns the generic `location.href` property to the search string: You can use it on any content page.

The code in Listing 28-6, Listing 28-7, and Listing 28-8 establishes a frameset containing two frames. In the left frame is a table of contents that allows you to navigate among three different pages, the first of which is initially displayed in the right frame. The interesting thing about the example is how you can specify a new page in the `content` parameter of the search property; the page is then opened within the frameset. For example, the following URL would result in the page `hello.html` being opened in the right frame:

```
jsb28-06.html?content=hello.html
```

In this example URL, the frameset page is first opened due to the inclusion of the file `jsb28-06.html`, whereas the `hello.html` file is specified as the value of the `content` parameter.

Note

While this frameset example is an easy way to demonstrate the uses of the various location object properties, keep in mind that the use of framesets is no longer common programming practice. First, it does not support accessibility. Second, CSS Positioning is the standard for placing content on a webpage. ■

Related Item: `location.href` property

Methods

`assign("URL")`

Returns: Nothing

Compatibility: WinIE3+, MacIE3+, NN2+, Moz+, Safari+, Opera+, Chrome+

In earlier discussions about the `location` object, I said that you navigate to another page by assigning a new URL to the `location` object or `location.href` property. The `location.assign()` method does the same thing. In fact, when you set the `location` object to a URL, JavaScript silently applies the `assign()` method. No particular penalty or benefit comes from using the `assign()` method except perhaps making your code more understandable to others.

Related Item: `location.href` property

`reload(unconditionalGETBoolean)`

Returns: Nothing

Compatibility: WinIE4+, MacIE4+, NN3+, Moz+, Safari+, Opera+, Chrome+

The `location.reload()` method may be named inappropriately because it makes you think of the Reload/Refresh button in the browser toolbar. The `reload()` method is actually more powerful than the Reload/Refresh button in that it clears form control values that otherwise might survive the Reload/Refresh button. Note that IE for the Mac, Opera, and WebKit browsers such as Safari and Chrome do not preserve form control settings even with a soft reload.

Most form elements retain their screen states when you click Reload/Refresh. Text and `textarea` objects maintain whatever text is inside them; radio buttons and checkboxes maintain their checked status; `select` objects remember which item is selected. About the only items the Reload/Refresh button destroys are global variable values and any settable, but not visible, property (for example, the value of a `hidden input` object). I call this kind of reload a *soft reload.* A *hard reload,* on the other hand, should reset all data associated with a page, including default form selections.

locationObject.reload()

Browsers are frustratingly irregular about the ways they reload a document in the memory cache. In theory, an application of the `location.reload()` method should retrieve the page from the cache if the page is still available there (and the `history.go(0)` method should be even gentler, preserving form element settings). Adding a `true` parameter to the method is supposed to force an *unconditional GET* to the server, ignoring the cached version of the page. Yet when it is crucial for your application to get a page from the cache (for speed) or from the server (to guarantee a fresh copy), the browser behaves just the opposite of the way you want it to behave. Meta tags supposedly designed to prevent caching of a page rarely, if ever, work. Some scripters have had success in reloading the page from the server by setting `location.href` to the URL of the page, plus a slightly different search string (for example, based on a string representation of the `Date` object), so that there is no match for the URL in the cache.

The bottom line is that you should be prepared to try different schemes to achieve the effect you want. Also be prepared not to get the results you want. In other words, learn to live with the fact that you don't really have exacting control over retrieving a fresh page.

Listing 28-9 provides a means of testing the different outcomes of a soft reload versus a hard reload. Open this example page in a browser, and click a radio button. Then enter some new text, and make a choice in the `select` object. Clicking the Soft Reload/Refresh button invokes a method that reloads the document as though you had clicked the browser's Reload/Refresh button. It also preserves the visible properties of form elements. The Hard Reload button invokes the `location.reload()` method, which resets all objects to their default settings.

LISTING 28-9

Hard versus Soft Reloading

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Reload Comparisons</title>
      <script type="text/javascript">
      function hardReload()
      {
         location.reload(true);
      }
      function softReload()
      {
         history.go(0);
      }
      </script>
   </head>
   <body>
      <form name="myForm">
         <input type="radio" name="rad1" value="1" />Radio 1<br />
         <input type="radio" name="rad1" value="2" />Radio 2<br />
         <input type="radio" name="rad1" value="3" />Radio 3
         <p><input type="text" name="entry" value="Original" /></p>
         <p><select name="theList">
```

```
            <option>Red</option>
            <option>Green</option>
            <option>Blue</option>
         </select></p>
      <hr />
      <input type="button" value="Soft Reload" onclick="softReload()" />
      <input type="button" value="Hard Reload" onclick="hardReload()" />
   </form>
 </body>
</html>
```

Related Item: `history.go()` method

replace("*URL*")

Returns: Nothing

Compatibility: WinIE4+, MacIE4+, NN3+, Moz+, Safari+, Opera+, Chrome+

In a complex web site, you may have pages that you do not want to appear in the user's history list. For example, a registration sequence may lead the user to one or more intermediate HTML documents that won't make much sense to the user later. Or you may have a one-time introduction page that appears only the first time a user visits your site. You especially don't want users to see these pages again if they use the Back button to return to a previous URL. The `location.replace()` method navigates to another page, but it does not let the current page stay in the queue of pages accessible via the Back button.

Although you cannot prevent a document from appearing in the history list while the user views that page, you can instruct the browser to load another document into the window and replace the current history entry with the entry for the new document. This trick does not empty the history list but removes the current item from the list before the next URL is loaded. Removing the item from the history list prevents users from seeing the page again by clicking the Back button later.

Listing 28-10 shows how to use the `replace()` method to direct a web browser to a new URL. Calling the `location.replace()` method navigates to another URL, similarly to assigning a URL to the location. The difference is that the document doing the calling doesn't appear in the history list after the new document loads. You can verify this by trying to click the Back button to return to the page after clicking Replace Me in Listing 28-10; the button is dimmed because the page no longer exists in the browser history. Also check the history listing (in your browser's usual spot for this information) before and after clicking Replace Me.

LISTING 28-10

Invoking the location.replace() Method

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>location.replace() Method</title>
```

continued

LISTING 28-10 *(continued)*

```
      <script type="text/javascript">
      function doReplace()
      {
         location.replace("jsb28-01.html");
      }
      </script>
   </head>
   <body>
      <form name="myForm">
         <input type="button" value="Replace Me" onclick="doReplace()" />
      </form>
   </body>
</html>
```

Related Item: `history` object

history Object

Properties	Methods	Event Handler
`current`	`back()`	(None)
`length`	`forward()`	
`next`	`go()`	
`previous`		

Syntax

Accessing `history` object properties or methods:

```
[window.]history.property | method([parameters])
```

About this object

As a user surfs the web, the browser maintains a list of URLs for the most recent stops. This list is represented in the scriptable object model by the `history` object. A script cannot surreptitiously extract actual URLs maintained in this list unless you use signed scripts (in NN4+/Moz; see Chapter 49, "Security and Netscape Signed Scripts" on the CD-ROM), and the user grants permission. Under unsigned conditions, a script can methodically navigate to each URL in the history (by relative number or by stepping back one URL at a time), in which case the user sees the browser navigating on its own as though possessed by a spirit. Good Netiquette dictates that you do not navigate a user outside your web site without the user's explicit permission.

One application for the `history` object and its `back()` or `go()` methods is to provide the equivalent of a Back button in your HTML documents. That button triggers a script that checks for any items in the history list and then goes back one page. Your document doesn't have to know anything about the URL from which the user lands at your page; it delegates the specifics of the navigation back to the browser.

The behavior of the Back and Forward buttons is also available through a pair of window methods: `window.back()` and `window.forward()`. The `history` object methods are specific to a frame that is part of the reference. When the `parent.frameName.history.back()` method reaches the end of history for that frame, further invocations of that method are ignored.

IE's history mechanism is not localized to a particular frame of a frameset. Instead, the `history.back()` and `history.forward()` methods mimic the physical act of clicking the toolbar buttons. If you want to ensure cross-browser, if not cross-generational, behavior in a frameset, address references to the `history.back()` and `history.forward()` methods to the parent window.

Opera's history mechanism is also interesting. Let's say that you are developing a web page. You load it into Opera and observe how it behaves. You then edit the page, save it, and go back to Opera to observe the behavior. If you were to click the back button or use any of the methods described in this chapter, you would go back to the previous version of the web page you just edited.

You should use the `history` object and its methods with extreme care. Your design must be smart enough to watch what the user is doing with your pages (for example, by checking the current URL before navigating with these methods). Otherwise, you run the risk of confusing your user by navigating to unexpected places. Your script can also get into trouble because it cannot detect where the current document is in the Back/Forward sequence in history. Your script is also at the mercy of how the browser's history mechanism operates.

Properties

current
next
previous

Value: String Read-Only

Compatibility: WinIE-, MacIE-, NN4+, Moz+, Safari-, Opera+ (current only), Chrome-

To determine where to go when you click the Back and Forward buttons, the browser maintains a list of URLs visited. To someone trying to invade your privacy and see what sites and pages you frequent, this information is valuable. That's why the three properties that expose the actual URLs in the history list are restricted to pages with signed scripts (NN4+/Moz), and whose visitors have given permission to read sensitive browser data (see Chapter 49 on the CD-ROM).

With signed scripts and permission, you can look through the entire array of history entries in any frame or window. Because the list is an array, you can extract individual items by index value. For example, if the array has 10 entries, you can see the fifth item by using normal array indexing methods:

```
var fifthEntry = window.history[4];
```

No property or method exists that directly reveals the index value of the currently loaded URL, but you can script an educated guess by comparing the values of the current, next, and previous properties of the `history` object against the entire list.

We personally don't like some unknown entity watching over our shoulders while we're on the Net, so we respect that same feeling in others and therefore discourage the use of these powers, unless the user is given adequate warning. The signed script permission dialog box does not offer enough detail about the consequences of revealing this level of information. This means that you should explicitly notify users of the fact that you are accessing their history, even when you have implicit permission via a signed script.

Related Item: `history.length` property

length

Value: Number Read-Only

Compatibility: WinIE3+, MacIE3+, NN2+, Moz+, Safari+, Opera+, Chrome+

Use the `history.length` property to count the items in the history list. Unfortunately, this nugget of information is not particularly helpful in scripting navigation relative to the current location because your script cannot extract anything from the place in the history queue where the current document is located. If the current document is at the top of the list (the most recently loaded), you can calculate relative to that location. But users can use the Go/View menu to jump around the history list as they like. The position of a listing in the history list does not change by virtue of navigating back to that document. A `history.length` of 1, however, indicates that the current document is the first one the user has loaded since starting the browser software.

Listing 28-11 shows how to use the `length` property to notify users of how many pages they've visited.

LISTING 28-11

A Browser History Count

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>History Object</title>
      <script type="text/javascript">
      function showCount()
      {
         var histCount = window.history.length;
         if (histCount > 5)
         {
            alert("My, my, you\'ve been busy. You have visited " + histCount +
               " pages so far.");
         }
         else
         {
            alert("You have been to " + histCount + " Web pages this
               session.");
         }
```

```
        }
      </script>
   </head>
   <body>
      <form>
         <input type="button" name="activity" value="My Activity"
         onclick="showCount()" />
      </form>
   </body>
</html>
```

Related Items: None

Methods

back()
forward()

Returns: Nothing

Compatibility: WinIE3+, MacIE3+, NN2+, Moz+, Safari+, Opera+, Chrome+

Although the names might lead you to believe that these methods mimic the buttons on a browser's toolbar, they do not. The `history.back()` method is window/frame specific, meaning that if you direct successive `back()` methods to a frame within a frameset, the method is ignored when it reaches the first document to be loaded into that frame. The Back button and the `window.back()` method unload the frameset and continue taking you back through the browser's global history.

If you deliberately lead a user to a dead end in your web site, you should make sure that the HTML document provides a way to navigate back to a recognizable spot. Because you can easily create a new window that has no toolbar or menu bar (non-Macintosh browsers), you may end up stranding your users because they have no way of navigating out of a cul-de-sac in such a window. A button in your document should give the user a way back to the last location.

Unless you need to perform some additional processing prior to navigating to the previous location, you can simply place this method as the parameter to the event handler attribute of a button definition. To guarantee compatibility across all browsers, direct this method at the parent document when it's used from within a frameset.

Less likely to be scripted than the `history.back()` action is the method that performs the opposite action: navigating forward one step in the browser's history list. The only time you can confidently use the `history.forward()` method is in balancing the use of the `history.back()` method in the same script — where your script closely keeps track of how many steps the script heads in either direction. Use the `history.forward()` method with extreme caution, and only after performing extensive user testing on your web pages to make sure that you've covered all user possibilities. Similar to navigating backward via `history.back()`, forward progress when using `history.forward()` extends only through the history listing for a given window or frame, not the entire browser history list.

Listing 28-12 and Listing 28-13 provide a little workshop in which you can test the behavior of a variety of forms of backward and forward navigation in different browsers.

LISTING 28-12

Navigation Lab Frameset

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
   <title>Back and Forward</title>
   </head>
   <frameset cols="45%,55%">
      <frame name="controller" src="jsb28-13.html" />
      <frame name="display" src="jsb28-01.html" />
   </frameset>
</html>
```

LISTING 28-13

Navigation Lab Control Panel

```
<!DOCTYPE html>
<html>
  <head>
    <meta http-equiv="content-type" content="text/html;charset=utf-8">
    <title>Lab Controls</title>
  </head>
  <body>
    <b>Load a series of documents into the right frame by clicking some
    of these links (make a note of the sequence you click on):</b>
    <p><a href="jsb28-01.html" target="display">Listing 28-1</a><br />
       <a href="jsb28-05.html" target="display">Listing 28-5</a><br />
       <a href="jsb28-09.html" target="display">Listing 28-9</a><br /></p>
    <hr />
    <form name="input">
        <b>Click on the various buttons below to see the results in this
        frameset:</b>
        <ul>
           <li><tt>history.back()</tt> and <tt>history.forward()</tt> for
              righthand frame:<input type="button" value="Back"
             onclick="parent.display.history.back()" /><input type="button"
              value="Forward" onclick="parent.display.history.forward()" />
              </li>
           <li><tt>history.back()</tt> for this frame:<input type="button"
               value="Back" onclick="history.back()" /></li>
           <li><tt>history.back()</tt> for parent:<input type="button"
               value="Back" onclick="parent.history.back()" /></li>
        </ul>
     </form>
  </body>
</html>
```

Related Items: `history.go()` method

go(*relativeNumber* | *"URLOrTitleSubstring"*)

Returns: Nothing

Compatibility: WinIE3+, MacIE3+, NN2+, Moz+, Safari+, Opera+, Chrome+

Use the `history.go()` method to script navigation within the history list currently stored in the browser. If you elect to use a URL as a parameter, however, that precise URL must already exist in the history listing. Therefore, do not regard this method as an alternative to setting the `window.location` object to a brand-new URL.

For navigating `n` steps in either direction along the history list, use the `relativeNumber` parameter of the `history.go()` method. This number is an integer value that indicates which item in the list to use, relative to the current location. For example, if the current URL is at the top of the list (that is, the Forward button in the toolbar is dimmed), you need to use the following method to jump to the URL two items backward in the list:

```
history.go(-2);
```

In other words, the current URL is the equivalent of `history.go(0)` (a method that reloads the window). A positive integer indicates a jump that many items forward in the history list. Thus, `history.go(-1)` is the same as `history.back()`, whereas `history.go(1)` is the same as `history.forward()`.

Alternatively, you can specify one of the URLs or document titles stored in the browser's history list (titles appear in the Go/View menu). As security and privacy concerns have increased over time, this variant of the `go()` method has been reined in. It's best not to use the string parameter in your scripting; in fact, none of the current browser versions support this variant.

Like most other history methods, your script finds it difficult to manage the history list or the current URL's spot in the queue. That fact makes it even more difficult for your script to determine intelligently how far to navigate in either direction, or to which specific URL or title matches it should jump. Use this method only for situations in which your web pages are in strict control of the user's activity (or for designing scripts for yourself that automatically crawl around sites according to a fixed regimen). When you give the user control over navigation, you have no guarantee that the history list will be what you expect, and any scripts you write that depend on a `history` object will likely break.

In practice, this method mostly performs a soft reload of the current window using the `0` parameter.

Tip

If you are developing a page for all scriptable browsers, be aware that the reloading of a page with `history.go(0)` in IE often returns to the server to reload the page rather than reloading from the cache. ■

Listing 28-14 contains sample code that demonstrates how to navigate the history list via the `go()` method. Fill in either the number or text field of the page in Listing 28-14, and then click the associated button. The script passes the appropriate kind of data to the `go()` method. Unless you are using an older browser version, using the matching string text box and its button will do nothing. Be sure to use negative numbers for visiting a page earlier in the history.

Note

Mozilla browsers respond only to the integer offset approach to using the `history.go()` method. ■

LISTING 28-14

Navigating to an Item in History

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>history.go() Method</title>
      <script type="text/javascript">
      function doGoNum(form) {
         window.history.go(parseInt(form.histNum.value));
      }
      function doGoTxt(form) {
         window.history.go(form.histWord.value);
      }
      </script>
   </head>
   <body>
      <form>
         <b>Calling the history.go() method:</b>
         <hr />
         Enter a number (+/-):<input type="text" name="histNum" size="3"
         value="0" /> <input type="button" value="Go to Offset"
         onclick="doGoNum(this.form)" />
         <p><div>Enter either a word found in a document title in your
                 history, or a word in a document URL in your history
                 (remember, this will only work if your browser version is
                 older):</div>
         <input type="text" name="histWord" />
            <input type="button" value="Go to Match"
            onclick="doGoTxt(this.form)" /></p>
      </form>
   </body>
</html>
```

Related Items: `history.back()`, `history.forward()`, `location.reload()` methods

CHAPTER 29

Document and Body Objects

IN THIS CHAPTER

Accessing arrays of objects contained by the document object

Writing new document content to a window or frame

Using the body element for IE window measurements

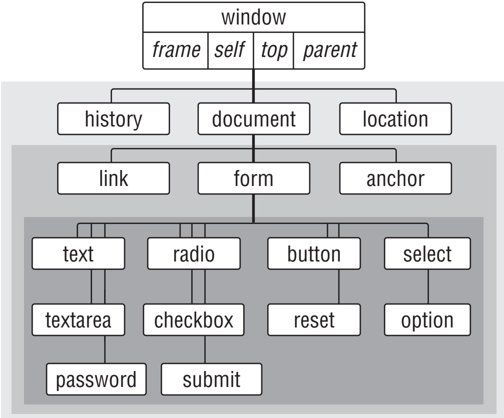
User interaction is a vital aspect of client-side JavaScript scripting, and most of the communication between script and user takes place by way of the `document` object and its components. Understanding the scope of the `document` object within each of the object models you support is key to implementing successful cross-browser applications.

Review the `document` object's place within the original object hierarchy. Figure 29-1 shows that the `document` object is a pivotal point for a large percentage of objects. In the W3C DOM, the `document` object plays an even more important role as the container of all element objects delivered with the page: The `document` object is the root of the entire document tree.

In fact, the `document` object and all that it contains is so big that we have divided its discussion into many chapters, each focusing on related object groups. This chapter looks at the `document` object and `body` object (which have conceptual relationships), whereas each of the succeeding chapters in this part of the book details objects contained by the `document` object.

FIGURE 29-1

The basic document object model hierarchy.



document Object

Compatibility: WinIE3+, MacIE3+, NN2+, Moz+, Safari+, Opera+, Chrome+

Properties	Methods	Event Handlers
`activeElement`	`attachEvent()`†	`onactivate`†
`alinkColor`	`captureEvents()`	`onbeforecut`†
`all`†	`clear()`	`onbeforedeactivate`†
`anchors[]`	`clearAttributes()`†	`onbeforeeditfocus`†
`applets[]`	`close()`	`onbeforepaste`†
`attributes`†	`createAttribute()`	`onclick`†
`baseURI`	`createCDATASection()`	`oncontextmenu`†
`bgColor`	`createComment()`	`oncontrolselect`†
`body`	`createDocumentFragment()`	`oncut`†
`charset`	`createElement()`	`ondblclick`†
`characterSet`	`createElementNS()`	`ondrag`†
`childNodes`†	`createEvent()`	`ondragend`†
`compatMode`	`createEventObject()`	`ondragenter`†
`contentType`	`createNSResolver()`	`ondragleave`†
`cookie`	`createRange()`	`ondragover`†

Properties	Methods	Event Handlers
defaultCharset	createStyleSheet()	ondragstart†
defaultView	createTextNode()	ondrop†
designMode	createTreeWalker()	onhelp†
doctype	detachEvent()†	onkeydown†
documentElement	elementFromPoint()	onkeypress†
documentURI	evaluate()	onkeyup†
domain	execCommand()	onmousedown†
embeds[]	focus()†	onmousemove†
expando	getElementById()	onmouseout†
fgColor	getElementsByName()	onmouseover†
fileCreatedDate	getElementsByTagName()†	onmouseup†
fileModifiedDate	getElementsByTagNameNS()†	onpaste†
fileSize	hasFocus()	onpropertychange†
firstChild†	importNode()	onreadystatechange†
forms[]	mergeAttributes()†	onresizeend†
frames[]	open()	onresizestart†
height	queryCommandEnabled()	onselectionchange
id†	queryCommandIndterm()	onstop
images[]	queryCommandState()	
implementation	queryCommandSupported()	
inputEncoding	queryCommandText()	
lastChild†	queryCommandValue()	
lastModified	recalc()	
layers[]	releaseCapture()†	
linkColor	releaseEvents()	
links[]	routeEvent()	
location	setActive()†	
media	write()	
mimeType	writeln()	
nameProp		
namespaces[]		

Properties	Methods	Event Handlers
`namespaceURI`†		
`nextSibling`†		
`nodeName`†		
`nodeType`†		
`ownerDocument`†		
`parentNode`†		
`parentWindow`		
`plugins[]`		
`previousSibling`†		
`protocol`		
`readyState`†		
`referrer`		
`scripts[]`		
`security`		
`selection`		
`strictErrorChecking`		
`styleSheets[]`		
`tags[]`		
`title`		
`uniqueID`†		
`URL`		
`URLUnencoded`		
`vlinkColor`		
`width`		
`xmlEncoding`		
`xmlStandalone`		
`xmlVersion`		

†See Chapter 26, ''Generic HTML Element Objects.''

Syntax

Accessing `document` object properties or methods:

```
[window.]document.property | method([parameters])
```

About this object

A document object encompasses the totality of what exists inside the content region of a browser window or window frame (excluding toolbars, status lines, and so on). The document is a combination of the content and interface elements that make the web page worth visiting. In modern browsers the document object also serves as the root node of a page's hierarchical tree of nodes — that from which all other nodes grow.

Because the document object isn't explicitly represented in an HTML document by tags or any other notation, the original designers of JavaScript and object models decided to make the document object the portal to many settings that were represented in HTML as belonging to the body element. Therefore, the body element's tag contains attributes for document-wide attributes, such as background color (bgcolor) and link colors in various states (alink, link, and vlink). The original designers decided to have the body element also serve as an HTML container for forms, links, and anchors. The document object, therefore, assumed a majority of the role of the body element. But even then, the document object became the most convenient place to bind some properties that extend beyond the body element, such as the title element and the URL of the link that referred the user to the page. When viewed within the context of the HTML source code, the original document object is somewhat schizophrenic. Even so, the document object has worked well as the basis for references to original object model objects, such as forms, images, and applets.

This, of course, was before every HTML element, including the body element, was exposed as an object through modern object models. Amazingly, even with the IE4+ object model and W3C DOM — both of which treat the body element as an object separate from the document object — script compatibility with the original object model is quite easily accomplished. The document object has assumed a new schizophrenia, splitting its personality between the original object model and the one that places the document object at the root of the hierarchy, quite separate from the body element object it contains. The object knows which face to put on based on the rest of the script syntax that follows it. This means that quite often there are multiple ways to achieve the same reference. For example, you can use the following statement in all scriptable browsers to get the number of form objects in a document:

```
document.forms.length
```

In IE4+, you can also use

```
document.tags["form"].length
```

And in the W3C DOM as implemented in IE5+ and NN6+/Moz/Safari/Opera/Chrome, you can use

```
document.getElementsByTagName("form").length
```

Modern browsers provide a generic approach to accessing elements (getElementsByTagName() method in the W3C DOM) to meet the requirements of object models that expose every HTML (and XML) element as an object.

Promoting the body element to the ranks of exposed objects presented its own challenges to the new object model designers. The body element is the true owner of some properties that the original document object had to take on by default. Most properties that belonged to the original document object were renamed in their transfer to the body element. For example, the original document.alinkColor property is the body.aLink property in the modern model. But the bgColor property has not been renamed. For the sake of code compatibility, modern browsers recognize both properties, even though the W3C DOM has removed the old versions of the properties

for the newly conceived `document` object. Considering the fact that modern browsers are now prevalent, you should be able to stick with the new properties from here on.

Properties

activeElement

Value: Object reference Read-Only

Compatibility: WinIE4+, MacIE4+, NN-, Moz+, Safari+, Opera+, Chrome+

Originally just in IE4+, a script can examine the `document.activeElement` property to see which element currently has focus. The value returned is an element object reference. You can use any of the properties and methods listed in Chapter 26 to find out more about the object.

Although the element used to generate a mouse or keyboard event will most likely have focus, don't rely on the `activeElement` property to find out which element generated an event. The IE `event.srcElement` property is far more reliable.

Example

Use The Evaluator (see Chapter 4, "JavaScript Essentials") with IE4+ to experiment with the `activeElement` property. Type the following statement into the top text box:

```
document.activeElement.value
```

After you press the Enter key, the Results box shows the value of the text box you just typed into (the very same expression you just typed). But if you then click the Evaluate button, you will see the `value` property of that button object appear in the Results box.

Related Item: `event.srcElement` property

alinkColor
bgColor
fgColor
linkColor
vlinkColor

Value: Hexadecimal triplet or color name string Mostly Read/Write

Compatibility: WinIE3+, MacIE3+, NN2+, Moz+, Safari+, Opera+, Chrome+

These five properties are the script equivalent of the `<body>` tag attributes of the same name (although the property names are case-sensitive). All five settings can be read via the `document.body` object in modern browsers. Values for all color properties can be either the common HTML hexadecimal triplet value (for example, `"#00FF00"`) or any of the standard color names.

Example

We select some color values at random to plug into three settings of the group of ugly colors for Listing 29-1. The smaller window displays a dummy button so that you can see how its display contrasts with color settings. Notice that the script sets the colors of the smaller window by rewriting the entire window's HTML code. After changing colors, the script displays the color values in the original window's textarea. Even though some colors are set with the color constant values, properties come

back in the hexadecimal triplet values. You can experiment to your heart's content by changing color values in the listing. Every time you change the values in the script, save the HTML file and reload it in the browser.

LISTING 29-1

Tweaking the Color of Page Elements

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Color Me</title>
      <script type="text/javascript">
         // may be blocked at load time by browser popup blockers
         var newWindow = window.open("","","height=150,width=300");

         function defaultColors()
         {
            return "bgcolor='#c0c0c0' vlink='#551a8b' link='#0000ff'";
         }

         function uglyColors()
         {
            return "bgcolor='yellow' vlink='pink' link='lawngreen'";
         }

         function showColorValues()
         {
            var result = "";
            result += "bgColor: " + newWindow.document.bgColor + "\n";
            result += "vlinkColor: " + newWindow.document.vlinkColor + "\n";
            result += "linkColor: " + newWindow.document.linkColor + "\n";
            document.forms[0].results.value = result;
         }

         // dynamically writes contents of another window
         function drawPage(colorStyle)
         {
            // work around popup blockers
            if (!newWindow || newWindow.closed)
            {
               newWindow = window.open("","","height=150,width=300");
            }
            var thePage = "";
            thePage += "<html><head><title>Color Sampler<\/title><\/head><body ";
            if (colorStyle == "default")
            {
               thePage += defaultColors();
            }
            else
            {
```

continued

LISTING 29-1 *(continued)*

```
            thePage += uglyColors();
        }
        thePage += ">Just so you can see the variety of items and colors, <a ";
        thePage += "href='http://www.nowhere.com'>here\'s a link<\/a>, and <a
        href='http://home.netscape.com'> here is another link <\/a> you can
        use on-line to visit and see how its color differs from the standard
        link.";
        thePage += "<form>";
        thePage += "<input type='button' name='sample' value='Just a Button'>";
        thePage += "<\/form><\/body><\/html>";
        newWindow.document.write(thePage);
        newWindow.document.close();
        showColorValues();
    }

    // the following works properly only in Windows Navigator
    function setColors(colorStyle)
    {
        if (colorStyle == "default")
        {
            document.bgColor = "#c0c0c0";
        }
        else
        {
            document.bgColor = "yellow";
        }
    }

    // bind the event handlers
    function addEvent(elem, evtType, func)
    {
        if (elem.addEventListener)
        {
            elem.addEventListener(evtType, func, false);
        }
        else if (elem.attachEvent)
        {
            elem.attachEvent("on" + evtType, func);
        }
        else
        {
            elem["on" + evtType] = func;
        }
    }
    addEvent(window, "load", function()
    {
        addEvent(document.getElementById("default1"), "click",
            function(evt) {drawPage("default")});
        addEvent(document.getElementById("weird1"), "click",
            function(evt) {drawPage("ugly")});
```

```
            addEvent(document.getElementById("default2"), "click",
               function(evt) {setColors("default")});
            addEvent(document.getElementById("weird2"), "click",
               function(evt) {setColors("ugly")});
         });
      </script>
   </head>
   <body>
      Try the two color schemes on the document in the small window.
      <form>
         <input type="button" id="default1" name="default" value='Default Colors' />
         <input type="button" id="weird1" name="weird" value="Ugly Colors" />
         <p>
            <textarea name="results" rows="3" cols="20"></textarea>
         </p>
         <hr />
         These buttons change the current document.
         <p>
            <input type="button" id="default2" name="default"
                value='Default Colors' />
            <input type="button" id="weird2" name="weird" value="Ugly Colors" />
         </p>
      </form>
   </body>
</html>
```

Note

The examples in this chapter take advantage of the modern approach to event handling, which involves the `addEventListener()` **(NN6+/Moz/W3C) and** `attachEvent()` **(IE5+) methods. This event-handling technique is explained in detail in Chapter 32, "Event Objects."** ■

Related Items: `body.aLink`, `body.bgColor`, `body.link`, `body.text`, `body.vLink` properties

anchors[]

Value: Array of anchor objects Read-Only

Compatibility: WinIE3+, MacIE3+, NN2+, Moz+, Safari+, Opera+, Chrome+

Anchor objects (described in Chapter 30, "Link and Anchor Objects") are points in an HTML document marked with `<a name="">` tags. Anchor objects are referenced in URLs by a hash value between the page URL and anchor name. Like other object properties that contain a list of nested objects, the `document.anchors` property delivers an indexed array of anchors in a document. Use the array references to pinpoint a specific anchor for retrieving any anchor property.

Anchor arrays begin their index counts with `0`: The first anchor in a document, then, has the reference `document.anchors[0]`. And, as is true with any built-in array object, you can find out how many entries the array has by checking the `length` property. For example:

```
alert("This document has " + document.anchors.length + " anchors.");
```

***documentObject*.anchors[]**

The `document.anchors` property is read-only. To script navigation to a particular anchor, assign a value to the `window.location` or `window.location.hash` object, as described in the `location` object discussion in Chapter 28, "Location and History Objects."

Example

In Listing 29-2, we append an extra script to Listing 28-1 to demonstrate how to extract the number of anchors in the document. The document dynamically writes the number of anchors found in the document. You will not likely ever need to reveal such information to users of your page, and the `document.anchors` property is not one that you will call frequently. The object model defines it automatically as a document property while defining actual anchor objects.

LISTING 29-2

Using Anchors to Navigate Through a Page

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>document.anchors Property</title>
      <script type="text/javascript">
         function goNextAnchor(where)
         {
            window.location.hash = where;
         }

         // bind the event handlers
         function addEvent(elem, evtType, func)
         {
            if (elem.addEventListener) {
               elem.addEventListener(evtType, func, false);
            }
            else if (elem.attachEvent)
            {
               elem.attachEvent("on" + evtType, func);
            }
            else
            {
               elem["on" + evtType] = func;
            }
         }
         addEvent(window, "load", function()
         {
            addEvent(document.getElementById("next1"), "click",
               function(evt) {goNextAnchor("sec1")});
            addEvent(document.getElementById("next2"), "click",
               function(evt) {goNextAnchor("sec2")});
            addEvent(document.getElementById("next3"), "click",
               function(evt) {goNextAnchor("sec3")});
            addEvent(document.getElementById("next4"), "click",
```

```
            function(evt) {goNextAnchor("start")});
      });
   </script>
</head>
<body>
   <h1><a id="start" name="start">Top</a></h1>
   <form>
      <input type="button" id="next1" name="next" value="NEXT" />
   </form>
   <hr />
   <h1><a id="sec1" name="sec1">Section 1</a></h1>
   <form>
      <input type="button" id="next2" name="next" value="NEXT" />
   </form>
   <hr />
   <h1><a id="sec2" name="sec2">Section 2</a></h1>
   <form>
      <input type="button" id="next3" name="next" value="NEXT" />
   </form>
   <hr />
   <h1><a id="sec3" name="sec3">Section 3</a></h1>
   <form>
      <input type="button" id="next4" name="next" value="BACK TO TOP" />
   </form>
   <hr />
   <p>
      <script type="text/javascript">
        document.write("<i>There are " +
                       document.anchors.length +
                       " anchors defined for this document<\/i>")
      </script>
   </p>
</body>
</html>
```

Related Items: `anchor`, `location` objects; `document.links` property

applets[]

Value: Array of applet objects Read-Only

Compatibility: WinIE3+, MacIE3+, NN2+, Moz+, Safari+, Opera+, Chrome+

The `applets` property refers to Java applets defined in a document by the `<applet>` tag. An applet is not officially an object in the document until the applet loads completely.

Most of the work you do with Java applets from JavaScript takes place through the methods and variables defined inside the applet. Although you can reference an applet according to its indexed array position within the `applets` array, you will more likely use the applet object's name in the reference to avoid any confusion.

Example

The `document.applets` property is defined automatically as the browser builds the object model for a document that contains applet objects. You will rarely access this property, except to determine how many applet objects a document has, as in this example:

```
var numApplets = document.applets.length;
```

Related Item: `applet` object

baseURI

Value: String   Read-Only

Compatibility: WinIE-, MacIE-, NN7+, Moz+, Safari+, Opera+, Chrome+

The `baseURI` property reveals the absolute base URI of the document. You can check the base URI of a document in the Evaluator (see Chapter 4, "JavaScript Essentials") by entering the following:

```
document.baseURI
```

Related Item: `document.documentURI` property

bgColor

(See `alinkColor`)

body

Value: `body` element object   Read/Write

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

The `document.body` property is a shortcut reference to the `body` element object in modern object models. As you can see in the discussion of the `body` element object later in this chapter, that object has many key properties that govern the look of the entire page. Because the `document` object is the root of all references within any window or frame, the `document.body` property is easier to use to get to the `body` properties than the longer references normally used to access HTML element objects in both the IE4+ and W3C object models.

Example

Use The Evaluator (see Chapter 4, "JavaScript Essentials") to examine properties of the `body` element object. First, to prove that the `document.body` is the same as the element object that comes back from longer references, enter the following statement into the top text box with either IE5+, NN6+/Moz, or some other W3C browser:

```
document.body == document.getElementsByTagName("body")[0]
```

Next, check out the `body` object's property listings later in this chapter and enter the listings into the top text box to review their results. For example:

```
document.body.bgColor
document.body.tagName
```

The main point to take from this example is that the `document.body` reference provides a simpler and more direct means of accessing a document's `body` object without having to use the `getElementsByTagName()` method.

Related Item: `body element` object

charset

Value: String Read/Write

Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari+, Opera+, Chrome+

The `charset` property reveals the character set originally used by the browser (IE4+) to render the current document (the NN6+/Moz version of this property is called `characterSet`). You can find possible values for this property at

```
ftp://ftp.isi.edu/in-notes/iana/assignments/character-sets
```

Each browser and operating system has its own default character set. Values may also be set through a `<meta>` tag.

Example

Use The Evaluator (Chapter 4, "JavaScript Essentials") to experiment with the `charset` property. To see the default setting applied to the page, enter the following statement into the top text box:

```
document.charset
```

If you are running IE5+ for Windows and you enter the following statement, the browser will apply a different character set to the page:

```
document.charset = "iso-8859-2"
```

If your version of Windows does not have that character set installed in the system, the browser may ask permission to download and install the character set.

Related Items: `characterSet`, `defaultCharset` properties

characterSet

Value: String Read/Write

Compatibility: WinIE-, MacIE-, NN6+, Moz+, Safari+, Opera+, Chrome+

The `characterSet` property reveals the character set used by the browser to render the current document (the IE version of this property is called `charset`). You can find possible values for this property at

```
http://www.iana.org/assignments/character-sets
```

Each browser and operating system has its own default character set. Values may also be set through a `<meta>` tag.

Example

Use The Evaluator (Chapter 4, "JavaScript Essentials") to experiment with the `characterSet` property in NN6+/Moz. To see the default setting applied to the page, enter the following statement into the top text box:

```
document.characterSet
```

Related Item: `charset` property

compatMode

Value: String Read-Only

Compatibility: WinIE6+, MacIE6+, NN7+, Moz+, Safari+, Opera+, Chrome+

The `compatMode` property reveals the compatibility mode for the document, as determined by the `DOCTYPE` element's content. The value for this property can be one of the following string constants: `BackCompat` or `CSS1Compat`. The default setting for the `compatMode` property is `BackCompat`, which means that the document is not W3C standards-compliant; in other words, the document is in Quirks mode. Otherwise the document is in Strict mode and is W3C standards-compliant. By *standards-compliant* I'm referring to the CSS1 standard.

Example

You may find it useful to check the compatibility mode of a document in order to carry out processing specific to one of the modes. Following is an example of how you might branch to carry out processing for backward-compatible documents:

```
if (document.compatMode == "BackCompat") {
   // perform backward compatible processing
}
```

Related Items: *Standards Compatibility Modes* (see Chapter 4, "JavaScript Essentials")

contentType

Value: String Read-Only

Compatibility: WinIE-, MacIE-, NN7+, Moz+, Safari-, Opera-, Chrome-

The `contentType` property holds the content type (MIME type) of the document. For a normal HTML document, the value of this property is `text/html`.

cookie

Value: String Read/Write

Compatibility: WinIE3+, MacIE3+, NN2+, Moz+, Safari+, Opera+, Chrome+

The cookie mechanism in a web browser lets you store small pieces of information on the client computer in a reasonably secure manner. In other words, when you need some tidbit of information to persist at the client level while either loading diverse HTML documents or moving from one session to another, the cookie mechanism saves the day. The cookie is commonly used as a means to store the username and password you enter into a password-protected web site. The first time you enter this

information into a form, the server-side form processing program has the browser write the information back to a cookie on your hard disk (usually after encrypting the password). Rather than bothering you to enter the username and password the next time you access the site, the server searches the cookie data stored for that particular server and extracts the username and password for automatic validation processing behind the scenes.

Other applications of the cookie include storing user preferences and information about the user's previous visit to the site. Preferences may include font styles or sizes and whether the user prefers viewing content inside a frameset or not. As shown in Chapter 57, "Application: Intelligent 'Updated' Flags" (on the CD-ROM), a time stamp of the previous visit can allow a coded HTML page to display highlighted images next to content that has changed since the user's last visit, even if you have updated the page several times in the interim. Rather than hard-wiring new flags for *your* last visit, the scripts highlight what's new for the visitor.

The cookie file

Allowing some foreign server program to read from and write to your hard disk may give you pause, but browser cookie mechanisms don't just open up your drive's directory for the world to see (and to corrupt). Instead, the cookie mechanism provides access to a type of text file (Internet Explorer) or just one special text file (browsers other than IE) located in a platform-specific spot on your drive.

In Mozilla-based browsers, for example, the cookie file is named `cookies.txt` and is located in a directory (whose name ends in `.slt`) within the browser's profile area. In Windows, that location is `C:\\Windows\Application Data\Mozilla\Profiles\[profilename]\`; in Mac OSX, the location is `[user]/Library/Mozilla/Profiles/[profilename]/`. Internet Explorer for Windows uses a different filing system: all cookies for each domain are saved in a domain-specific file inside the `C:\\Windows\Temporary Internet Files\` directory. Filenames begin with `Cookie:` and include the username and domain of the server that wrote the cookie. Safari cookies are recorded in an XML file named `Cookies.plist` within the `[user]/Library/Cookies/` directory. Opera cookies are recorded in a binary file named `cookies4.dat` within the `[user]\Application Data\Opera\Opera` directory. Google Chrome cookies are recorded in a SQLite database file named `Cookies` within the `[user]\Local Settings\Application Data\Google\Chrome\User Data\Default` directory. Be aware that if you are testing with Google Chrome, your code will not work unless the files are coming from an HTTP server. Chrome intentionally disables cookies on `file:///` documents.

A cookie file is a text file. If curiosity drives you to open a cookie file, we recommend that you do so only with a copy saved in another directory or folder. Any alteration to the existing file can mess up whatever valuable cookies are stored there for sites you regularly visit. The data format for cookie files differs across browsers, in line with the different methodologies used for filing cookies. Inside the Mozilla file (after a few comment lines warning you not to manually alter the file) are lines of tab-delimited text. Each return-delimited line contains one cookie's information. The cookie file is just like a text listing of a database. In each of the IE cookie files, the same data points are stored for a cookie as for Mozilla, but the items are in a return-delimited list. The structure of these files is of no importance to scripting cookies, because all browsers utilize the same syntax for reading and writing cookies through the `document.cookie` property.

Note

As you experiment with browser's cookies, you will be tempted to look into the cookie file after a script writes some data to the cookie. The cookie file usually will not contain the newly written data, because in most browsers cookies are transferred to disk only when the user quits the browser; conversely, the cookie file is read into the browser's memory when it is launched. While you read, write, and delete cookies during a browser session, all activity is performed in memory (to speed up the process) to be saved later. ■

A cookie record

Among the fields of each cookie record are the following (not necessarily in this order):

- Domain of the server that created the cookie
- Information on whether you need a secure HTTP connection to access the cookie
- Pathname of URL(s) capable of accessing the cookie
- Expiration date of the cookie
- Name of the cookie entry
- String data associated with the cookie entry

Note that cookies are domain-specific. In other words, if one domain creates a cookie, another domain cannot access it through the browser's cookie mechanism behind your back. That reason is why it's generally safe to store what we call *throwaway passwords* (the username/password pairs required to access some free registration-required sites) in cookies. Moreover, sites that store passwords in a cookie usually do so as encrypted strings, making it more difficult for someone to hijack the cookie file from your unattended PC and figure out what your personal password scheme may be.

Cookies also have expiration dates. Because some browsers may allow no more than a fixed number of cookies (1000 in Firefox), the cookie file can get pretty full over the years. Therefore, if a cookie needs to persist past the current browser session, it should have an expiration date established by the cookie writer. Browsers automatically clean out any expired cookies.

Not all cookies have to last beyond the current session, however. In fact, a scenario in which you use cookies temporarily while working your way through a web site is quite typical. Many shopping sites employ one or more temporary cookie records to behave as the shopping cart for recording items you intend to purchase. These items are copied to the order form at checkout time. But after you submit the order form to the server, that client-side data has no particular value. As it turns out, if your script does not specify an expiration date, the browser keeps the cookie fresh in memory without writing it to the cookie file. When you quit the browser, that cookie data disappears as expected.

JavaScript access

Scripted access to cookies from JavaScript is limited to setting the cookie (with a number of optional parameters) and getting the cookie data (but with none of the parameters).

The original object model defines cookies as properties of documents, but this description is somewhat misleading. If you use the default path to set a cookie (that is, the current directory of the document whose script sets the cookie in the first place), all documents in that same server directory have read and write access to the cookie. A benefit of this arrangement is that if you have a scripted application that contains multiple documents, all documents served from the same directory can share the cookie data. Modern browsers, however, impose a limit of 20 named cookie entries (that is, one name/value pair) for any domain. If your cookie requirements are extensive, you need to fashion ways of concatenating cookie data. (We do this in the Decision Helper application in Chapter 58 on the CD-ROM.)

Saving cookies

To write cookie data to the cookie file, you use a simple JavaScript assignment operator with the `document.cookie` property. But the formatting of the data is crucial to achieving success. Here is the syntax for assigning a value to a cookie (optional items are in brackets; placeholders for data you supply are in italics):

```
document.cookie = "cookieName=cookieData
        [; expires=timeInGMTString]
```

```
[; path=pathName]
[; domain=domainName]
[; secure]"
```

Let's examine each of the properties individually.

Name/Data

Each cookie must have a name and a string value (even if that value is an empty string). Such name/value pairs are fairly common in HTML, but they look odd in an assignment statement. For example, if you want to save the string "Fred" to a cookie named "userName," the JavaScript statement is

```
document.cookie = "userName=Fred";
```

If the browser sees no existing cookie in the current domain with this name, it automatically creates the cookie entry for you; if the named cookie already exists, the browser replaces the old data with the new data. Retrieving the `document.cookie` property at this point yields the following string:

```
userName=Fred
```

You can omit all the other cookie-setting properties, in which case the browser uses default values, as explained in a following section. For temporary cookies (those that don't have to persist beyond the current browser session), the name/value pair is usually all you need.

The entire name/value pair must be a single string with no semicolons, commas, or character spaces. To take care of spaces between words, preprocess the value with the JavaScript `encodeURIComponent()` function, which URI-encodes the spaces as `%20` — and then be sure to convert the value to restore the human-readable spaces (through `decodeURIComponent()`) when you retrieve the cookie later.

You cannot save a JavaScript array or object to a cookie. But with the help of the `Array.join()` method, you can convert an array to a string; use `String.split()` to re-create the array after reading the cookie at a later time.

Expires

Expiration dates, when supplied, must be passed as Greenwich Mean Time (GMT) strings (see Chapter 17, "The Date Object," about time data). To calculate an expiration date based on today's date, use the JavaScript `Date` object as follows:

```
var exp = new Date();
var oneYearFromNow = exp.getTime() + (365 * 24 * 60 * 60 * 1000);
exp.setTime(oneYearFromNow);
```

Since the `getTime()` and `setTime()` methods operate in milliseconds, the year you're adding to the current date must be converted to milliseconds. After making the calculation, the date is converted to the accepted GMT string format:

```
document.cookie = "userName=Fred; expires=" + exp.toGMTString();
```

In the cookie file, the expiration date and time is stored as a numeric value (in seconds), but to set it, you need to supply the time in GMT format. You can delete a cookie before it expires by setting the

named cookie's expiration date to a time and date earlier than the current time and date. The safest expiration parameter is

```
expires=Thu, 01-Jan-70 00:00:01 GMT;
```

Omitting the expiration date signals to the browser that this cookie is temporary. The browser never writes it to the cookie file and forgets it when you quit the browser.

Path

For client-side cookies, the default path setting (the current directory) is usually the best choice. You can, of course, create a duplicate copy of a cookie with a separate path (and domain) so that the same data is available to a document located in another area of your site (or the Web).

Domain

To help synchronize cookie data with a particular document (or group of documents), the browser matches the domain of the current document with the domain values of cookie entries in the cookie file. Therefore, if you were to display a list of all cookie data contained in a `document.cookie` property, you would get back all the name-value cookie pairs from the cookie file whose domain parameter matches that of the current document.

Unless you expect the document to be replicated in another server within your domain, you can usually omit the `domain` parameter when saving a cookie. Default behavior automatically supplies the domain of the current document to the cookie file entry. Be aware that a domain setting must have at least two periods, such as

```
.google.com
.hotwired.com
```

Or, you can write an entire URL to the domain, including the `http://` protocol.

SECURE

If you omit the `SECURE` parameter when saving a cookie, you imply that the cookie data is accessible to any document or server-side program from your site that meets the other domain- and path-matching properties. For client-side scripting of cookies, you should omit this parameter when saving a cookie.

Retrieving cookie data

Cookie data retrieved through JavaScript arrives as one string, which contains the whole name-data pair. Even though the cookie file stores other parameters for each cookie, you can retrieve only the name-data pairs through JavaScript. Moreover, when two or more (up to a maximum of 20) cookies meet the current domain criteria, these cookies are also lumped into that string, delimited by a semicolon and space. For example, a `document.cookie` string may look like this:

```
userName=Fred; password=NikL2sPacU
```

In other words, you cannot treat named cookies as objects. Instead, you must parse the entire cookie string, extracting the data from the desired name-data pair.

When you know that you're dealing with only one cookie (and that no more will ever be added to the domain), you can customize the extraction based on known data, such as the cookie name. For example, with a cookie name that is seven characters long, you can extract the data with a statement such as this:

```
var data = decodeURIComponent(document.cookie.substring
     (7,document.cookie.length));
```

The first parameter of the `substring()` method includes the equal sign to separate the name from the data; this is where the 7 comes from in the code. This example works with single cookies only because it assumes that the cookie starts at the beginning of the cookie file, which may not be the case if the file contains multiple cookies.

A better approach to cookie extraction is to create a general-purpose function that can work with single- or multiple-entry cookies. For example:

```
function getCookieData(labelName) {
   var labelLen = labelName.length;
   // read cookie property only once for speed
   var cookieData = document.cookie;
   var cLen = cookieData.length;
   var i = 0;
   var cEnd;
   while (i < cLen) {
      var j = i + labelLen;
      if (cookieData.substring(i,j) == labelName) {
         cEnd = cookieData.indexOf(";",j);
         if (cEnd == -1) {
            cEnd = cookieData.length;
         }
         return decodeURIComponent(cookieData.substring(j+1, cEnd));
      }
      i++;
   }
   return "";
}
```

Calls to this function pass the label name of the desired cookie as a parameter. The function parses the entire cookie string, chipping away any mismatched entries (through the semicolons) until it finds the cookie name.

If all of this cookie code still makes your head hurt, you can turn to a set of functions devised by experienced JavaScripter and web site designer Bill Dortch of hIdaho Design. His cookie functions provide generic access to cookies that you can use in all of your cookie-related pages. Listing 29-3 shows Bill's cookie functions, which include a variety of safety nets for date calculation bugs that appeared in some legacy versions of Netscape Navigator. The code is updated with modern URL encoding and decoding methods. Don't be put off by the length of the listing: Most of the lines are comments.

LISTING 29-3

Bill Dortch's Cookie Functions

```
<html>
   <head>
      <title>Cookie Functions</title>
   </head>
   <body>
      <script type="text/javascript">
      //
      //  Cookie Functions -- "Night of the Living Cookie" Version (25-Jul-96)
      //
      //  Written by:  Bill Dortch, hIdaho Design
      //  The following functions are released to the public domain.
      //
      //  This version takes a more aggressive approach to deleting
      //  cookies.  Previous versions set the expiration date to one
      //  millisecond prior to the current time; however, this method
      //  did not work in Netscape 2.02 (though it does in earlier and
      //  later versions), resulting in "zombie" cookies that would not
      //  die.  DeleteCookie now sets the expiration date to the earliest
      //  usable date (one second into 1970), and sets the cookie's value
      //  to null for good measure.
      //
      //  Also, this version adds optional path and domain parameters to
      //  the DeleteCookie function.  If you specify a path and/or domain
      //  when creating (setting) a cookie**, you must specify the same
      //  path/domain when deleting it, or deletion will not occur.
      //
      //  The FixCookieDate function must now be called explicitly to
      //  correct for the 2.x Mac date bug.  This function should be
      //  called *once* after a Date object is created and before it
      //  is passed (as an expiration date) to SetCookie.  Because the
      //  Mac date bug affects all dates, not just those passed to
      //  SetCookie, you might want to make it a habit to call
      //  FixCookieDate any time you create a new Date object:
      //
      //    var theDate = new Date();
      //    FixCookieDate (theDate);
      //
      //  Calling FixCookieDate has no effect on platforms other than
      //  the Mac, so there is no need to determine the user's platform
      //  prior to calling it.
      //
      //  This version also incorporates several minor coding improvements.
      //
      //  **Note that it is possible to set multiple cookies with the same
      //  name but different (nested) paths.  For example:
      //
      //    SetCookie ("color","red",null,"/outer");
      //    SetCookie ("color","blue",null,"/outer/inner");
```

```
//
//  However, GetCookie cannot distinguish between these and will return
//  the first cookie that matches a given name.  It is therefore
//  recommended that you *not* use the same name for cookies with
//  different paths.  (Bear in mind that there is *always* a path
//  associated with a cookie; if you don't explicitly specify one,
//  the path of the setting document is used.)
//
//  Revision History:
//
//    "JavaScript Bible 6th Edition" Version (28-July-2006)
//      - Replaced deprecated escape()/unescape() functions with
//        encodeURI() and decodeURI() functions
//
//    "Toss Your Cookies" Version (22-Mar-96)
//      - Added FixCookieDate() function to correct for Mac date bug
//
//    "Second Helping" Version (21-Jan-96)
//      - Added path, domain and secure parameters to SetCookie
//      - Replaced home-rolled encode/decode functions with
//        new (then) escape/unescape functions
//
//    "Free Cookies" Version (December 95)
//
//
//  For information on the significance of cookie parameters,
//  and on cookies in general, please refer to the official cookie
//  spec, at:
//
//      http://www.netscape.com/newsref/std/cookie_spec.html
//
//********************************************************************
//
// "Internal" function to return the decoded value of a cookie
//
function getCookieVal (offset)
{
  var endstr = document.cookie.indexOf (";", offset);
    if (endstr == -1)
    {
      endstr = document.cookie.length;
    }
  return decodeURI(document.cookie.substring(offset, endstr));
}

//
//  Function to correct for 2.x Mac date bug.  Call this function to
//  fix a date object prior to passing it to SetCookie.
//  IMPORTANT:  This function should only be called *once* for
//  any given date object!  See example at the end of this document.
//
```

continued

LISTING 29-3 *(continued)*

```
function FixCookieDate (date)
{
  var base = new Date(0);
  var skew = base.getTime(); // dawn of (Unix) time - should be 0
    if (skew > 0)
    {  // Except on the Mac - ahead of its time
       date.setTime (date.getTime() - skew);
    }
}

//
//  Function to return the value of the cookie specified by "name".
//    name - String object containing the cookie name.
//    returns - String object containing the cookie value, or null if
//      the cookie does not exist.
//
function GetCookie (name)
{
   var arg = name + "=";
   var alen = arg.length;
   var clen = document.cookie.length;
   var i = 0;
   while (i < clen)
   {
     var j = i + alen;
       if (document.cookie.substring(i, j) == arg)
       {
          return getCookieVal (j);
       }
     i = document.cookie.indexOf(" ", i) + 1;
       if (i == 0)
       {
          break;
       }
   }
   return null;
}

//
//  Function to create or update a cookie.
//    name - String object containing the cookie name.
//    value - String object containing the cookie value.  May contain
//      any valid string characters.
//    [expires] - Date object containing the expiration data of the
//      cookie.  If omitted or null, expires the cookie at the end of the
//      current session.
//    [path] - String object indicating the path for which the cookie is
//      valid.
//      If omitted or null, uses the path of the calling document.
```

```
//    [domain] - String object indicating the domain for which the cookie
//      is valid. If omitted or null, uses the domain of the calling
//      document.
//    [secure] - Boolean (true/false) value indicating whether cookie
//      transmission requires a secure channel (HTTPS).
//
//  The first two parameters are required.  The others, if supplied, must
//  be passed in the order listed above.  To omit an unused optional
//  field, use null as a place holder.  For example, to call SetCookie
//  using name, value and path, you would code:
//
//      SetCookie ("myCookieName", "myCookieValue", null, "/");
//
//  Note that trailing omitted parameters do not require a placeholder.
//
//  To set a secure cookie for path "/myPath", that expires after the
//  current session, you might code:
//
//      SetCookie (myCookieVar, cookieValueVar, null, "/myPath", null,
//         true);
//
function SetCookie (name,value,expires,path,domain,secure)
{
   document.cookie = name + "=" + encodeURIComponent (value) +
      ((expires) ? "; expires=" + expires.toGMTString() : "") +
      ((path) ? "; path=" + path : "") +
      ((domain) ? "; domain=" + domain : "") +
      ((secure) ? "; secure" : "");
}

//  Function to delete a cookie. (Sets expiration date to start of epoch)
//    name -   String object containing the cookie name
//    path -   String object containing the path of the cookie to delete.
//             This MUST be the same as the path used to create the
//             cookie, or null/omitted if
//             no path was specified when creating the cookie.
//    domain - String object containing the domain of the cookie to
//             delete.  This MUST be the same as the domain used to
//             create the cookie, or null/omitted if no domain was
//             specified when creating the cookie.
//
function DeleteCookie (name,path,domain)
{
   if (GetCookie(name))
   {
      document.cookie = name + "=" +
         ((path) ? "; path=" + path : "") +
         ((domain) ? "; domain=" + domain : "") +
         "; expires=Thu, 01-Jan-70 00:00:01 GMT";
   }
}
```

continued

LISTING 29-3 *(continued)*

```
      //
      //  Examples
      //
      var expdate = new Date ();
      FixCookieDate (expdate); // Correct for Mac date bug (call only once)
      expdate.setTime (expdate.getTime() + (24 * 60 * 60 * 1000)); // 24 hrs
      SetCookie ("ccpath", "http://www.hidaho.com/colorcenter/", expdate);
      SetCookie ("ccname", "hIdaho Design ColorCenter", expdate);
      SetCookie ("tempvar", "This is a temporary cookie.");
      SetCookie ("ubiquitous", "This cookie will work anywhere in this ↩
         domain",null,"/");
      SetCookie ("paranoid", "This cookie requires secure ↩
         communications",expdate,"/",null,true);
      SetCookie ("goner", "This cookie must die!");
      document.write (document.cookie + "<br>");
      DeleteCookie ("goner");
      document.write (document.cookie + "<br>");
      document.write ("ccpath = " + GetCookie("ccpath") + "<br>");
      document.write ("ccname = " + GetCookie("ccname") + "<br>");
      document.write ("tempvar = " + GetCookie("tempvar") + "<br>");
      </script>
   </body>
</html>
```

Extra batches

You may design a site that needs more than 20 cookies for a given domain. For example, in a shopping site, you never know how many items a customer may load into the shopping cart cookie.

Because each named cookie stores plain text, you can create your own text-based data structures to accommodate multiple pieces of information per cookie. (But also watch out for a practical limit of 2,000 characters per name/value pair within the 4,000 character maximum for any domain's combined cookies.) The trick is determining a delimiter character that won't be used by any of the data in the cookie. In Decision Helper (in Chapter 58), for example, we use a period to separate multiple integers stored in a cookie.

With the delimiter character established, you must then write functions that concatenate these "subcookies" into single cookie strings and extract them on the other side. It's a bit more work, but well worth the effort, to have the power of persistent data on the client.

Example

Experiment with the last group of statements in Listing 29-3 to create, retrieve, and delete cookies. You can also experiment with The Evaluator by assigning a name/value pair string to `document.cookie`, and then examining the value of the `cookie` property.

Related Items: `String` object methods (see Chapter 15, "The String Object")

`defaultCharset`

Value: String Read/Write

Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari+, Opera-, Chrome+

The `defaultCharset` property reveals the character set used by the browser to render the current document. You can find possible values for this property at

```
http://www.iana.org/assignments/character-sets
```

Each browser and operating system has its own default character set. Values may also be set through a `<meta>` tag. The difference between the `defaultCharset` and `charset` properties is not clear, especially because both are read/write (although modifying the `defaultCharset` property has no visual effect on the page). However, if your scripts temporarily modify the `charset` property, you can use the `defaultCharset` property to return to the original character set:

```
document.charset = document.defaultCharset;
```

Example

Use The Evaluator (Chapter 4, "JavaScript Essentials") to experiment with the `defaultCharset` property. To see the default setting applied to the page, enter the following statement into the top text box:

```
document.defaultCharset
```

Related Items: `charset`, `characterSet` properties

`defaultView`

Value: `window` or `frame` object reference Read-Only

Compatibility: WinIE-, MacIE-, NN6+, Moz+, Safari+, Opera+, Chrome+

The `defaultView` property returns a reference to the object serving as the viewer for the document. The viewer is responsible for rendering the document, and in Mozilla the object returned in the `defaultView` property is the `window` or `frame` object that contains the document. This W3C DOM Level 2 property provides access to the computed CSS values being applied to any HTML element (through the `document.defaultView.getComputedStyle()` method).

Related Items: `window` and `frame` properties; `window.getComputedStyle()` method

`designMode`

Value: String Read/Write

Compatibility: WinIE5+, MacIE-, NN7.1, Moz1.4+, Safari+, Opera+, Chrome+

The `designMode` property is applicable for IE only when WinIE5+ technology is being used as a component in another application. The property controls whether the browser module is being used for HTML editing. Modifying the property from within a typical HTML page in the IE5+ browser has no effect. But on the Mozilla side, the property can be used to turn an `iframe` element's `document` object into an HTML editable document. Visit `http://www.mozilla.org/editor` for current details and examples.

doctype

Value: `DocumentType` object reference Read-Only

Compatibility: WinIE+, MacIE5+, NN6+, Moz+, Safari+, Opera+, Chrome+

The `doctype` property comes from the W3C Core DOM and returns a `DocumentType` object — a representation of the Document Type Definition (DTD) information for the document. The `DocumentType` object (if one is explicitly defined in the source code) is the first child node of the root document node (and is thus a sibling to the `HTML` element). In IE, this property is only supported in XML documents; HTML documents always return `null`.

Table 29-1 shows the typical `DocumentType` object property list and values for a generic HTML page. Future DOM specifications will allow these properties to be read/write.

Related Items: `Node` object (see Chapter 25, "Document Object Model Essentials")

Example

Take a look at the `document.doctype` object by entering the following line of code in the bottom text field of the Evaluator web page (see Chapter 4):

```
document.doctype
```

TABLE 29-1

DocumentType Object in NN6+/Moz

Property	Value
baseURI	http://www.dannyg.com/index.html
entities	null
internalSubset	null
name	HTML
nodeName	HTML
nodeType	10
notations	null
publicId	-//W3C//DTD XHTML 1.0 Transitional//EN
systemId	http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd

If you pay close attention you'll notice that the `publicId` property is actually set to `-//W3C//DTD HTML 4.01 Transitional//EN`, which is different from the value shown in Table 29-1. This reveals the fact that the Evaluator page declares itself as an HTML 4.01 document.

documentElement

Value: HTML or XML element object reference Read-Only

Compatibility: WinIE5+, MacIE5+, NN6+, Moz+, Safari+, Opera+, Chrome+

The `documentElement` property returns a reference to the HTML (or XML) element object that contains all of the content of the current document. The naming of this property is a bit misleading, because the root document node is not an element, but its only child node is the HTML (or XML) element for the page. At best, you can think of this property as providing scripts with an *element face* to the `document` object and document node associated with the page currently loaded in the browser.

The `document.documentElement` object represents the `html` element for a page, whereas `docoument.body` represents the `body` element. This explains why `document.body` is a child of the `document.documentElement` object.

Example

Use The Evaluator (Chapter 4, "JavaScript Essentials") to examine the behavior of the `documentElement` property. In IE5+/W3C, enter the following statement into the top text field:

```
document.documentElement.tagName
```

The result is `HTML`, as expected.

Related Item: `ownerDocument` property (see Chapter 26, "Generic HTML Element Objects")

documentURI

Value: String Read-Only

Compatibility: WinIE-, MacIE-, NN8+, Moz1.7+, Safari+, Opera+, Chrome+

The `documentURI` property contains the location of the document. This is the W3C DOM Level 3 equivalent of the non-W3C DOM `location.href` property. Use The Evaluator (Chapter 4) to view the document URI by entering the following:

```
document.documentURI
```

Related Item: `document.baseURI` property

domain

Value: String Read/Write

Compatibility: WinIE4+, MacIE4+, NN3+, Moz+, Safari+, Opera+, Chrome+

Security restrictions can get in the way of sites that have more than one server at their domain. Because some objects, especially the `location` object, prevent access to properties of other servers displayed in other frames, legitimate access to those properties is blocked. For example, it's not uncommon for popular sites to have their usual public access site on a server named something such as `www.popular.com`. If a page on that server includes a front end to a site search engine located at `search.popular.com`, visitors who use browsers with these domain restrictions security restrictions are denied access.

To guard against that eventuality, a script in documents from two servers can instruct the browser to think both servers are the same. In the preceding example, you would set the `document.domain` property in both documents to `popular.com`. Without specifically setting the property, the default value includes the server name as well, thus causing a mismatch between hostnames.

Before you start thinking that you can spoof your way into other servers, be aware that you can set the `document.domain` property only to servers with the same domain (following the two-dot rule) as the document doing the setting. Therefore, documents originating only from *xxx*`.popular.com` can set their `document.domain` properties to `popular.com` server.

Related Items: `window.open()` method; `window.location` object; security (see Chapter 49, "Security and Netscape Signed Scripts" on the CD-ROM)

embeds[]

Value: Array of `embed` element objects Read-Only

Compatibility: WinIE4+, MacIE4+, NN3+, Moz+, Safari+, Opera+, Chrome+

Although now supplanted by the `<object>` tag, the `<embed>` tag used to be the markup that loaded data requiring a plug-in application to play or display. The `document.embeds` property is an array of `embed` element objects within the document:

```
var count = document.embeds.length;
```

Related Item: `embed` element object (see Chapter 44, "Embedded Objects")

expando

Value: Boolean Read/Write

Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-, Opera-, Chrome-

Microsoft calls any custom property that is not a native property of the `document` object an *expando* property. By default, most objects in recent generations of browsers allow scripts to add new properties of objects as a way to temporarily store data without explicitly defining global variables. For example, if you want to maintain an independent counter of how often a function is invoked, you can create a custom property of the `document` object and use it as the storage facility:

```
document.counter = 0;
```

IE4+ enables you to control whether the `document` object is capable of accepting `expando` properties. The default value of the `document.expando` property is `true`, thus allowing custom properties. But the potential downside to this permissiveness, especially during the page construction phase, is that a misspelled native property name is gladly accepted by the `document` object. You may not be aware of why the title bar of the browser window doesn't change when you assign a new string to the `document.Title` property (which, in the case-sensitive world of JavaScript, is distinct from the native `document.title` property).

Example

Use The Evaluator (Chapter 4, "JavaScript Essentials") to experiment with the `document.expando` property in IE4+. Begin by proving that the `document` object can normally accept custom properties. Type the following statement into the top text field:

```
document.spooky = "Boo!"
```

This property is now set and stays that way until the page is either reloaded or unloaded.

Now freeze the `document` object's properties with the following statement:

```
document.expando = false
```

If you try to add a new property, such as the following, you receive an error:

```
document.happy = "tra la"
```

Interestingly, even though `document.expando` is turned off, the first custom property is still accessible and modifiable.

Related Item: `prototype` property of custom objects (see Chapter 23, "Function Objects and Custom Objects")

fgColor

(See `alinkColor`)

fileCreatedDate
fileModifiedDate
fileSize

Value: String, Integer (fileSize) Read-Only

Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-, Opera-, Chrome-

These three IE-specific properties return information about the file that holds the current document. The first two properties (not implemented in MacIE) reveal the dates on which the current document's file was created and modified. For an unmodified file, its creation and modified dates are the same. The `fileSize` property reveals the number of bytes of the file.

Date values returned for the first two properties are in a format similar to mm/dd/yyyy. Note, however, that the values contain only the date and not the time. In any case, you can use the values as the parameter to a `new Date()` constructor function. You can then use date calculations for such information as the number of days between the current day and the most recent modification.

Not all servers may provide the proper date or size information about a file, or in a format that IE can interpret. Test your implementation on the deployment server to ensure compatibility.

Also, be aware that these properties can be read-only for a file that is loaded in the browser. JavaScript by itself cannot get this information about files that are on the server, but not loaded into the browser.

Example

Listing 29-4 dynamically generates several pieces of content relating to the creation and modification dates of the file, as well as its size. More importantly, the listing demonstrates how to turn a value returned by the file date properties into a genuine date object that can be used for date calculations. In the case of Listing 29-4, the calculation is the number of full days between the creation date and the day someone views the file. Notice that the dynamically generated content is added very simply through the `innerText` properties of carefully located `span` elements in the body content.

LISTING 29-4

Displaying File Information for a Web Page

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>fileCreatedDate and fileModifiedDate Properties</title>
      <script type="text/javascript">
         function fillInBlanks()
         {
            var created = document.fileCreatedDate;
            var modified = document.fileModifiedDate;
            document.getElementById("created").innerText = created;
            document.getElementById("modified").innerText = modified;
            var createdDate = new Date(created).getTime();
            var today = new Date().getTime();
            var diff = Math.floor((today - createdDate) / (1000*60*60*24));
            document.getElementById("diff").innerText = diff;
            document.getElementById("size").innerText = document.fileSize;
         }

         // bind the event handlers
         function addEvent(elem, evtType, func)
         {
            if (elem.addEventListener)
            {
               elem.addEventListener(evtType, func, false);
            }
            else if (elem.attachEvent)
            {
               elem.attachEvent("on" + evtType, func);
            }
            else
            {
               elem["on" + evtType] = func;
            }
         }
         addEvent(window, "load", function()
         {
            fillInBlanks();
         });
      </script>
   </head>
   <body>
      <h1>fileCreatedDate and fileModifiedDate Properties</h1>
      <hr />
      <p>This file (<span id="size"> </span> bytes) was created on
         <span id="created"> </span> and most recently modified on
         <span id="modified"> </span>.
```

```
      </p>
      <p>It has been <span id="diff"> </span> days since this file was
         created.
      </p>
   </body>
</html>
```

Related Item: `lastModified` property

forms[]

Value: Array Read-Only

Compatibility: WinIE3+, MacIE3+, NN2+, Moz+, Safari+, Opera+, Chrome+

As we show in Chapter 34, "Form and Related Objects," which is dedicated to the `form` object, an HTML form (anything defined inside a `<form>...</form>` tag pair) is a JavaScript object unto itself. You can create a valid reference to a form according to its name (assigned through a form's `name` attribute). For example, if a document contains the following form definition:

```
<form name="phoneData">
   input item definitions
</form>
```

then your scripts can refer to the form object by name:

```
document.phoneData
```

However, a `document` object also tracks its forms in another way: as an array of `form` objects. The first item of a `document.forms` array is the form that loaded first (it was foremost from the top of the HTML code). If your document defines one form, the `forms` property is an array one entry in length; with three separate forms in the document, the array is three entries long.

Use standard array notation to reference a particular form from the `document.forms` array. For example, the first form in a document (the zeroth entry of the `document.forms` array) is referenced as

```
document.forms[0]
```

Any of the form object's properties or methods are available by appending the desired property or method name to the reference. For example, to retrieve the value of an input text field named `homePhone` from the second form of a document, the reference you use is

```
document.forms[1].homePhone.value
```

One advantage to using the `document.forms` property for addressing a form object or element instead of the actual form name is that you may be able to generate a library of generalizable scripts that know how to cycle through all available forms in a document and then hunt for a form that has some special element and property. The following script fragment (part of a *repeat loop* described more

fully in Chapter 21, "Control Structures and Exception Handling") uses a loop-counting variable (i) to help the script check all forms in a document:

```
for (var i = 0; i < document.forms.length; i++) {
   if (document.forms[i]. ... ) {
      statements
   }
}
```

One more variation on forms array references enables you to substitute the name of a form (as a string) for the `forms` array index. For example, the form named `phoneData` can be referenced as:

```
document.forms["phoneData"]
```

If you take a lot of care in assigning names to objects, you will likely prefer the `document.formName` style of referencing forms. In this book, you see both indexed array and form name style references. The advantage of using name references is that even if you redesign the page and change the order of forms in the document, references to the named forms will still be valid, whereas the index numbers of the forms will have changed. Also see the discussion in Chapter 34, "Form and Related Objects," of the `form` object and how to pass a form's data to a function.

Example

The document in Listing 29-5 is set up to display an alert dialog box that simulates navigation to a particular music site, based on the selected status of the blues check box. The user input here is divided into two forms: one form with the check box and the other form with the button that does the navigation. A block of copy fills the space in between. Clicking the bottom button (in the second form) triggers the function that fetches the checked property of the blues check box by using the `document.forms[i]` array as part of the address.

LISTING 29-5

A Simple Form Example

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>document.forms example</title>
      <script type="text/javascript">
         function goMusic()
         {
            if (document.forms[0].bluish.checked)
            {
               alert("Now going to the Blues music area...");
            }
            else
            {
               alert("Now going to the Rock music area...");
            }
         }
```

```
        // bind the event handlers
        function addEvent(elem, evtType, func)
        {
           if (elem.addEventListener)
           {
              elem.addEventListener(evtType, func, false);
           }
           else if (elem.attachEvent)
           {
              elem.attachEvent("on" + evtType, func);
           }
           else
           {
              elem["on" + evtType] = func;
           }
        }
        addEvent(window, "load", function()
        {
           addEvent(document.getElementById("visit"), "click", goMusic);
        });
      </script>
   </head>
   <body>
      <form name="theBlues">
         <input type="checkbox" name="bluish" />Check here if you've got the
         blues.
      </form>
      <hr />
      M<br />
      o<br />
      r<br />
      e<br />
      <br />
      C<br />
      o<br />
      p<br />
      y<br />
      <hr />
      <form name="visit">
         <input type="button" id="visit" value="Visit music site" />
      </form>
   </body>
</html>
```

Related Item: `form` object (see Chapter 34, “Form and Related Objects”)

frames[]

Value: Array Read-Only

Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-, Opera+, Chrome-

The `document.frames` property is similar to the `window.frames` property, but its association with the `document` object may seem a bit illogical at times. The objects contained by the array returned from the property are window objects, which means they are the window objects of any defined `frame` elements (from a framesetting document) or `iframe` elements (from a plain HTML document). Distinguishing the window objects from the `iframe` element objects is important. Window objects have different properties and methods from the `frame` and `iframe` element objects. The latter's properties typically represent the attributes for those element's tags. If a document contains no `iframe` elements, the `document.frames` array length is zero.

Although you can access an individual frame object through the typical array syntax (for example, `document.frames[0]`), you can also use alternate syntax that Microsoft provides for collections of objects. The index number can also be placed inside parentheses, as in:

```
document.frames(0)
```

Moreover, if the frames have values assigned to their `name` attributes, you can use the name (in string form) as a parameter:

```
document.frames("contents")
```

And if the collection of frames has more than one frame with the same name, you must take special care. Using the duplicated name as a parameter forces the reference to return a collection of frame objects that share that name. Or, you can limit the returned value to a single instance of the duplicate-named frames by specifying an optional second parameter indicating the index. For example, if a document has two `iframe` elements with the name `contents`, a script could reference the second `window` object as:

```
document.frames("contents", 1)
```

For the sake of cross-browser compatibility, my preference for referencing frame window objects is through the `window.frames` property.

Example

See Listings 27-7 and 27-8 for examples of using the `frames` property with window objects.

Related Item: `window.frames` property

height
width

Value: Integer — Read-Only

Compatibility: WinIE-, MacIE-, NN4+, Moz+, Safari+, Opera-, Chrome+

The `height` and `width` properties provide the pixel dimensions of the content within the current window (or frame). If the document's content is smaller than the size of the browser's content region, the dimensions returned by these properties include the blank space to the right or bottom edges of the content area of the window. But if the content extends beyond the viewable edges of the content region, the dimensions include the unseen content as well. The corresponding measures in Internet Explorer are the `document.body.scrollHeight` and `document.body.scrollWidth` properties. These IE properties are also supported by most modern browsers.

Example

Use The Evaluator (Chapter 4, "JavaScript Essentials") to examine the `height` and `width` properties of that document. Enter the following statement into the top text box and click the Evaluate button:

```
"height=" + document.height + "; width=" + document.width
```

Resize the window so that you see both vertical and horizontal scroll bars in the browser window and click the Evaluate button again. If either or both numbers get smaller, the values in the Results box are the exact size of the space occupied by the document. But if you expand the window to well beyond where the scroll bars are needed, the values extend to the number of pixels in each dimension of the window's content region.

Related Items: `document.body.scrollHeight`, `document.body.scrollWidth` properties

images[]

Value: Array Read-Only

Compatibility: WinIE4+, MacIE3+, NN3+, Moz+, Safari+, Opera+, Chrome+

With images treated as first-class objects, beginning with NN3 and IE4, it's only natural for a document to maintain an array of all the image tags defined on the page (just as it does for links and anchors). The primary importance of having images as objects is that you can modify their content (the source file associated with the rectangular space of the image) on-the-fly. You can find details about the `image` object in Chapter 31, "Image, Area, Map, and Canvas Objects."

Use image array references to pinpoint a specific image for the retrieval of any image property, or for assigning a new image file to its `src` property. Image arrays begin their index counts with `0`: The first image in a document has the reference `document.images[0]`. And, as with any array object, you can find out how many images the array contains by checking the `length` property. For example:

```
var imageCount = document.images.length;
```

Images can also have names, so if you prefer, you can refer to the image object by its name, as in

```
var imageLoaded = document.imageName.complete;
```

or

```
var imageLoaded = document.images[imageName].complete;
```

The `document.images` array is a useful guide to knowing whether a browser supports swappable images. Any browser that treats an `img` element as an object always forms a `document.images` array in the page. If no images are defined in the page, the array is still there, but its length is zero. The array's existence, however, is the clue about image object compatibility. Because the `document.images` array evaluates to an array object when present, the expression can be used as a conditional expression for branching to statements that involve image swapping:

```
if (document.images) {
   // image swapping or precaching here
}
```

Browsers that don't have this property (legacy and, potentially, mobile) evaluate `document.images` as `undefined`, and thus the condition is treated as a `false` value.

Example

The `document.images` property is defined automatically as the browser builds the object model for a document that contains image objects. See the discussion about the `Image` object in Chapter 31, for reference examples.

Related Item: `Image` object (see Chapter 31, "Image, Area, Map, and Canvas Objects")

`implementation`

Value: Object Read-Only

Compatibility: WinIE6+, MacIE5+, NN6+, Moz+, Safari+, Opera+, Chrome+

The Core W3C DOM defines the `document.implementation` property as an avenue to let scripts find out what DOM features (that is, modules of the DOM standard) are implemented for the current environment. Although the object returned by the property (a `DOMImplementation` object) has no properties, it has a method, `hasFeature()`, which lets scripts find out, for example, whether the environment supports HTML or just XML. The first parameter of the `hasFeature()` method is the feature in the form of a string. The second parameter is a string form of the version number. The method returns a Boolean value.

The "Conformance" section of the W3C DOM specification governs the module names. (The standard also allows browser-specific features to be tested through the `hasFeature()` method.) Module names include strings such as `HTML`, `XML`, `MouseEvents`, and so on.

Version numbering for W3C DOM modules corresponds to the W3C DOM level. Thus, the version for the XML DOM module in DOM Level 2 is known as `2.0`. Note that versions refer to DOM modules and not, for example, the separate HTML standard.

Example

Use The Evaluator (Chapter 4, "JavaScript Essentials") to experiment with the `document.implementation.hasFeature()` method. Enter the following statements one at a time into the top text field and examine the results:

```
document.implementation.hasFeature("HTML","1.0")
document.implementation.hasFeature("HTML","2.0")
document.implementation.hasFeature("HTML","3.0")
document.implementation.hasFeature("CSS","2.0")
document.implementation.hasFeature("CSS2","2.0")
```

Feel free to try other values. For some reason, as of version 7, Internet Explorer returns false for some features that it indeed supports, such as CSS 2.0. In other words, it's probably not a good idea to place a lot of trust in the IE results of the `hasFeature()` method, at least for the time being.

`inputEncoding`

Value: String Read-Only

Compatibility: WinIE-, MacIE-, NN-, Moz1.8+, Safari+, Opera-, Chrome+

The input encoding of a document is the character encoding that is in effect at the time when the document is parsed. For example, ISO-8859-1 is a common character encoding that you may see reported by the `inputEncoding` property.

lastModified

Value: Date string Read-Only

Compatibility: WinIE3+, MacIE3+, NN2+, Moz+, Safari+, Opera+, Chrome+

Every disk file maintains a modified time stamp, and most (but not all) servers are configured to expose this information to a browser accessing a file. This information is available by reading the `document.lastModified` property. If your server supplies this information to the client, you can use the value of this property to present this information for readers of your web page. The script automatically updates the value for you, rather than requiring you to hand-code the HTML line every time you modify the home page. Be aware that if you are testing with Safari or Google Chrome, your code will not work unless the files are coming from an HTTP server.

If the value returned to you displays itself as a date in 1969, it means that you are positioned somewhere west of GMT, or Greenwich Mean Time (some number of time zones west of GMT at 1 January 1970), and the server is not providing the proper data when it serves the file. Sometimes server configuration can fix the problem, but not always.

The returned value is not a date object, but rather a straight string consisting of the time and date, as recorded by the document's file system. The format of the string varies from browser to browser and version to version. You can, however, usually convert the date string to a JavaScript date object and use the date object's methods to extract selected elements for recompilation into readable form. Listing 29-6 shows an example.

Even local file systems don't necessarily provide the correct data for every browser to interpret. But put that same file on a UNIX or Windows web server, and the date appears correctly when accessed through the Net.

Example

Experiment with the `document.lastModified` property with Listing 29-6. But also be prepared for inaccurate readings depending on the location of the file.

LISTING 29-6

Putting a Time Stamp on a Page

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Time Stamper</title>
   </head>
   <body>
      <center>
         <h1>GiantCo Home Page</h1>
      </center>
      <script type="text/javascript">
         update = new Date(document.lastModified);
         theMonth = update.getMonth() + 1;
         theDate = update.getDate();
```

continued

LISTING 29-6 *(continued)*

```
         theYear = update.getFullYear();
         document.writeln("<I>Last updated:" +
                          theMonth + "/" + theDate +
                          "/" + theYear + "<\/I>");
      </script>
      <hr />
   </body>
</html>
```

As noted at great length in the `Date` object discussion in Chapter 17, you should be aware that date formats vary greatly from country to country. Some of these formats use a different order for date elements. When you hard-code a date format, it may take a form that is unfamiliar to other users of your page.

Related Item: `Date` object (see Chapter 17, "The Date Object")

layers[]

Value: Array Read-Only

Compatibility: WinIE-, MacIE-, NN4, Moz-, Safari-, Opera-, Chrome-

The `layer` object is the NN4 way of exposing positioned elements to the object model. Thus, the `document.layers` property is an array of positioned elements in the document. The `layer` object and `document.layers` property are orphaned in NN4, and their importance is all but gone now that Mozilla has taken over. Chapter 43 includes several examples of how to carry out similar functionality as the `document.layers` property using the standard W3C DOM.

Related Item: `layer` object (see Chapter 43, "Positioned Objects")

linkColor

(See `alinkColor`)

links[]

Value: Array Read-Only

Compatibility: WinIE3+, MacIE3+, NN2+, Moz+, Safari+, Opera+, Chrome+

The `document.links` property is similar to the `document.anchors` property, except that the objects maintained by the array are link objects — items created with `<a href="">` tags. Use the array references to pinpoint a specific link for retrieving any link property, such as the target window specified in the link's HTML definition.

Link arrays begin their index counts with `0`: The first link in a document has the reference `document.links[0]`. And, as with any array object, you can find out how many entries the array has by checking the `length` property. For example:

```
var linkCount = document.links.length;
```

Entries in the `document.links` property are full-fledged `location` objects, which means you have the same properties available to each member of the `links[]` array as you do in the `location` object.

Example

The `document.links` property is defined automatically as the browser builds the object model for a document that contains link objects. You rarely access this property, except to determine the number of link objects in the document.

Related Items: `link` object; `document.anchors` property

location
URL

Value: String — Read/Write and Read-Only (see text)

Compatibility: WinIE4+, MacIE4+, NN3+, Moz+, Safari+, Opera+, Chrome+

The `document.URL` property is similar to the `window.location` property. A `location` object, you may recall from Chapter 28, "Location and History Objects," consists of a number of properties regarding the document currently loaded in a window or frame. Assigning a new URL to the `location` object (or `location.href` property) tells the browser to load the page from that URL into the frame. The `document.URL` property, on the other hand, is simply a string (read-only in Navigator, Mozilla, and Safari) that reveals the URL of the current document. The value may be important to your script, but the property does not have the object power of the `window.location` object. You cannot change (assign another value to) this property value because a document has only one URL: its location on the Net (or your hard disk) where the file exists, and what protocol is required to get it.

This may seem like a fine distinction, and it is. The reference you use (`window.location` object or `document.URL` property) depends on what you are trying to accomplish specifically with the script. If the script is changing the content of a window by loading a new URL, you have no choice but to assign a value to the `window.location` object. Similarly, if the script is concerned with the component parts of a URL, the properties of the `location` object provide the simplest avenue to that information. To retrieve the URL of a document in string form (whether it is in the current window or in another frame), you can use either the `document.URL` property or the `window.location.href` property.

Note

The `document.URL` property replaces the old `document.location` property, which is still supported in most browsers. ■

Example

HTML documents in Listings 29-7 through 29-9 create a test lab that enables you to experiment with viewing the `document.URL` property for different windows and frames in a multiframe environment. Results are displayed in a table, with an additional listing of the `document.title` property to help you identify documents being referred to. The same security restrictions that apply to retrieving `window.location` object properties also apply to retrieving the `document.URL` property from another window or frame.

LISTING 29-7

A Simple Frameset for the URL Example

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>document.URL Reader</title>
   </head>
   <frameset rows="60%,40%">
      <frame name="Frame1" src="jsb29-09.html" />
      <frame name="Frame2" src="jsb29-08.html" />
   </frameset>
</html>
```

LISTING 29-8

Showing Location Information for Different Contexts

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>URL Property Reader</title>
      <script type="text/javascript">
         function fillTopFrame()
         {
            newURL=prompt("Enter the URL of a document to show in the top frame:","");
            if (newURL != null && newURL != "")
            {
               top.frames[0].location = newURL;
            }
         }

         function showLoc(item)
         {
            var windName = item.value;
            var theRef = windName + ".document";
            item.form.dLoc.value = decodeURIComponent(eval(theRef + ".URL"));
            item.form.dTitle.value = decodeURIComponent(eval(theRef + ".title"));
         }

         // bind the event handlers
         function addEvent(elem, evtType, func)
         {
            if (elem.addEventListener)
            {
```

```
               elem.addEventListener(evtType, func, false);
            }
            else if (elem.attachEvent)
            {
               elem.attachEvent("on" + evtType, func);
            }
            else
            {
               elem["on" + evtType] = func;
            }
         }
         addEvent(window, "load", function()
         {
            addEvent(document.getElementById("opener"), "click", fillTopFrame);
            addEvent(document.getElementById("parent"), "click",
               function(evt) {showLoc(document.getElementById("parent"));});
            addEvent(document.getElementById("upper"), "click",
               function(evt) {showLoc(document.getElementById("upper"));});
            addEvent(document.getElementById("this"), "click",
               function(evt) {showLoc(document.getElementById("this"));});
         });
      </script>
   </head>
   <body>
      Click the "Open URL" button to enter the location of an HTML document to
      display in the upper frame of this window.
      <form>
         <input type="button" id="opener" name="opener" value="Open URL..." />
      </form>
      <hr />
      <form>
         Select a window or frame to view each document property values.
         <p>
            <input type="radio" id="parent" name="whichFrame" value="parent" />
            Parent window
            <input type="radio" name="whichFrame" id="upper"
               value="top.frames[0]" />
            Upper frame
            <input type="radio" name="whichFrame" id="this" value="top.frames[1]" />
            This frame
         </p>
         <table border="2">
            <tr>
               <td align="right">document.URL:</td>
               <td><textarea name="dLoc" rows="3" cols="30" wrap="soft">
                   </textarea>
               </td>
            </tr>
            <tr>
               <td align="right">document.title:</td>
               <td><textarea name="dTitle" rows="3" cols="30" wrap="soft">
                   </textarea>
```

continued

LISTING 29-8 *(continued)*

```
          </td>
        </tr>
      </table>
    </form>
  </body>
</html>
```

LISTING 29-9

A Placeholder Page for the URL Example

```
<!DOCTYPE html>
<html>
  <head>
    <meta http-equiv="content-type" content="text/html;charset=utf-8">
    <title>Opening Placeholder</title>
  </head>
  <body>
    Initial place holder. Experiment with other URLs for this frame (see
    below).
  </body>
</html>
```

Related Items: `location` object; `location.href`, `URLUnencoded` properties

media

Value: String — Read/Write

Compatibility: WinIE5.5+, MacIE-, NN-, Moz-, Safari-, Opera-, Chrome-

The `document.media` property indicates the output medium for which content is formatted. The property actually returns an empty string as of IE7, but the intention appears to be to provide a way to use scripting to set the equivalent of the CSS2 `@media` rule (one of the so-called *at* rules because of the at symbol). This style sheet rule allows browsers to assign separate styles for each type of output device on which the page is rendered (for example, perhaps a different font for a printer versus the screen). In practice, however, this property is not modifiable, at least through IE8.

Related Items: None

mimeType

Value: String — Read-Only

Compatibility: WinIE5+, MacIE-, NN-, Moz-, Safari-, Opera-, Chrome-

Although this property is readable in WinIE5+, its value is not, strictly speaking, a MIME type, at least not in traditional MIME format. Moreover, the results are inconsistent between IE versions 5, 6,

7, and 8. Perhaps this property will be of more use in an XML, rather than an HTML, document environment. In any case, this property in no way exposes supported MIME types in the current browser.

namePropPlaceholder

Value: String Read-Only

Compatibility: WinIE6+, MacIE-, NN-, Moz-, Safari-, Opera-, Chrome-

The `nameProp` property returns a string containing the title of the document, which is the same as `document.title`. If the document doesn't have a title, `nameProp` contains an empty string.

Related Item: `title` property

namespaces[]

Value: Array of `namespace` objects Read-Only

Compatibility: WinIE5.5+, MacIE-, NN-, Moz-, Safari-, Opera-, Chrome-

A `namespace` object can dynamically import an XML-based IE Element Behavior. The `namespaces` property returns an array of all `namespace` objects defined in the current document.

Related Items: None

parentWindow

Value: `window` object reference Read-Only

Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-, Opera+, Chrome-

The `document.parentWindow` property returns a reference to the `window` object containing the current document. The value is the same as any reference to the current window.

Example

To prove that the `parentWindow` property points to the document's window, you can enter the following statement into the top text field of The Evaluator (see Chapter 4, "JavaScript Essentials"):

```
document.parentWindow == self
```

This expression evaluates to `true` only if both references are of the same object.

Related Item: `window` object

plugins[]

Value: Array Read-Only

Compatibility: WinIE4+, MacIE4+, NN4+, Moz+, Safari+, Opera+, Chrome+

The `document.plugins` property returns the same array of `embed` element objects that you get from the `document.embeds` property. This property has been deprecated in favor of `document.embeds`.

Related Item: `document.embeds` property

protocol

Value: String Read/Write

Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-, Opera-, Chrome-

The IE-specific `document.protocol` property returns the plain-language version of the protocol that was used to access the current document. For example, if the file is accessed from a web server, the property returns `Hypertext Transfer Protocol`. This property differs from the `location.protocol` property, which returns the portion of the URL that includes the often more cryptic protocol abbreviation (for example, `http:`). As a general rule, you want to hide all of this stuff from a web application user.

Example

If you use The Evaluator (Chapter 4, "JavaScript Essentials") to test the `document.protocol` property, you will find that it displays `File Protocol` in the results because you are accessing the listing from a local hard disk or CD-ROM. However, if you upload the Evaluator web page to a web server and access it from there, you will see the expected `Hypertext Transfer Protocol` result.

Related Item: `location.protocol` property

referrer

Value: String Read-Only

Compatibility: WinIE3+, MacIE3+, NN2+, Moz+, Safari+, Opera+, Chrome+

When a link from one document leads to another, the second document can, under JavaScript control, reveal the URL of the document containing the link. The `document.referrer` property contains a string of that URL. This feature can be a useful tool for customizing the content of pages based on the previous location the user was visiting within your site. A referrer contains a value only when the user reaches the current page through a link. Any other method of navigation (such as through the history, bookmarks, or by manually entering a URL) sets this property to an empty string.

Caution

The `document.referrer` property usually returns an empty string unless the files are retrieved from a web server. ■

Example

This demonstration requires two documents (and you'll need to access the documents from a web server). The first document, in Listing 29-10, simply contains one line of text as a link to the second document. In the second document, shown in Listing 29-11, a script verifies the document from which the user came through a link. If the script knows about that link, it displays a message relevant to the experience the user had at the first document. Also try opening Listing 29-11 in a new browser window from the Open File command in the File menu to see how the script won't recognize the referrer.

LISTING 29-10

An Example Referrer Page

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>document.referrer Property 1</title>
   </head>
   <body>
      <h1><a href="jsb29-11.html">Visit my sister document</a></h1>
   </body>
</html>
```

LISTING 29-11

Determining the Referrer when a Page Is Visited Through a Link

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>document.referrer Property 2</title>
   </head>
   <body>
      <h1>
         <script type="text/javascript">
            alert(document.referrer.length + "    :    " + document.referrer);
               if (document.referrer.length > 0 &&
                   document.referrer.indexOf("jsb29-10.html") != -1)
               {
                  document.write("How is my brother document?");
               }
               else
               {
                  document.write("Hello, and thank you for stopping by.");
            }
         </script>
      </h1>
   </body>
</html>
```

Related Item: `link` object

`scripts[]`

Value: Array　　　　Read-Only

Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari+, Opera+, Chrome+

Originally IE-specific, the `document.scripts` property returns an array of all `script` element objects in the current document. You can reference an individual `script` element object to read not only the properties it shares with all HTML element objects (see Chapter 26, "Generic HTML Element Objects"), but also script-specific properties, such as `defer`, `src`, and `htmlFor`. The actual scripting is accessible through either the `innerText` or `text` properties for any `script` element object.

Although the `document.scripts` array is read-only, many properties of individual `script` element objects are modifiable. Adding or removing `script` elements impacts the length of the `document.scripts` array. Don't forget, too, that if your scripts need to access a specific `script` element object, you can assign an `id` attribute to it and reference the element directly.

This property is the same as the W3C browser expression `document .getElementsByTagName("script")`, which returns an array of the same objects.

Example

You can experiment with the `document.scripts` array in The Evaluator (Chapter 4, "JavaScript Essentials"). For example, you can see that only one `script` element object is in the Evaluator page if you enter the following statement into the top text field:

```
document.scripts.length
```

If you want to view all of the properties of that lone `script` element object, enter the following statement into the bottom text field:

```
document.scripts[0]
```

Among the properties are both `innerText` and `text`. If you assign an empty string to either property, the scripts are wiped out from the object model, but not from the browser. The scripts disappear because after they were loaded, they were cached outside of the object model. Therefore, if you enter the following statement into the top field:

```
document.scripts[0].text = ""
```

the script contents are gone from the object model, yet subsequent clicks of the Evaluate and List Properties buttons (which invoke functions of the `script` element object) still work.

Related Item: `script` element object (Chapter 40, "HTML Directive Objects")

security

Value: String Read-Only

Compatibility: WinIE5.5+, MacIE-, NN-, Moz-, Safari-, Opera-, Chrome-

The `security` property reveals information about a security certificate, if one is associated with the current document.

Related Items: none

selection

Value: Object Read-Only

Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-, Opera+, Chrome-

The `document.selection` property returns a `selection` object whose content is represented in the browser window as a body text selection. That selection can be explicitly performed by the

user (by clicking and dragging across some text) or created under script control through the WinIE `TextRange` object (see Chapter 33). Because script action on a selection (for example, finding the next instance of selected text) is performed through the `TextRange` object, converting a selection to a `TextRange` object using the `document.selection.createRange()` method is common practice. See the `selection` object in Chapter 33 for more details.

Be aware that you cannot script interaction with text selections through user interface elements, such as buttons. Clicking a button gives focus to the button and deselects the selection. Use other events, such as `document.onmouseup` to trigger actions on a selection.

Example

Refer to Listings 26-36 and 26-44 in Chapter 26 to see the `document.selection` property in action for script-controlled copying and pasting (WinIE only).

Related Items: `selection`, `TextRange` objects

strictErrorChecking

Value: String Read-Only

Compatibility: WinIE-, MacIE-, NN-, Moz1.8+, Safari-, Opera-, Chrome-

The `strictErrorChecking` property reveals the error-checking mode for the document. More specifically, if the property is set to `true` (the default), exceptions and errors related to DOM operations are reported. Otherwise, DOM-related exceptions may not be thrown, and errors may not be reported.

styleSheets[]

Value: Array Read-Only

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

The `document.styleSheets` array consists of references to all `style` element objects in the document. Not included in this array are style sheets that are assigned to elements by way of the `style` attribute inside a tag, or linked in through `link` elements. See Chapter 38 for details about the `styleSheet` object.

Related Item: `styleSheet` object (see Chapter 38, "Style Sheet and Style Objects")

title

Value: String Read-Only and Read/Write

Compatibility: WinIE3+, MacIE3+, NN2+, Moz+, Safari+, Opera+, Chrome+

A document's title is the text that appears between the `<title>...</title>` tag pair in an HTML document's `head` portion. The title usually appears in the title bar of the browser window in a single-frame presentation, or in a tabbed pane within a multi-paned browser window. Only the title of the topmost framesetting document appears as the title of a multiframe window. Even so, the `title` property for an individual document within a frame is available through scripting. For example, if two frames are available (`UpperFrame` and `LowerFrame`), a script in the document occupying the `LowerFrame` frame can reference the `title` property of the other frame's document, such as this:

```
parent.UpperFrame.document.title
```

The `document.title` property is a holdover from the original document object model. HTML elements in recent browsers have an entirely different application of the `title` property (see Chapter 26). In modern browsers (IE4+/W3C/Moz/Safari/Opera/Chrome), you should address the document's title by way of the `title` element object directly.

Related Items: `history` object

URL

(See `location`)

URLUnencoded

Value: String Read-Only

Compatibility: WinIE5.5+, MacIE-, NN-, Moz+, Safari+, Opera+, Chrome+

The `document.URL` property returns a URL-encoded string, meaning that non-alphanumeric characters in the URL are converted to URL-friendly characters (for example, a space becomes `%20`). You can always use the `decodeURI()` function on the value returned by the `document.URL` property, although in IE the `URLUnencoded` property does that for you. If there are no URL-encoded characters in the URL, then both properties return identical strings.

Related Items: `document.URL` property

vlinkColor

(See `alinkColor`)

width

(See `height`)

xmlEncoding
xmlStandalone
xmlVersion

Value: String Read-Only

Compatibility: WinIE-, MacIE-, NN-, Moz1.8+, Safari+, Opera-, Chrome+

These three properties reveal information about the document as it pertains to XML. More specifically, they convey the XML encoding of the document, whether or not the document is a standalone XML document, and the XML version number of the document, respectively. If any of the property values cannot be determined, their values remain `null`.

Methods

captureEvents(*eventTypeList*)

Returns: Nothing

Compatibility: WinIE-, MacIE-, NN4+, Moz-, Safari-, Opera-, Chrome-

In Navigator 4 only, the natural propagation of an event is downward from the `window` object, through the `document` object, and eventually reaching its target. For example, if you click a button, the `click` event first reaches the `window` object; then it goes to the `document` object; if the button is defined within a layer, the event also filters through that layer; eventually (in a split second) the event reaches the button, where an `onclick` event handler is ready to act on that click.

Event capture with different syntax has been standardized in the W3C DOM and is implemented in W3C browsers, such as Firefox and Camino (Mozilla). More specifically, the W3C event capture model introduces the concept of an event listener, which enables you to bind an event handler function to an event. See the `addEventListener()` method in Chapter 26 for the W3C counterpart to the NN4 `captureEvents()` method. Also, see Chapter 32 for more details on the combination of event capture and event bubbling in the W3C DOM.

clear()

Returns: Nothing

Compatibility: WinIE3+, MacIE3+, NN2+, Moz+, Safari+, Opera+, Chrome+

Ever since NN2, the `document.clear()` method was intended to clear the current document from the browser window. This method is quite impractical, because you typically need some further scripts to execute after you clear the document, but if the scripts are gone, nothing else happens.

In practice, the `document.clear()` method never did what it was supposed to do (and in earlier browsers easily caused browser crashes). We strongly recommend against using `document.clear()`, even in preparation for generating a new page's content with `document.write()`. The `document.write()` method clears the original document from the window before adding new content. If you truly want to empty a window or frame, then use `document.write()` to write a blank HTML document or to load an empty HTML document from the server.

Related Items: `document.close()`, `document.write()`, `document.writeln()` methods

close()

Returns: Nothing

Compatibility: WinIE3+, MacIE3+, NN2+, Moz+, Safari+, Opera+, Chrome+

Whenever a layout stream is opened to a window through the `document.open()` method or through either of the document writing methods (which also open the layout stream), you must close the stream after the document is written. This causes the `Layout:Complete` and `Done` messages to appear in the status line (although you may experience some bugs in the status message on some platforms). The document-closing step is very important to prepare the window for the next potential round of replenishment with new script-assembled HTML. If you don't close the document, subsequent writing is appended to the bottom of the document.

Some or all of the data specified for the window won't display properly until you invoke the `document.close()` method, especially when images are being drawn as part of the document stream. A common symptom is the momentary appearance and then disappearance of the document parts. If you see such behavior, look for a missing `document.close()` method after the last `document.write()` method.

Fixing the Sticky Wait Cursor

From time to time, various browsers fail to restore the cursor to normal after `document.write()` and `document.close()` (and some other content-modifying scripts). The cursor stubbornly remains in the wait mode, or the progress bar keeps spinning when, in truth, all processing has been completed. Albeit ugly, one workaround that we have found effective is to force an extra `document.close()` via a `javascript:` pseudo-URL (just adding another `document.close()` to your script doesn't do the trick). For use within a frameset, the `javascript:` URL must be directed to the top of the frameset hierarchy, whereas the `document.close()` method is aimed at the frame that had its content changed. For example, if the change is made to a frame named `content`, create a function, such as the following:

```
function recloseDoc() {
   top.location.href =
      "javascript:void (parent.content.document.close())";
}
```

If you place this function in the framesetting document, scripts that modify the content frame can invoke this script after any operation that prevents the normal cursor from appearing.

Example

Before you experiment with the `document.close()` method, be sure you understand the `document.write()` method described later in this chapter. After that, make a separate set of the three documents for that method's example (Listings 29-14 through 29-16 in a different directory or folder). In the `takePulse()` function listing, comment out the `document.close()` statement, as shown here:

```
msg += "<p>Make it a great day!</body></html>";
parent.frames[1].document.write(msg);
//parent.frames[1].document.close();
```

Now try the pages on your browser. You see that each click of the upper button appends text to the bottom frame, without first removing the previous text. The reason is that the previous layout stream was never closed. The document thinks that you're still writing to it. Also, without properly closing the stream, the last line of text may not appear in the most recently written batch.

Related Items: `document.open()`, `document.clear()`, `document.write()`, `document.writeln()` methods

createAttribute("*attributeName*")

Returns: Attribute object reference

Compatibility: WinIE6+, MacIE5+, NN6+, Moz+, Safari+, Opera+, Chrome+

The `document.createAttribute()` method generates an attribute node object (formally known as an `Attr` object in W3C DOM terminology) and returns a reference to the newly created object. Invoking the method assigns only the name of the attribute, so it is up to your script to assign a value to the object's `nodeValue` property and then plug the new attribute into an existing element through

that element's `setAttributeNode()` method (described in Chapter 26). The following sequence generates an attribute that becomes an attribute of a `table` element:

```
var newAttr = document.createAttribute("width");
newAttr.nodeValue = "80%";
document.getElementById("myTable").setAttributeNode(newAttr);
```

Attributes do not always have to be attributes known to the HTML standard, because the method also works for XML elements, which have custom attributes.

Example

To create an attribute and inspect its properties, enter the following text into the top text box of The Evaluator (Chapter 4, "JavaScript Essentials"):

```
a = document.createAttribute("author")
```

Now enter `a` into the bottom text box to inspect the properties of an `Attr` object.

Related Items: `setAttributeNode()` method (see Chapter 26, "Generic HTML Element Objects")

createCDATASection("*data*")

Returns: CDATA section object reference

Compatibility: WinIE-, MacIE5, NN7+, Moz+, Safari+, Opera+, Chrome+

The `document.createCDATASection()` method generates a CDATA section node for whatever string you pass as the parameter. The value of the new node becomes the string that you pass.

createComment("*commentText*")

Returns: Comment object reference

Compatibility: WinIE6+, MacIE5+, NN6+, Moz+, Safari+, Opera+, Chrome+

The `document.createComment()` method creates an instance of a comment node. Upon creation, the node is in memory and available to be inserted into the document via any node's `appendChild()` or `insertBefore()` method.

Related Items: `appendChild()`, `insertBefore()` methods

createDocumentFragment()

Returns: Document fragment object reference

Compatibility: WinIE6+, MacIE5+, NN6+, Moz+, Safari+, Opera+, Chrome+

The `document.createDocumentFragment()` method creates an instance of an empty document fragment node. This node serves as a holder that can be used to assemble a sequence of nodes in memory. After creating and assembling nodes into the document fragment, the entire fragment can be inserted into the document tree.

A document fragment is particularly helpful when your scripts assemble an arbitrary sequence of element and text nodes. By providing a parent node for all content within, the fragment node supplies the necessary parent node context for W3C DOM node methods, such as `appendChild()`, during the content assembly process. If you then append or insert the document fragment node to an

element in the rendered document tree, the fragment wrapper disappears, leaving just its content in the desired location in the document. Therefore, a typical usage pattern for a document fragment is to begin by creating an empty fragment node (through the `createDocumentFragment()` method), populate it at will with newly created element or text nodes or both, and then use the appropriate node method on a document tree's element to append, insert, or replace, using the fragment node as the source material.

Related Items: None

`createElement("tagName")`
`createElementNS("namespaceURI", "`*`tagName`*`")`

Returns: Element object reference

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

The `document.createElement()` and `document.createElementNS()` methods generate an element object for whatever HTML (or XML) tag name you pass as the parameter. An object created in this manner is not officially part of the current document node tree because it has not yet been placed into the document. But these methods are the way you begin assembling an element object that eventually gets inserted into the document. The `createElementNS()` method is identical to `createElement()`, except that the latter method accepts an extra parameter that you use to pass a namespace URI for the element. Additionally, the tag name that you specify when creating an element via `createElementNS()` must be a qualified name. Note, however, that `createElementNS()` is not supported in Internet Explorer through version 8.

The returned value is a reference to the object. Properties of that object include all properties (set to default values) that the browser's object model defines for that element object. Your scripts can then address the object through this reference to set the object's properties. Typically, you do this before the object is inserted into the document, especially because otherwise read-only properties can be modified before the element is inserted into the document.

After the object is inserted into the document, the original reference (for example, a global variable used to store the value returned from the `createElement()` method) still points to the object, even while it is in the document and being displayed for the user. To demonstrate this effect, consider the following statements that create a simple paragraph element containing a text node:

```
var newText = document.createTextNode("Four score and seven years ago...");
var newElem = document.createElement("p");
newElem.id = "newestP";
newElem.appendChild(newText);
document.body.appendChild(newElem);
```

At this point, the new paragraph is visible in the document. But you can now modify, for example, the style of the paragraph by addressing either the element in the `document` object model or the variable that holds the reference to the object you created:

```
newElem.style.fontSize = "20pt";
```

or

```
document.getElementById("newestP").style.fontSize = "20pt";
```

The two references are inextricably connected and always point to the exact same object. Therefore, if you want to use a script to generate a series of similar elements (for example, a bunch of `li` list item

elements), then you can use `createElement()` to make the first one and set all properties that the items have in common. Then use `cloneNode()` to make a new copy, which you can then treat as a separate element (and probably assign unique IDs to each element).

When scripting in the W3C DOM environment, you may rely on `document.createElement()` frequently to generate new content for a page or portion thereof (unless you prefer to use the convenience `innerHTML` property to add content in the form of strings of HTML). In a strict W3C DOM environment, creating new elements is not a matter of assembling HTML strings, but rather creating genuine element (and text node) objects.

Example

Chapter 26, "Generic HTML Element Objects," contains numerous examples of the `document.createElement()` method in concert with methods that add or replace content to a document. See Listings 26-10, 26-21, 26-22, 26-28, 26-29, and 26-31.

Related Item: `document.createTextNode()` method

createEvent("*eventType*")

Returns: Event object reference

Compatibility: WinIE-, MacIE-, NN6+, Moz+, Safari+, Opera+, Chrome+

The `document.createEvent()` method creates an instance of a W3C DOM `Event` object of the specified event category. Upon creation, the generic event must be initialized as a particular event type, and any other relevant properties set for the event. After successfully initializing the event, you can fire it through a call to the `dispatchEvent()` method.

Event types recognized by Mozilla are `KeyEvents`, `MouseEvents`, `MutationEvents`, and `UIEvents`. Beginning with Mozilla 1.7.5, the following additional types may also be used: `Event`, `KeyboardEvent`, `MouseEvent`, `MutationEvent`, `MutationNameEvent`, `TextEvent`, `UIEvent`. The process of initializing each of these event types requires its own series of parameters in the associated `initEvent()` method. See Chapter 32 for more details.

Example

Following is an example of how you might create an event, initialize it to a specific event type, and send it to a given element:

```
var evt = document.createEvent("MouseEvents");
evt.initEvent("mouseup", true, true);
document.getElementById("myButton").dispatchEvent(evt);
```

Related Items: `createEventObject()` method; W3C DOM event object (see Chapter 32, "Event Objects")

createEventObject([*eventObject*])

Returns: `event` object

Compatibility: WinIE5.5+, MacIE-, NN-, Moz-, Safari-, Opera-, Chrome-

The IE-specific `createEventObject()` method creates an `event` object, which can then be passed as a parameter to the `fireEvent()` method of any element object. The `event` object created by this event is just like an `event` object created by a user or system action.

An optional parameter enables you to base the new event on an existing `event` object. In other words, the properties of the newly created `event` object pick up all the properties of the `event` object passed as a parameter, which then enables you to modify properties of your choice. If you provide no parameter to the method, then you must fill the essential properties manually. For more about the properties of an `event` object, see Chapter 32.

Example

See the discussion of the `fireEvent()` method in Chapter 26 for an example of the sequence to follow when creating an event to fire on an element.

Related Items: `createEvent()` method; `fireEvent()` method (see Chapter 26); `event` object (see Chapter 32)

createNSResolver(*nodeResolver*)

Returns: XPath namespace resolver object reference

Compatibility: WinIE-, MacIE-, NN7+, Moz+, Safari+, Opera+, Chrome+

The `createNSResolver()` method is used in the context of XPath to alter a node so that it can resolve namespaces. This is necessary as part of the evaluation of an XPath expression. The only parameter is the node that is to serve as the basis for the namespace resolver.

Related Item: `evaluate()` method

createRange()

Returns: `Range` object reference

Compatibility: WinIE-, MacIE-, NN6+, Moz+, Safari+, Opera+, Chrome+

The `document.createRange()` method creates an empty W3C DOM `Range` object with the boundary points of the range collapsed to the point before the first character of the rendered body text.

Related Item: `Range` object

createStyleSheet(["*URL*"[, *index*]])

Returns: `styleSheet` object reference

Compatibility: WinIE4+, MacIE4, NN-, Moz-, Safari-, Opera-, Chrome-

The IE-specific `createStyleSheet()` method creates a `styleSheet` object, a type of object that includes `style` element objects as well as style sheets that are imported into a document through the `link` element. Thus, you can dynamically load an external style sheet even after a page has loaded.

Unlike the other create methods entering W3C DOM usage, the `createStyleSheet()` method not only creates the style sheet, but it inserts the object into the document object model immediately. Thus, any style sheet rules that belong (or are assigned to) that object take effect on the page right away. If you'd rather create a style sheet and delay its deployment, you should use the `createElement()` method and element object assembly techniques.

If you don't specify any parameters to the method in WinIE, an empty `styleSheet` object is created. It is assumed that you will then use `styleSheet` object methods, such as `addRule()` to add the details to the style sheet. To link in an external style sheet file, assign the file's URL to

the first parameter of the method. The newly imported style sheet is appended to the end of the `document.styleSheets` array of `styleSheet` objects. An optional second parameter lets you specify precisely where in the sequence of style sheet elements the newly linked style sheet should be inserted. A style sheet rule for any given selector is overridden by a style sheet for the same selector that appears later in the sequence of style sheets in a document.

Example

Listing 29-12 demonstrates adding an internal and external style sheet to a document. For the internal addition, the `addStyle1()` function invokes `document.createStyleSheet()` and adds a rule governing the `p` elements of the page. In the `addStyle2()` function, an external file is loaded. That file contains the following two style rules:

```
h2 {font-size:20pt; color:blue;}
p  {color:blue;}
```

Notice that by specifying a position of zero for the imported style sheet, the addition of the internal style sheet always comes afterward in `styleSheet` object sequence. Thus, except when you deploy only the external style sheet, the red text color of the `p` elements override the blue color of the external style sheet. If you remove the second parameter of the `createStyleSheet()` method in `addStyle2()`, the external style sheet is appended to the end of the list. If it is the last style sheet to be added, the blue color prevails. Repeatedly clicking the buttons in this example continues to add the style sheets to the document.

LISTING 29-12

Creating and Applying Style Sheets

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>document.createStyleSheet() Method</title>
      <script type="text/javascript">
         function addStyle1()
         {
            var newStyle = document.createStyleSheet();
            newStyle.addRule("P", "font-size:16pt; color:red");
         }

         function addStyle2()
         {
            var newStyle = document.createStyleSheet("jsb29-12.css",0);
         }

         // bind the event handlers
         function addEvent(elem, evtType, func)
         {
            if (elem.addEventListener)
            {
               elem.addEventListener(evtType, func, false);
```

continued

LISTING 29-12 *(continued)*

```
            }
            else if (elem.attachEvent)
            {
               elem.attachEvent("on" + evtType, func);
            }
            else
            {
               elem["on" + evtType] = func;
            }
         }
         addEvent(window, "load", function()
         {
            addEvent(document.getElementById("addint"), "click", addStyle1);
            addEvent(document.getElementById("addext"), "click", addStyle2);
         });
      </script>
   </head>
   <body>
      <h1>document.createStyleSheet() Method</h1>
      <hr />
      <form>
         <input type="button" id="addint" value="Add Internal" /> 
         <input type="button" id="addext" value="Add External" />
      </form>
      <h2>Section 1</h2>
      <p>Lorem ipsum dolor sit amet, consectetaur adipisicing elit, sed do
         eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim
         adminim veniam, quis nostrud exercitation ullamco laboris nisi ut
         aliquip ex ea commodo consequat.</p>
      <h2>Section 2</h2>
      <p>Duis aute irure dolor in reprehenderit involuptate velit esse cillum
         dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non
         proident, sunt in culpa qui officia deseruntmollit anim id est
         laborum.</p>
   </body>
</html>
```

Related Item: `styleSheet` object (see Chapter 38, “Style Sheet and Style Objects”)

createTextNode("*text*")

Returns: Object

Compatibility: WinIE5+, MacIE5+, NN6+, Moz+, Safari+, Opera+, Chrome+

A text node is a W3C DOM object that contains body text without any HTML (or XML) tags, but is usually contained by (meaning, is a child of) an HTML (or XML) element. Without the IE `innerText` convenience property for modifying the text of an element, the W3C DOM relies on the node hierarchy of a document (Mozilla exceeds the W3C DOM by providing an `innerHTML`

property, which you can use to replace text in an element). To insert or replace text inside an HTML element in the W3C DOM way, you create the text node and then use methods of the parent element (for example, `appendChild()`, `insertBefore()`, and `replaceChild()`, all described in Chapter 26) to modify the document's content. To generate a fresh text node, use `document.createTextNode()`.

The sole parameter of the `createTextNode()` method is a string whose text becomes the `nodeValue` of the text node object returned by the method. You can also create an empty text node (passing an empty string) and assign a string to the `nodeValue` of the object later. As soon as the text node is present in the document object model, scripts can simply change the `nodeValue` property to modify the text of an existing element. For more details on the role of text nodes in the W3C DOM, see Chapter 25, "Document Object Model Essentials."

Example

Although Chapters 25 and 26 (Listing 26-21, for instance) provide numerous examples of the `createTextNode()` method at work, using The Evaluator is instructive to see just what the method generates in IE5+/W3C. You can use one of the built-in global variables of The Evaluator to hold a reference to a newly generated text node by entering the following statement into the top text field:

```
a = document.createTextNode("Hello")
```

The Results box shows that an object was created. Now, look at the properties of the object by entering `a` into the bottom text field. The precise listings of properties varies between IE5+ and W3C browsers, but the W3C DOM properties that they share in common indicate that the object is a node type 3 with a node name of `#text`. No parents, children, or siblings exist yet because the object created here is not part of the document hierarchy tree until it is explicitly added to the document.

To see how insertion works, enter the following statement into the top text field to append the text node to the `myP` paragraph:

```
document.getElementById("myP").appendChild(a)
```

The word *Hello* appears at the end of the simple paragraph lower on the page. Now you can modify the text of that node either through the reference, from the point of view of the containing `p` element, or through the global variable reference for the newly created node:

```
document.getElementById("myP").lastChild.nodeValue = "Howdy"
```

or

```
a.nodeValue = "Howdy"
```

Related Item: `document.createElement()` method

createTreeWalker(*rootNode, whatToShow, filterFunction, entityRefExpansion*)

Returns: `TreeWalker` object reference

Compatibility: WinIE-, MacIE-, NN7+, Moz1.4+, Safari+, Opera+, Chrome+

The `document.createTreeWalker()` method creates an instance of a `TreeWalker` object that can be used to navigate the document tree. The first parameter to the method indicates the node in

the document that is to serve as the root node of the tree. The second parameter is an integer constant that specifies one of several built-in filters for selecting nodes to be included in the tree. Following are the possible acceptable values for this parameter:

NodeFilter.SHOW_ALL	NodeFilter.SHOW_ATTRIBUTE
NodeFilter.SHOW_CDATA_SECTION	NodeFilter.SHOW_COMMENT
NodeFilter.SHOW_DOCUMENT	NodeFilter.SHOW_DOCUMENT_FRAGMENT
NodeFilter.SHOW_DOCUMENT_TYPE	NodeFilter.SHOW_ELEMENT
NodeFilter.SHOW_ENTITY	NodeFilter.SHOW_ENTITY_REFERENCE
NodeFilter.SHOW_NOTATION	NodeFilter.SHOW_PROCESSING_INSTRUCTION
NodeFilter.SHOW_TEXT	

The third parameter to the `createNodeIterator()` method is a reference to a filter function that can filter nodes even further than the `whatToShow` parameter. This function must accept a single node and return an integer value based upon one of the following constants: `NodeFilter.FILTER_ACCEPT`, `NodeFilter.FILTER_REJECT`, or `NodeFilter.FILTER_SKIP`. The idea is that you code the function to perform a test on each node and return an indicator value that lets the node iterator know whether or not to include the node in the tree. Your function doesn't loop through nodes. The `TreeWalker` object mechanism repetitively invokes the function as needed to look for the presence of whatever characteristic you wish to use as a filter.

The final parameter to the method is a Boolean value that determines whether or not the content of entity reference nodes should be treated as hierarchical nodes. This parameter applies primarily to XML documents.

Related Items: `TreeWalker` object

elementFromPoint(*x*, *y*)

Returns: Element object reference

Compatibility: WinIE4+, MacIE4+, NN-, Moz+, Safari+, Opera+, Chrome+

Originally IE-specific, the `elementFromPoint()` method returns a reference to whatever element object occupies the point whose integer coordinates are supplied as parameters to the method. The coordinate plane is that of the document, whose top-left corner is at point `0,0`. This coordinate plane can be very helpful in interactive designs that need to calculate collision detection between positioned objects or mouse events.

When more than one object occupies the same point (for example, one element is positioned atop another), the element with the highest z-index value is returned. A positioned element always wins when placed atop a normal body-level element. And if multiple overlapping positioned elements have the same z-index value (or none by default), the element that comes last in the source code order is returned for the coordinate that they share in common.

Example

Listing 29-13 is a document that contains many different types of elements, each of which has an ID attribute assigned to it. The `onmouseover` event handler for the `document` object invokes a

function that finds out which element the cursor is over when the event fires. Note that the event coordinates are `event.clientX` and `event.clientY`, which use the same coordinate plane as the page for their point of reference. As you roll the mouse over every element, its ID appears on the page. Some elements, such as `br` and `tr`, occupy no space in the document, so you cannot get their IDs to appear. On a typical browser screen size, a positioned element rests atop one of the paragraph elements so that you can see how the `elementFromPoint()` method handles overlapping elements. If you scroll the page, the coordinates for the event and the page's elements stay in sync.

LISTING 29-13

Tracking the Mouse as It Passes over Elements

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>document.elementFromPoint() Method</title>
      <script type="text/javascript">
         function replaceHTML(elem, text)
         {
            while(elem.firstChild)
            {
               elem.removeChild(elem.firstChild);
            }
            elem.appendChild(document.createTextNode(text));
         }

         function showElemUnderneath(evt)
         {
            var elem
            elem = document.elementFromPoint(evt.clientX, evt.clientY);
            replaceHTML(document.getElementById("mySpan"), elem.id);
         }

      </script>
   </head>
   <body id="myBody" onmousemove="showElemUnderneath(event)">
      <h1 id="header">document.elementFromPoint() Method</h1>
      <hr id="myHR" />
      <p id="instructions">Roll the mouse around the page. The coordinates
         of the mouse pointer are currently atop an element
         whose ID is: "<span id='mySpan' style='font-weight:bold;'></span>".</p>
      <br id="myBR" />
      <form id="myForm">
         <input id="myButton" type="button" value="Sample Button" /> 
      </form>
      <table border="1" id="myTable">
         <tr id="tr1">
            <td id="td_A1">Cell A1</td>
            <td id="td_B1">Cell B1</td>
         </tr>
```

continued

LISTING 29-13 *(continued)*

```
      <tr id="tr2">
         <td id="td_A2">Cell A2</td>
         <td id="td_B2">Cell B2</td>
      </tr>
   </table>
   <h2 id="sec1">Section 1</h2>
   <p id="p1">Lorem ipsum dolor sit amet, consectetaur adipisicing elit, sed
      do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim
      adminim veniam, quis nostrud exercitation ullamco laboris nisi ut
      aliquip ex ea commodo consequat.</p>
   <h2 id="sec2">Section 2</h2>
   <p id="p2">Duis aute irure dolor in reprehenderit involuptate velit esse
      cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat
      cupidatat non proident, sunt in culpa qui officia deseruntmollit anim
      id est laborum.</p>
   <div id="myDIV"
        style="position:absolute; top:340px; left:300px; background-color:yellow;">
      Here is a positioned element.
   </div>
  </body>
</html>
```

Related Items: `event.clientX`, `event.clientY` properties; positioned objects (see Chapter 43, “Positioned Objects”)

`evaluate("expression", contextNode, resolver, type, result)`

Returns: XPath result object reference

Compatibility: WinIE-, MacIE-, NN7+, Moz+, Safari+, Opera+, Chrome+

The `document.evaluate()` method evaluates an XPath expression and returns an XPath result object (`XPathResult`). The most important parameter to this method is the first one, which contains the actual XPath expression as a string. The second parameter is the context node to which the expression applies, whereas the third parameter is the namespace resolver (see the `createNSResolver()` method). The `resolver` parameter can be specified as `null` as long as there aren't any namespace prefixes used within the expression.

The `type` parameter determines the type of the result of the expression and is specified as one of the XPath result types, such as 0 for any type, 1 for number, 2 for string, and so forth. Finally, a reusable `result` object can be specified in the last parameter, which will then be modified and returned from the method as the result of the expression.

Related Item: `createNSResolver()` method

`execCommand("`*`commandName`*`"[, `*`UIFlag`*`] [, `*`param`*`])`

Returns: Boolean

Compatibility: WinIE4+, MacIE-, NN7.1+, Moz1.3+, Safari1.3+, Opera+, Chrome+

The execCommand() method is the JavaScript gateway to a set of commands that is outside of the methods defined for objects in the object model. A series of related methods (queryCommandEnable() and others) also facilitate management of these commands.

The syntax for the execCommand() method requires at least one parameter, a string version of the command name. Command names are not case-sensitive. An optional second parameter is a Boolean flag to instruct the command to display any user interface artifacts that may be associated with the command. The default is false. For the third parameter, some commands require that an attribute value be passed for the command to work. For example, to set the font size of a text range, the syntax is

```
myRange.execCommand("FontSize", true, 5);
```

The execCommand() method returns Boolean true if the command is successful; false if not successful. Some commands can return values (for example, finding out the font name of a selection), but those are accessed through the queryCommandValue() method.

In Internet Explorer, most of these commands operate on body text selections that are TextRange objects. As described in Chapter 33, a TextRange object must be created under script control. But a TextRange object can be created in response to a user selecting some text in the document. Because a TextRange object is independent of the element hierarchy (indeed, a TextRange can spread across multiple nodes), it cannot respond to style sheet specifications. Thus, many of the commands that can operate on a TextRange object have to do with formatting or modifying the text. For a list of commands that work exclusively on TextRange objects, see the TextRange.execCommand() method in Chapter 33.

Although many of the commands intended for the TextRange also work when invoked from the document object, in this section the focus is on those commands that have scope over the entire document. Table 29-2 lists those few commands that work with the document. Also listed are many commands that work exclusively on text selections in the document, whether the selections are made manually by the user or with the help of the TextRange object (see Chapter 33).

Mozilla 1.4 and Safari 1.3 added a feature that allows scripts to turn an iframe element's document object into an HTML editable document. Here is an example of how to center the selected text in an iframe with an ID of msg:

```
document.getElementById("msg").contentDocument.execCommand("JustifyCenter");
```

Note that the contentDocument property is used to access the iframe as a document. Visit http://www.mozilla.org/editor for additional details and examples of the document.execCommand() method.

Example

You can find many examples of the execCommand() method for the TextRange object in Chapter 33. But you can try out the document-specific commands in The Evaluator (see Chapter 4) in Internet Explorer if you like. Try each of the following statements in the top text box and click the Evaluate button:

```
document.execCommand("Refresh")
document.execCommand("SelectAll")
document.execCommand("Unselect")
```

TABLE 29-2

document.execCommand() Commands

Command	Parameter	Description
BackColor	Color String	Encloses the current selection with a font element whose style attribute sets the background-color style to the parameter value.
CreateBookmark	Anchor String	Encloses the current selection (or text range) with an anchor element whose name attribute is set to the parameter value.
CreateLink	URL String	Encloses the current selection with an a element whose href attribute is set to the parameter value.
decreaseFontSize	none	Encloses the current selection with a small element.
Delete	none	Deletes the current selection from the document.
FontName	Font Face(s)	Encloses the current selection with a font element whose face attribute is set to the parameter value.
FontSize	Size String	Encloses the current selection with a font element whose size attribute is set to the parameter value.
foreColor	Color String	Encloses the current selection with a font element whose color attribute is set to the parameter value.
FormatBlock	Block Element String	Encloses the current selection with the specified block element. IE only supports the h1 through h6, address and pre block elements. Other browsers support all block elements.
increaseFontSize	none	Encloses the current selection with a big element.
Indent	None	Indents the current selection.
InsertHorizontalRule	Id String	Encloses the current selection in an hr element, whose id attribute is set to the parameter value.
InsertImage	Id String	Encloses the current selection in an img element, whose id attribute is set to the parameter value.
InsertParagraph	Id String	Encloses the current selection in an p element, whose id attribute is set to the parameter value.
JustifyCenter	None	Centers the current selection.
JustifyFull	None	Full-justifies the current selection.
JustifyLeft	None	Left-justifies the current selection.
JustifyRight	None	Right-justifies the current selection.
Outdent	None	Outdents the current selection.

TABLE 29-2 *(continued)*

Command	Parameter	Description
`Refresh`	None	Reloads the page.
`RemoveFormat`	None	Removes formatting for the current selection.
`SelectAll`	None	Selects all text of the document.
`UnBookmark`	None	Removes anchor tags that surround the current selection.
`Unlink`	None	Removes link tags that surround the current selection.
`Unselect`	None	Deselects the current selection anywhere in the document.

All methods return `true` in the Results box.

Because any way you can evaluate a statement in The Evaluator forces a body selection to become de-selected before the evaluation takes place, you can't experiment this way with the selection-oriented commands.

Related Items: `queryCommandEnabled()`, `queryCommandIndterm()`, `queryCommandState()`, `queryCommandSupported()`, `queryCommandText()`, `queryCommandValue()` methods

getElementById("*elementID*")

Returns: Element object reference

Compatibility: WinIE5+, MacIE5+, NN6+, Moz+, Safari+, Opera+, Chrome+

The `document.getElementById()` method is the W3C DOM syntax for retrieving a reference to any element in a document that has a unique identifier assigned to its `id` attribute. If the document contains more than one instance of an ID, the method returns a reference to the first element in source code order with that ID. Because this method is such an important avenue to writing references to objects that are to be modified under script control, you can see how important it is to assign unique IDs to elements.

This method's name is quite a finger twister for scripters, especially compared to the IE4+ convention of letting a reference to any element begin simply with the object's ID. However, the `getElementById()` method is the modern way of acquiring an element's reference for W3C DOM-compatible browsers, including IE. When you type this method, be sure to use a lowercase `d` as the last character of the method name.

Unlike some other element-oriented methods (for example, `getElementsByTagName()`), which can be invoked on any element in a document, the `getElementById()` method works exclusively with the `document` object.

Example

You can find many examples of this method in use throughout this book, but you can take a closer look at how it works by experimenting in The Evaluator (Chapter 4, "JavaScript Essentials"). A number of elements in The Evaluator have IDs assigned to them, so that you can use the method to inspect the objects and their properties. Enter the following statements into both the top and bottom text

fields of The Evaluator. Results from the top field are references to the objects; results from the bottom field are lists of properties for the particular object.

```
document.getElementById("myP")
document.getElementById("myEM")
document.getElementById("myTitle")
document.getElementById("myScript")
```

Related Item: `getElementsByTagName()` method (see Chapter 26, "Generic HTML Element Objects")

getElementsByName("*elementName*")

Returns: Array

Compatibility: WinIE5+, MacIE5+, NN6+, Moz+, Safari+, Opera+, Chrome+

The `document.getElementsByName()` method returns an array of references to objects whose `name` attribute is assigned the element name passed as the method's attribute. Although other browsers recognize `name` attributes even for elements that don't have them by default, IE does not. Therefore, for maximum cross-browser compatibility, use this method only to locate elements that have `name` attributes defined for them by default, such as form control elements. If the element does not exist in the document, the method returns an array of zero length.

For the most part, you are best served by using IDs on elements and the `getElementById()` method to unearth references to individual objects. But some elements, especially the `input` element of type radio, use the `name` attribute to group elements together. In that case, a call to `getElementsByName()` returns an array of all elements that share the name — facilitating perhaps a `for` loop that inspects the `checked` property of a radio button group. Thus, instead of using the old-fashioned approach by way of the containing form object:

```
var buttonGroup = document.forms[0].radioGroupName;
```

you can retrieve the array more directly:

```
var buttonGroup = document.getElementsByName(radioGroupName);
```

In the latter case, you operate independently of the containing form object's index number or name. This assumes, of course, that a group name is not shared elsewhere on the page, which would certainly lead to confusion.

Example

Use The Evaluator (Chapter 4, "JavaScript Essentials") to test out the `getElementsByName()` method. All form elements in the upper part of the page have names associated with them. Enter the following statements into the top text field and observe the results:

```
document.getElementsByName("output")
document.getElementsByName("speed").length
document.getElementsByName("speed")[0].value
```

You can also explore all of the properties of the text field by typing the following expression into the bottom field:

```
document.getElementsByName("speed")[0]
```

Related Items: document.getElementsById(), getElementsByTagName() methods

importNode(node, deep)

Returns: Node object reference

Compatibility: WinIE-, MacIE-, NN7+, Moz+, Safari+, Opera+, Chrome+

The document.importNode() method imports a node from another document object into the current document object. A copy of the original node is made when the node is imported, meaning that the original node remains unchanged. The second parameter to the method is a Boolean value that determines whether or not the node's entire subtree is imported (true) or just the node itself (false).

open(["*mimeType*"] [,"*replace*"])

Returns: Nothing

Compatibility: WinIE3+, MacIE3+, NN2+, Moz+, Safari+, Opera+, Chrome+

Opening a document is different from opening a window. In the case of a window, you're creating a new object, both on the screen and in the browser's memory. Opening a document, on the other hand, tells the browser to get ready to accept some data for display in the window named or implied in the reference to the document.open() method. (For example, parent.frames[1] .document.open() may refer to a different frame in a frameset, whereas document.open() implies the current window or frame.) Therefore, the method name may mislead newcomers because the document.open() method has nothing to do with loading documents from the web server or hard disk. Rather, this method is a prelude to sending data to a window through the document.write() or document.writeln() methods. In a sense, the document.open() method merely opens the valve of a pipe; the other methods send the data down the pipe like a stream, and the document.close() method closes that valve as soon as the page's data has been sent in full.

The document.open() method is optional because a document.write() method that attempts to write to a closed document automatically clears the old document and opens the stream for a new one. Whether or not you use the document.open() method, be sure to use the document.close() method after all the writing has taken place.

An optional parameter to the document.open() method enables you to specify the nature of the data being sent to the window. A MIME (Multipurpose Internet Mail Extension) type is a specification for transferring and representing multimedia data on the Internet (originally for mail transmission, but now applicable to all Internet data exchanges). You've seen MIME depictions in the list of helper applications in your browser's preferences settings. A pair of data type names separated by a slash represents a MIME type (such as text/html and image/gif). When you specify a MIME type as a parameter to the document.open() method, you're instructing the browser about the kind of data it is about to receive, so that it knows how to render the data. Common values that most browsers accept are:

```
text/html
text/plain
image/gif
image/jpeg
image/xbm
```

If you omit the parameter, JavaScript assumes the most popular type, `text/html` — the kind of data you typically assemble in a script prior to writing to the window. The `text/html` type includes any images that the HTML references. Specifying any of the image types means that you have the raw binary representation of the image that you want to appear in the new document — possible, but unlikely.

Another possibility is to direct the output of a `write()` method to a plug-in. For the `mimeType` parameter, specify the plug-in's MIME type (for example, `application/x-director` for Shockwave). Again, the data you write to a plug-in must be in a form that it knows how to handle. The same mechanism also works for writing data directly to a helper application.

Note

IE accepts only the `text/html` MIME type parameter. ■

Modern browsers include a second, optional parameter to the method: `replace`. This parameter does for the `document.open()` method what the `replace()` method does for the `location` object. For `document.open()`, it means that the new document you are about to write replaces the previous document in the window or frame, with the end result that the replaced window or frame's history is not recorded.

Example

You can see an example of where the `document.open()` method fits in the scheme of dynamically creating content for another frame in the discussion of the `document.write()` method later in this chapter.

Related Items: `document.close()`, `document.clear()`, `document.write()`, `document.writeln()` methods

queryCommandEnabled("*commandName*")
queryCommandIndterm("*commandName*")
queryCommandState("*commandName*")
queryCommandSupported("*commandName*")
queryCommandText("*commandName*")
queryCommandValue("*commandName*")

Returns: Various values

Compatibility: WinIE4+, MacIE-, NN7.1, Moz1.3+, Safari+, Opera+, Chrome+

These six methods lend further support to the `execCommand()` method for `document` and `TextRange` objects. If you choose to use the `execCommand()` method to achieve some stylistic change on a text selection, you can use some of these query methods to make sure the browser supports the desired command and to retrieve any returned values. Table 29-3 summarizes the purpose and returned values for each of the query methods. Note that browser support for these seven methods is uneven.

Because the `execCommand()` method cannot be invoked on a page while it is still loading, any such invocations that may collide with the loading of a page should check with `queryCommandEnabled()` prior to invoking the command. Validating that the browser version

running the script supports the desired command is also a good idea. Therefore, you may want to wrap any command call with the following conditional structure:

```
if (document.queryCommandEnabled(commandName) &&
    document.queryCommandSupported(commandName)) {
    //  ...
}
```

TABLE 29-3

Query Commands

queryCommand	Returns	Description
`Enabled`	Boolean	Reveals whether the `document` or `TextRange` object is in a suitable state to be invoked.
`Indterm`	Boolean	Reveals whether the command is in an indeterminate state.
`CommandState`	Boolean | `null`	Reveals whether the command has been completed (`true`), is still working (`false`), or is in an indeterminate state (`null`).
`Supported`	Boolean	Reveals whether the command is supported in the current browser.
`Text`	String	Returns any text that may be returned by a command.
`Value`	Varies	Returns whatever value (if any) is returned by a command.

When using a command to read information about a selection, use the `queryCommandText()` or `queryCommandValue()` methods to catch that information (recall that the `execCommand()` method itself returns a Boolean value regardless of the specific command invoked).

Example

See the examples for these methods covered under the `TextRange` object in Chapter 33, "Body Text Objects."

Related Items: `TextRange` object (see Chapter 33); `execCommand()` method

recalc([*allFlag*])

Returns: Nothing

Compatibility: WinIE5+, MacIE-, NN-, Moz-, Safari-, Opera-, Chrome-

IE5 introduced the concept of dynamic properties. With the help of the `setExpression()` method of all elements and the `expression()` style sheet value, you can establish dependencies between object properties and potentially dynamic properties, such as a window's size or a draggable element's

location. After those dependencies are established, the `document.recalc()` method causes those dependencies to be recalculated — usually in response to some user action, such as resizing a window or dragging an element.

The optional parameter is a Boolean value. The default value, `false`, means that the recalculations are performed only on expressions for which the browser has detected any change since the last recalculation. If you specify `true`, however, all expressions are recalculated, whether they have changed or not.

Mozilla 1.4 includes a feature that allows scripts to turn an `iframe` element's document object into an HTML editable document. Part of the scripting incorporates the `document.execCommand()` and related methods. Visit `http://www.mozilla.org/editor` for current details and examples.

Example

You can see an example of `recalc()` in Listing 26-32 for the `setExpression()` method. In that example, the dependencies are between the current time and properties of standard element objects.

Related Items: `getExpression()`, `removeExpression()`, `setExpression()` methods (see Chapter 26, "Generic HTML Element Objects")

releaseEvents(*eventTypeList*)

Returns: Nothing

Compatibility: WinIE-, MacIE-, NN4+, Moz-, Safari-, Opera-, Chrome-

In NN4+, as soon as you enable event capture for a particular event type in a document, that capture remains in effect until the page unloads or you specifically disable the capture. You can turn off event capture for each event via the `releaseEvents()` method. See Chapter 32, "Event Objects," for more details on event capture and release.

Related Items: `captureEvents()`, `routeEvent()` methods

routeEvent([*eventObject*])

Returns: Nothing

Compatibility: WinIE-, MacIE-, NN4+, Moz-, Safari-, Opera-, Chrome-

If you capture a particular event type in NN4, your script may need to perform some limited processing on that event before letting it reach its intended target. To let an event pass through the object hierarchy to its target, you use the `routeEvent()` method, passing as a parameter the event object being handled in the current function. See Chapter 32, "Event Objects," for more details on event processing.

Related Items: `captureEvents()`, `releaseEvents()` methods

write("*string1*" [,"*string2*" ... [, "*stringn*"]])
writeln("*string1*" [,"*string2*" ... [, "*stringn*"]])

Returns: Boolean `true` if successful

Compatibility: WinIE3+, MacIE3+, NN2+, Moz+, Safari+, Opera+, Chrome+

Both of these methods send text to a document for display in its window. The only difference between the two methods is that `document.writeln()` appends a carriage return to the end of the string it sends to the document. This carriage return is helpful for formatting source code when viewed through the browser's source view window. For new lines in rendered HTML that is generated by these methods, you must still write a `<br>` to insert a line break.

A common, incorrect conclusion that many JavaScript newcomers make is that these methods enable a script to modify the contents of an existing document, which is not true. As soon as a document has loaded into a window (or frame), the only fully backward-compatible text that you can modify without reloading or rewriting the entire page is the content of `text` and `textarea` objects. In IE4+, you can modify HTML and text through the `innerHTML`, `innerText`, `outerHTML`, and `outerText` properties of any element. For W3C DOM–compatible browsers, you can modify an element's text by setting its `nodeValue` or `innerHTML` properties. The preferred approach for modifying the content of a node involves strict adherence to the W3C DOM, which requires creating and inserting or replacing new elements, as described in Chapter 26, and demonstrated in examples throughout this chapter and the rest of the book.

The two safest ways to use the `document.write()` and `document.writeln()` methods are to:

- Write some or all of the page's content by way of scripts embedded in the document
- Send HTML code either to a new window or to a separate frame in a multiframe window

For the first case, you essentially interlace script segments within your HTML. The scripts run as the document loads, writing whatever scripted HTML content you like. This task is exactly what you did in `script1.html` in Chapter 3, "Selecting and Using Your Tools." This task is also how you can have one page generate browser-specific HTML when a particular class of browser requires unique syntax.

In the latter case, a script can gather input from the user in one frame and then algorithmically determine the layout and content destined for another frame. The script assembles the HTML code for the other frame as a string variable (including all necessary HTML tags). Before the script can write anything to the frame, it can optionally open the layout stream (to close the current document in that frame) with the `parent.frameName.document.open()` method. In the next step, a `parent.frameName.document.write()` method pours the entire string into the other frame. Finally, a `parent.frameName.document.close()` method ensures that the total data stream is written to the window. Such a frame looks just the same as if it were created by a source document on the server rather than on-the-fly in memory. The `document` object of that window or frame is a full citizen as a standard `document` object. You can, therefore, even include scripts as part of the HTML specification for one of these temporary HTML pages.

After an HTML document (containing a script that is going to write via the `write()` or `writeln()` methods) loads completely, the page's incoming stream closes automatically. If you then attempt to apply a series of `document.write()` statements, the first `document.write()` method completely removes all vestiges of the original document. That includes all of its objects and scripted variable values. Therefore, if you try to assemble a new page with a series of `document.write()` statements, the script and variables from the original page will be gone before the second `document.write()` statement executes. To get around this potential problem, assemble the content for the new screen as one string variable, and then pass that variable as the parameter to a single `document.write()` statement. Also be sure to include a `document.close()` statement in the next line of script.

Assembling HTML in a script to be written via the `document.write()` method often requires skill in concatenating string values and nesting strings. A number of JavaScript `String` object shortcuts facilitate the formatting of text with HTML tags (see Chapter 15, "The String Object," for details).

If you are writing to a different frame or window, you are free to use multiple `document.write()` statements if you like. Whether your script sends lots of small strings via multiple `document.write()` methods, or assembles a larger string to be sent through one `document.write()` method, depends partly on the situation and partly on your own scripting style. From a performance standpoint, a fairly standard procedure is to do more preliminary work in memory and place as few I/O (input/output) calls as possible. On the other hand, making a difficult-to-track mistake is easier in string concatenation when you assemble longer strings. You should use the system that's most comfortable for you.

You may see another little-known way of passing parameters to these methods. Instead of concatenating string values with the plus (+) operator, you can also bring string values together by separating them with commas, in which case the strings appear to be arguments to the `document.write()` method. For example, the following two statements produce the same results:

```
document.write("Today is " + new Date());
document.write("Today is ",new Date());
```

Neither form is better than the other, so use the one that feels more comfortable to your existing programming style.

Note

Dynamically generating scripts requires an extra trick, especially in NN. The root of the problem is that if you try code such as `document.write("<script></script>")`, the browser interprets the end script tag as the end of the script that is doing the writing. You have to trick the browser by separating the end tag into a couple of components. Escaping the forward slash also helps. For example, if you want to load a different `.js` file for each class of browser, the code looks similar to the following:

```
// variable 'browserVer' is a browser-specific string
// and 'page' is the HTML your script is accumulating
// for document.write()
page += "<script type='text/javascript' src='" +
        browseVer + ".js'><" + "\/script>"; ■
```

Using the `document.open()`, `document.write()`, and `document.close()` methods to display images in a document requires some small extra steps. First, any URL assignments that you write via `document.write()` must be complete (not relative) URL references. Alternatively, you can write the `<base>` tag for the dynamically generated page so that its `href` attribute value matches that of the file that is writing the page.

The other image trick is to be sure to specify `height` and `width` attributes for every image, scripted or otherwise. Document-rendering performance is improved on all platforms, because the values help the browser lay out elements even before their details are loaded.

In addition to the `document.write()` example that follows (see Listings 29-14 through 29-16), you can find fuller implementations that use this method to assemble images and bar charts in many of the applications in Chapters 52 through 61 on the CD-ROM. Because you can assemble any valid HTML as a string to be written to a window or frame, a customized, on-the-fly document can be as elaborate as the most complex HTML document that you can imagine.

Example

The example in Listings 29-14 through 29-16 demonstrates several important points about using the `document.write()` and `document.writeln()` methods for writing to another frame. First is

the fact that you can write any HTML code to a frame, and the browser accepts it as if the source code came from an HTML file somewhere. In the example, we assemble a complete HTML document, including basic HTML tags for completeness.

LISTING 29-14

A Frameset for the Document Writing Example

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Writin' to the doc</title>
   </head>
   <frameset rows="50%,50%">
      <frame name="Frame1" src="jsb29-15.html" />
      <frame name="Frame2" src="jsb29-16.html" />
   </frameset>
</html>
```

LISTING 29-15

Writing a Document Based upon User Input

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Document Write Controller</title>
      <script type="text/javascript">
         function takePulse(form)
         {
            var msg = "<html><head><title>On The Fly with " +
                      form.yourName.value + "<\/title><\/head>";
            msg += "<body bgcolor='salmon'><h1>Good Day " +
                    form.yourName.value +
                   "!<\/h1><hr />";
            for (var i = 0; i < form.how.length; i++)
            {
               if (form.how[i].checked)
               {
                  msg += form.how[i].value;
                  break;
               }
            }
            msg += "<br />Make it a great day!<\/body><\/html>";
            parent.Frame2.document.write(msg);
            parent.Frame2.document.close();
         }
```

continued

LISTING 29-15 *(continued)*

```
      function getTitle()
      {
        alert("Lower frame document.title is now:" + parent.Frame2.document.title);
      }

      // bind the event handlers
      function addEvent(elem, evtType, func)
      {
         if (elem.addEventListener)
         {
            elem.addEventListener(evtType, func, false);
         }
         else if (elem.attachEvent)
         {
            elem.attachEvent("on" + evtType, func);
         }
         else
         {
            elem["on" + evtType] = func;
         }
      }
      addEvent(window, "load", function()
      {
         addEvent(document.getElementById("enter"), "click",
                  function(evt)
                  {takePulse(document.getElementById("enter").form)
                  });
         addEvent(document.getElementById("peek"), "click", getTitle);
      });
   </script>
</head>
<body>
   Fill in a name, and select how that person feels today. Then click "Write
   To Below" to see the results in the bottom frame.
   <form>
      Enter your first name:
      <input type="text" name="yourName" value="Dave" />
      <p>How are you today?
         <input type="radio" name="how"
                value="I hope that feeling continues forever."
                checked="checked" />
         Swell
         <input type="radio" name="how"
                value="You may be on your way to feeling Swell" />
         Pretty Good
         <input type="radio" name="how"
                value="Things can only get better from here." />So-So
      </p>
      <p><input type="button" id="enter" name="enter" value="Write To Below" /></p>
```

```
      <hr />
      <input type="button" id="peek" name="peek" value="Check Lower Frame Title" />
    </form>
  </body>
</html>
```

LISTING 29-16

A Placeholder Page for the Document Writing Example

```
<!DOCTYPE html>
<html>
  <head>
    <meta http-equiv="content-type" content="text/html;charset=utf-8">
    <title>Placeholder</title>
  </head>
  <body>
  </body>
</html>
```

It is important to note that this example customizes the content of the document based on user input. This customization makes the experience of working with your web page feel far more interactive to the user — yet you're doing it without any server-side programs.

The second point we want to bring home is that the document created in the separate frame by the `document.write()` method is a genuine `document` object. In this example, the `<title>` tag of the written document changes if you redraw the lower frame after changing the entry of the name field in the upper frame. If you click the lower button after updating the bottom frame, you see that the `document.title` property has, indeed, changed to reflect the `<title>` tag written to the browser in the course of displaying the frame's page. The fact that you can artificially create full-fledged, JavaScript `document` objects on the fly represents one of the most important powers of serverless CGI scripting (for information delivery to the user) with JavaScript. You have much to take advantage of here if your imagination is up to the task.

Note that you can easily modify Listing 29-15 to write the results to the same frame as the document containing the field and buttons. Instead of specifying the lower frame:

```
parent.frames[1].document.open();
parent.frames[1].document.write(msg);
parent.frames[1].document.close();
```

The code simply can use:

```
document.open();
document.write(msg);
document.close();
```

This code would replace the form document with the results and not require any frames in the first place. Because the code assembles all of the content for the new document into one variable value, that data survive the one `document.write()` method.

The frameset document (see Listing 29-14) creates a blank frame by loading a blank document (see Listing 29-16). An alternative that we highly recommend is to have the framesetting document fill the frame with a blank document of its own creation. See the section “Blank frames” in Chapter 27, “Window and Frame Objects,” for further details about this technique.

Related Items: `document.open()`; `document.close()`; `document.clear()` methods

Event handlers

onselectionchange

Compatibility: WinIE5.5+, MacIE-, NN-, Moz-, Safari-, Opera-, Chrome-

The `onselectionchange` event can be triggered by numerous user actions, although all of those actions occur on elements that are under the influence of the WinIE5.5+ edit mode.

Related Item: `oncontrolselect` event handler

onstop

Compatibility: WinIE5+, MacIE-, NN-, Moz-, Safari-, Opera-, Chrome-

The `onstop` event fires in WinIE5+ when the user clicks the browser’s Stop button. Use this event handler to stop potentially runaway script execution on a page, because the Stop button does not otherwise control scripts after a page has loaded. If you are having a problem with a runaway repeat loop during development, you can temporarily use this event handler to let you stop the script for debugging.

Example

Listing 29-17 provides a simple example of an intentional infinitely looping script. In case you load this page into a browser other than IE5+, you can click the Halt Counter button to stop the looping. The Halt Counter button, as well as the `onstop` event handler, invokes the same function.

LISTING 29-17

Stopping a Script Using the onstop Event Handler

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>onStop Event Handler</title>
      <script type="text/javascript">
         var counter = 0;
         var timerID;
         function startCounter()
         {
            document.forms[0].display.value = ++counter;
            //clearTimeout(timerID)
            timerID = setTimeout("startCounter()", 10);
         }
         function haltCounter()
```

```
        {
           clearTimeout(timerID);
           counter = 0;
        }

        // bind the event handlers
        function addEvent(elem, evtType, func)
        {
           if (elem.addEventListener)
           {
              elem.addEventListener(evtType, func, false);
           }
           else if (elem.attachEvent)
           {
              elem.attachEvent("on" + evtType, func);
           }
           else
           {
              elem["on" + evtType] = func;
           }
        }
        addEvent(window, "load", function()
        {
           addEvent(document, "stop", haltCounter);
           addEvent(document.getElementById("start"), "click", startCounter);
           addEvent(document.getElementById("halt"), "click", haltCounter);
        });
     </script>
  </head>
  <body>
     <h1>onStop Event Handler</h1>
     <hr />
     <p>Click the browser's Stop button (in IE) to stop the script counter.</p>
     <form>
        <p><input type="text" name="display" /></p>
        <input type="button" id="start" value="Start Counter" />
        <input type="button" id="halt" value="Halt Counter" />
     </form>
  </body>
</html>
```

Related Items: Repeat loops (Chapter 21, “Control Structures and Exception Handling”)

body Element Object

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

For HTML element properties, methods, and event handlers, see Chapter 26, “Generic HTML Element Objects.”

Properties	Methods	Event Handlers
aLink	createControlRange()	onafterprint
background	createTextRange()	onbeforeprint
bgColor	doScroll()	onscroll
bgProperties		
bottomMargin		
leftMargin		
link		
noWrap		
rightMargin		
scroll		
scrollLeft		
scrollTop		
text		
topMargin		
vLink		

Syntax

Accessing body element object properties or methods:

```
[window.] document.body.property | method([parameters])
```

About this object

In object models that reveal HTML element objects, the body element object is the primary container of the content that visitors see on the page. The body contains all rendered HTML. This special place in the node hierarchy gives the body object some unique powers, especially in the IE object model.

As if to signify the special relationship, both the IE and W3C object models provide the same shortcut reference to the body element: document.body. As a first-class HTML element object (as evidenced by the long lists of properties, methods, and event handlers covered in Chapter 26), you are also free to use other syntaxes to reach the body element.

You are certainly familiar with several body element attributes that govern body-wide content appearance, such as link colors (in three states) and background (color or image). But IE and NN/Mozilla (and the W3C so far) have some very different ideas about the body element's role in scripting documents. Many methods and properties that NN/Mozilla considers to be the domain of the window (for example, scrolling, inside window dimensions, and so forth), IE puts into the hands of the body element object. Therefore, whereas NN/Mozilla scrolls the window (and whatever it may contain), IE scrolls the body (inside whatever window it lives). And because the body element

fills the entire viewable area of a browser window or frame, that viewable rectangle is determined in IE by the body's `scrollHeight` and `scrollWidth` properties, whereas NN4+/Moz features `window.innerHeight` and `window.innerWidth` properties. This distinction is important to point out because when you are scripting window- or document-wide appearance factors, you may have to look for properties and methods for the `window` or `body` element object, depending on your target browser(s).

Note

Use caution when referencing the `document.body` object while the page is loading. The object may not officially exist until the page has completely loaded. If you need to set some initial properties through scripting, do so in response to the `onload` event handler located in the `<body>` tag. Attempts at setting `body` element object properties in immediate scripts inside the `head` element may result in error messages about the object not being found. ■

Properties

alink
bgColor
link
text
vLink

Value: Hexadecimal triplet or color name string Read/Write

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

The `aLink`, `link`, and `vLink` properties replaced the ancient `document` properties `alinkColor`, `linkColor`, and `vlinkColor`. The `bgColor` property is the same as the old `document.bgColor` property, while the `text` property replaced the `document.fgColor` property. These properties serve as the scripted equivalents of the HTML attributes for the `body` element — the new property names more closely align themselves with the HTML attributes than the old property names.

We use past tense when referring to these properties because CSS has largely made them obsolete. Granted, they still work, but will likely fall into disuse as web developers continue to embrace style sheets as the preferred means of altering color in web pages. Link colors that are set through pseudo-class selectors in style sheets (as `style` attributes of the `body` element) must be accessed through the `style` property for the `body` object.

background

Value: URL string Read/Write

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

The `background` property enables you to set or get the URL for the background image (if any) assigned to the `body` element. A `body` element's background image overlays the background color in case both attributes or properties are set. To remove an image from the document's background, set the `document.body.background` property to an empty string.

Similar to the properties that provide access to colors on the page, the background image in modern web pages should be set through style sheets, as opposed to the `body.background` property. In that case, you access the background programmatically through the `style` property of the `body` object.

bgColor

(See aLink)

bgProperties

Value: String constant Read/Write

Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-, Opera-, Chrome-

The IE-specific bgProperties property is an alternative way of adjusting whether the background image should remain fixed when the user scrolls the document or scroll with the document. Initial settings for this behavior should be done through the background-attachment CSS attribute and modified under script control by way of the body element's style.backgroundAttachment property.

No matter which way you reference this property, the only allowable values are the string constants scroll (the default) or fixed.

Example

Both of the following statements change the default behavior of background image scrolling in IE4+:

```
document.body.bgProperties = "fixed";
```

or

```
document.body.style.backgroundAttachment = "fixed";
```

The added benefit of using the style sheet version is that it also works in NN6+/Moz and other W3C browsers.

Related Item: body.background property

bottomMargin
leftMargin
rightMargin
topMargin

Value: Integer Read/Write

Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-, Opera-, Chrome-

The four IE-specific margin properties are alternatives to setting the corresponding four margin style sheet attributes for the body element (body.style.marginBottom, and so on). Style sheet margins represent blank space between the edge of an element's content and its next outermost container. In the case of the body element, that container is an invisible document container.

Of the four properties, only the one for the bottom margin may be confusing, if the content does not fill the vertical space of a window or frame. The margin value is not automatically increased to accommodate the extra blank space.

Example

Both of the following statements change the default left margin in IE4+. The style sheet version has an added benefit — it also works in NN6+/Moz and other W3C browsers.

```
document.body.leftMargin = 30;
```

and

```
document.body.style.marginLeft = 30;
```

Related Item: `style` object

`leftMargin`

(See `bottomMargin`)

`link`

(See `aLink`)

`noWrap`

Value: Boolean Read/Write

Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-, Opera-, Chrome-

The `nowrap` property enables you to modify the `body` element behavior normally set through the HTML `nowrap` attribute. Because the property name is a negative, the Boolean logic needed to control it can get confusing.

The default behavior for a `body` element is for text to wrap within the width of the window or frame. This behavior occurs when the value of `noWrap` is its default value of `false`. By turning `noWrap` to `true`, a line of text continues to render past the right edge of the window or frame until the HTML contains a line break (or end of paragraph). If the text continues on past the right edge of the window, the window (or frame) gains a horizontal scroll bar (unless, of course, a frame is set to not scroll).

By and large, users don't like to scroll in any direction if they don't have to. Unless you have a special need to keep single lines intact, let the default behavior rule the day.

Example

To change the word-wrapping behavior from the default, the statement is:

```
document.body.noWrap = true;
```

Related Items: None

`rightMargin`

(See `bottomMargin`)

`scroll`

Value: Constant string Read/Write

Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-, Opera-, Chrome-

The IE-specific `scroll` property provides scripted access to the IE-specific `scroll` attribute of a `body` element. By default, an IE `body` element displays a vertical scroll bar even if the height of the content does not warrant it; a horizontal scroll bar appears only when the content is forced to be wider than the window or frame. You can make sure that both scroll bars are hidden by setting the `scroll` attribute to `no` or changing it through a script. Possible values for this property are the constant strings `yes` and `no`.

Except for `frame` attributes and NN4+/Moz-signed scripts, other browsers do not provide facilities for turning off scroll bars under script control. You can generate a new window (via the `window.open()` method) and specify that its scroll bars be hidden.

Example

To change the scroll bar appearance from the default, the statement is:

```
document.body.scroll = "no";
```

Related Items: `window.scrollbars` property; `window.open()` method

scrollLeft
scrollTop

Value: Integer Read/Write

Compatibility: WinIE4+, MacIE4+, NN7+, Moz+, Safari+, Opera+, Chrome+

Even though the `scrollLeft` and `scrollTop` properties of the `body` object are the same as those for generic HTML element objects, they play an important role in determining the placement of positioned elements (described more fully in Chapter 43). Because the mouse event and element position properties tend to be relative to the visible content region of the browser window, you must take the scrolling values of the `document.body` object into account when assigning an absolute position. Values for both of these properties are integers representing pixels.

Example

Listing 29-18 is an unusual construction that creates a frameset and the content for each of the two frames, all within a single HTML document. In the left frame of the frameset are two fields that are ready to show the pixel values of the right frame's `xOffset` and `yOffset` properties. The content of the right frame is a 30-row table of fixed width (800 pixels). Mouse-click events are captured by the document level (see Chapter 32), allowing you to click any table or cell border, or outside the table, to trigger the `showOffsets()` function in the right frame. That function is a simple script that displays the page offset values in their respective fields in the left frame.

LISTING 29-18

Determining Scroll Values

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Master of all Windows</title>
      <script type="text/javascript">
         function leftFrame()
         {
            var output = "<html><body><h3>Body Scroll Values<\/h3><hr />\n";
            output += "<form>body.scrollLeft:<input type='text' ";
            output += "name='xOffset' size=4 /><br />\n";
            output += "body.scrollTop:<input type='text' ";
            output += "name='yOffset' size=4 /><br />\n";
            output += "<\/form><\/body><\/html>";
```

```
         return output;
      }

      function rightFrame()
      {
         var output = "<html><head><script type='text/javascript'>\n";
         output += "function showOffsets() {\n";
         output += "parent.readout.document.forms[0].xOffset.value ";
         output += "= document.body.scrollLeft\n";
         output += "parent.readout.document.forms[0].yOffset.value ";
         output += "= document.body.scrollTop\n}\n";
         output += "document.onclick = showOffsets\n";
         output += "<\/script><\/head><body><h3>Content Page<\/h3>\n";
         output += "Scroll this frame and click on a table border to view ";
         output += "page offset values.<br /><hr />\n";
         output += "<table border=5 width=800>";
         var oneRow = "<td>Cell 1<\/td><td>Cell 2<\/td><td>Cell ";
         oneRow += "3<\/td><td>Cell 4<\/td><td>Cell 5<\/td>";
         for (var i = 1; i <= 30; i++)
         {
            output += "<tr><td><b>Row " + i + "<\/b><\/td>" + oneRow + "<\/tr>";
         }
         output += "<\/table><\/body><\/html>";
         return output;
      }
   </script>
 </head>
 <frameset cols="30%,70%">
   <frame name="readout" src="javascript:parent.leftFrame()" />
   <frame name="display" src="javascript:parent.rightFrame()" />
 </frameset>
</html>
```

Related Items: `window.pageXOffset`, `window.pageYOffset` properties

`text`

(See `aLink`)

`topMargin`

(See `bottomMargin`)

`vLink`

(See `aLink`)

Methods

`createControlRange()`

Returns: Array

Compatibility: WinIE5+, MacIE-, NN-, Moz-, Safari-, Opera-, Chrome-

This method creates a control range in WinIE5+ browsers. Control ranges are used for control-based selection, as opposed to text-based selection made possible by text ranges. The method only applies to documents in edit mode. In regular document view mode, the `createControlRange()` method returns an empty array.

`createTextRange()`

Returns: Object

Compatibility: WinIE4+, MacIE-, NN-, Moz-, Safari-, Opera-, Chrome-

The `body` element object is the most common object used to generate a `TextRange` object in IE4+, especially when the text you are about to manipulate is part of the document's body text. The initial `TextRange` object returned from the `createTextRange()` method encompasses the entire `body` element's HTML and body text. Further action on the returned object is required to set the start and end point of the range. See the discussion of the `TextRange` object in Chapter 33 for more details.

Example

See Listing 30-10 for an example of the `createTextRange()` method in action.

Related Item: `TextRange` object (see Chapter 33, "Body Text Objects")

`doScroll(["`*`scrollAction`*`"])`

Returns: Nothing

Compatibility: WinIE5+, MacIE-, NN-, Moz-, Safari-, Opera-, Chrome-

Use the `doScroll()` method to simulate user action on the scroll bars inside a window or frame that holds the current document. This method comes in handy if you are creating your own scroll bars in place of the standard system scroll bars. Scrolling is instantaneous, however, rather than animated, even if the Display control panel is set for animated scrolling. The parameter for this method is one of the string constant values shown in Table 29-4. Occasionally, in practice, the longer scroll action names more closely simulate an actual click on the scroll bar component, whereas the shortcut versions may scroll at a slightly different increment.

Unlike scrolling to a specific pixel location (by setting the `body` element's `scrollTop` and `scrollLeft` properties), the `doScroll()` method depends entirely on the spatial relationship between the body content and the window or frame size. Also, the `doScroll()` method triggers the `onscroll` event handler for the `body` element object.

Be aware that scripted modifications to body content can alter these spatial relationships. IE is prone to being sluggish in updating all of its internal dimensions after content has been altered. Should you attempt to invoke the `doScroll()` method after such a layout modification, the scroll may not be performed as expected. You may find the common trick of using `setTimeout()` to delay the invocation of the `doScroll()` method by a fraction of a second.

Example

Use The Evaluator (Chapter 4, "JavaScript Essentials") to experiment with the `doScroll()` method in IE5+. Size the browser window so that at least the vertical scroll bar is active (meaning it has a

thumb region). Enter the following statement into the top text field and press the Enter key a few times to simulate clicking the PgDn key:

```
document.body.doScroll()
```

Return to the top of the page, and now do the same for scrolling by the increment of the scroll bar down arrow:

```
document.body.doScroll("down")
```

TABLE 29-4

document.body.doScroll() Parameters

Long Parameter	Short Parameter	Scroll Action Simulates
`scrollbarDown`	`down`	Clicking the down arrow.
`scrollbarHThumb`	n/a	Clicking the horizontal scroll bar thumb (no scrolling action).
`scrollbarLeft`	`left`	Clicking the left arrow.
`scrollbarPageDown`	`pageDown`	Clicking the page down area or pressing PgDn (default).
`scrollbarPageLeft`	`pageLeft`	Clicking the page left area.
`scrollbarPageRight`	`pageRight`	Clicking the page right area.
`scrollbarPageUp`	`pageUp`	Clicking the page up area or pressing PgUp.
`scrollbarRight`	`right`	Clicking the right arrow.
`scrollbarUp`	`up`	Clicking the up arrow.
`scrollbarVThumb`	n/a	Clicking the vertical scroll bar thumb (no scrolling action).

You can also experiment with upward scrolling. Enter the desired statement in the top text field and leave the text cursor in the field. Manually scroll to the bottom of the page and then press Enter to activate the command.

Related Items: `body.scroll`, `body.scrollTop`, `body.scrollLeft` properties; `window.scroll()`, `window.scrollBy()`, `window.scrollTo()` methods

Event handlers

onafterprint
onbeforeprint
onscroll

(See these event handlers for the `window` object, Chapter 27, “Window and Frame Objects.”)

TreeWalker Object

Property	Method	Event Handler
currentNode	firstChild()	
expandEntityReference	lastChild()	
filter	nextNode()	
root	nextSibling()	
whatToShow	parentNode()	
	previousNode()	
	previousSibling()	

Syntax

Creating a `TreeWalker` object:

```
var treewalk = document.createTreeWalker(document, whatToShow,
   filterFunction, entityRefExpansion);
```

Accessing `TreeWalker` object properties and methods:

```
TreeWalker.property | method([parameters])
```

Compatibility: WinIE-, MacIE-, NN7+, Moz+, Safari+, Opera+, Chrome+

About this object

The `TreeWalker` object serves as a container for a list of nodes that meet the criteria defined by the `document.createTreeWalker()` method, which is used to create the object. The list of nodes contained by a `TreeWalker` object conforms to the same hierarchical structure of the document from which they are referenced. The `TreeWalker` object provides a means of navigating through this list of nodes based on their inherent tree-like structure.

You can think of the `TreeWalker` object as somewhat of an iterator object since its main purpose is to provide a means of stepping through nodes in a list. However, in this case the list is a hierarchical tree, as opposed to a linear list. The `TreeWalker` object maintains a pointer inside the list of nodes that always points to the current node. Whenever you navigate through the list using the `TreeWalker` object, the navigation is always relative to the pointer. For example, referencing the previous or next node through calls to the `previousNode()` or `nextNode()` methods depends upon the current position of the node pointer in the tree.

Use the `document.createTreeWalker()` method to create a `TreeWalker` object for a particular document. This method requires a user function that serves as a filter for nodes selected to be part of the tree. A reference to the function is the third parameter of the method call. The return value of this user function can be one of three constant values, which indicate the status of the current node: `NodeFilter.FILTER_ACCEPT`, `NodeFilter.FILTER_REJECT`,

or `NodeFilter.FILTER_SKIP`. The difference between `NodeFilter.FILTER_REJECT` and `NodeFilter.FILTER_SKIP` is that descendents of skipped nodes may still qualify as part of the tree, whereas rejected nodes and their descendents are excluded altogether. Following is an example of a user function you could use to create a `TreeWalker` object:

```
function ratingAttrFilter(node)
{
   if (node.hasAttribute("rating"))
   {
      return NodeFilter.FILTER_ACCEPT;
   }
   return NodeFilter.FILTER_REJECT;
}
```

In this example function, only nodes containing an attribute named `rating` are allowed through the filter, which means that only those nodes will get added to the list (tree). With this function in place, you then call the `document.createTreeWalker()` method to create the `TreeWalker` object:

```
var myTreeWalker = document.createTreeWalker(document, NodeFilter.SHOW_ELEMENT,
                   ratingAttrFilter, false);
```

Now that the `TreeWalker` object is created, you can use its properties and methods to access individual nodes and navigate through the list.

Properties

currentNode

Value: Node reference Read/Write

Compatibility: WinIE-, MacIE-, NN7+, Moz+, Safari+, Opera+, Chrome+

The `currentNode` property returns a reference to the current node, which sits at the location of the tree's node pointer. Although you can use the `currentNode` property to access the current node, you can also use it to set the current node.

Example

To assign a node to the current position in the tree, just create an assignment statement using the `currentNode` property:

```
myTreeWalker.currentNode = document.getElementById("info");
```

Related Item: `root` property

expandEntityReference
filter
root
whatToShow

Value: See text Read-Only

Compatibility: WinIE-, MacIE-, NN7+, Moz+, Safari+, Opera+, Chrome+

These properties reflect the parameter values passed into the `document.createTreeWalker()` method upon the creation of the `TreeWalker` object.

Related Item: `document.createTreeWalker()` method

Methods

firstChild()
lastChild()
nextSibling()
parentNode()
previousSibling()

Returns: Node reference

Compatibility: WinIE-, MacIE-, NN7+, Moz+, Safari+, Opera+, Chrome+

These methods return references to nodes within the hierarchy of the tree-like list of nodes contained by the `TreeWalker` object. There is a parent-child relationship among all of the nodes in the tree, and these functions are used to obtain node references based upon this relationship. The node pointer within the tree moves to the new node whenever you use one of these methods to navigate to a given node. This means you can access the new node as the current node after calling one of these navigation methods.

Example

The following code shows how to obtain the tag name of the parent node of the current node in the `TreeWalker` object:

```
if (myTreeWalker.parentNode())
{
   var parentTag = myTreeWalker.currentNode.tagName;
}
```

Related Items: `nextNode()`, `previousNode()` methods

nextNode()
previousNode()

Returns: Node reference

Compatibility: WinIE-, MacIE-, NN7+, Moz+, Safari+, Opera+, Chrome+

The `nextNode()` and `previousNode()` methods navigate back and forth in the list of nodes contained by the `TreeWalker` object. It's important to note that these methods operate on the node list as if it had been flattened from a tree into a linear sequence of nodes. Both methods move the internal node pointer to the next or previous node, respectively.

Example

The following code demonstrates both the node filter function and a typical function you could use to display (in a series of alert windows, perhaps for debugging purposes) the IDs of all elements

inside the body that have `id` attributes assigned. The `nextNode()` method is called first to advance the `TreeWalker`'s node pointer to the first node of the collection, and then iteratively (inside a `do-while` construction) to obtain the next node that passes the node filter's test.

```
function idFilter(node)
{
   if (node.hasAttribute("id"))
   {
      return NodeFilter.FILTER_ACCEPT;
   }
   return NodeFilter.FILTER_SKIP;
}

function showIds()
{
   var tw =
   document.createTreeWalker(document.body, NodeFilter.SHOW_ELEMENT,
      idFilter, false);
   // make sure TreeWalker contains at least one node, and go to it if true
   if (tw.nextNode())
   {
      do
      {
         alert(tw.currentNode.id);
      } while (tw.nextNode());
   }
}
```

Related Item: `parentNode()` method

CHAPTER

30

Link and Anchor Objects

IN THIS CHAPTER

Differences among link, anchor, and a element objects

Scripting a link to invoke a script function

Scripting a link to swap an image on mouse rollovers

The Web is based on the notion that the world's information can be strung together by way of the *hyperlink* — the clickable hunk of text or image that enables an inquisitive reader to navigate to a further explanation or related material. Of all the document objects you work with in JavaScript, the link is the one that makes that connection. Anchors provide guideposts to specific locations within documents.

As scriptable objects going back to the first scriptable browsers, links and anchors are comparatively simple devices. But this simplicity belies their significance in the entire scheme of the Web. Under script control, links can be far more powerful than mere tethers to locations on the Web.

In modern browsers, the notion of separating links and anchors as similar yet distinctly different objects begins to fade. The association of the word *link* with objects is potentially confused by the newer browsers' recognition of the `link` element (see Chapter 40, "HTML Directive Objects"), which has an entirely different purpose as a scriptable object. Taking the place of the anchor and link objects is an HTML element object representing the element created by the `<a>` tag. As an element object, the `a` element assumes all the properties, methods, and event handlers are associated with all HTML element objects in modern object models. To begin making that transition, this chapter treats all three types of objects at the same time.

Anchor, Link, and a Element Objects

For HTML element properties, methods, and event handlers, see Chapter 26, "Generic HTML Element Objects."

Properties	Methods	Event Handlers
charset		
coords		
hash		
host		
hostname		
href		
hreflang		
methods		
mimeType		
name		
nameProp		
pathname		
port		
protocol		
rel		
rev		
search		
shape		
target		
type		
urn		

Syntax

Accessing Link Object Properties:

```
(all)      [window.]document.links[index].property
```

Accessing a element object properties:

```
(IE4+)     [window.]document.all.elemID.property | method([parameters])
(IE5+/W3C) [window.]document.getElementById("elemID").property |
    method([parameters])
```

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

About this object

A little scripting history can help you understand where the link and anchor objects came from and how the `a` element object evolved from them.

Using the terminology of the original object model, the anchor and link objects are both created in the object model from the `<a>` tag. What distinguishes a link from an anchor is the presence of the `href` attribute in the tag. Without an `href` attribute, the element is an anchor object, which has only a single property (`name`) in modern browsers. A link, on the other hand, is much more alive as an object — all because of the inclusion of an `href` attribute, which usually points to a URL to load into a window or frame.

When object models treat HTML elements as objects, both the anchor and link objects are subsumed by the `a` element object. Even so, one important characteristic from the original object holds true: All `a` element objects that behave as link objects (by virtue of the presence of an `href` attribute) are members of the `document.links` property array. Therefore, if your scripts need to inspect or modify properties of all link objects on a page, they can do so by way of a `for` loop through the array of link objects. This is true even if you script solely for modern browsers and want to, say, change a style attribute of all links (for example, change their `style.textDecoration` property from `none` to `underline`). The fact that the same element can have different behaviors depending on the existence of one attribute makes us think of the `a` element object as potentially two different animals. Therefore, when you see separate references to the link and anchor objects, you know the distinction between the two is important.

Scripting newcomers are often confused about the purpose of the `target` attribute of an `a` element when they want a scripted link to act on a different frame or window. Under plain HTML, the `target` attribute points to the frame or window into which the new document (the one assigned to the `href` attribute) is to load, leaving the current window or frame intact. But if you intend to use event handlers to navigate (by setting the `location.href` property), the `target` attribute does not apply to the scripted action. Instead, assign the new URL to the `location.href` property of the desired frame or window. For example, if one frame contains a table of contents consisting entirely of links, the `onclick` event handlers of those links can load other pages into the `main` frame by assigning the URL to the `parent.main.location.href` property. You must also cancel the default behavior of any link, as described in the discussion of the generic `onclick` event handler in Chapter 26.

When you want a click of the link (whether the link consists of text or an image) to initiate an action without actually navigating to another URL, you can use a special technique — the `javascript:` pseudo-URL — to direct the URL to a JavaScript function. The URL `javascript:functionName()` is a valid parameter for the `href` attribute (and not just in the link object). You can also add a special `void` operator that guarantees that the called function does not trigger any true linking action (`href="javascript: void someFunction()"`). Specifying an empty string for the `href` attribute yields an FTP-like file listing for the client computer — an undesirable artifact. Don't forget, too, that if the URL leads to a type of file that initiates a browser helper application (for example, playing a QuickTime movie), the helper app or plug-in loads and plays without changing the page in the browser window.

Note

Usage of the `javascript:` pseudo-URL is controversial. There is no published industry standard that supports it, even though most browsers do. It is also unfriendly to users who visit the page with scripting disabled or unavailable (for example, browsers designed for visually impaired users), because the links won't do anything, leading to frustration. You should also be aware that search engines won't follow these types of links when they work their way through a site. ■

A single link can change the content of more than one frame at the same time with the help of JavaScript. If you want only JavaScript-enabled browsers to act on such links, one approach is to use a `javascript:` pseudo-URL to invoke a function that changes the `location.href` properties of multiple frames. For example, consider the following function, which changes the content of two frames:

```
function navFrames(url1, url2)
{
   parent.product.location.href = url1;
   parent.accessories.location.href = url2;
}
```

Then you can have a `javascript:` pseudo-URL invoke this multipurpose function, and pass the specifics for the link as parameters:

```
<a href="javascript: void navFrames('products/gizmo344.html',
 'access/access344.html')">Deluxe Super Gizmo</a>
```

Or, if you want one link to do something for everyone, but something extra for JavaScript-enabled browsers (an approach that is desirable when designing a page for accessibility), you can combine the standard link behavior with an `onclick` event handler to take care of both situations:

```
function setAccessFrame(url)
{
   parent.accessories.location.href = url;
}
...
<a href="products/gizmo344.html" target="product"
onclick="setAccessFrame('access/access344.html')">Deluxe Super Gizmo</a>
```

Note

The property assignment event handling technique in the previous example is a deliberate simplification to make the code more readable. It is generally better to use the more modern approach of binding events using the `addEventListener()` (NN6+/Moz/W3C) or `attachEvent()` (IE5+) method. A modern cross-browser event handling technique is explained in detail in Chapter 32, "Event Objects." ■

Notice here that the `target` attribute is necessary for the standard link behavior, whereas the script assigns a URL to a frame's `location.href` property.

One additional technique allows a single link tag to operate for both scriptable and nonscriptable browsers. For nonscriptable browsers, establish a genuine URL to navigate from the link. Then make sure that the link's `onclick` event handler evaluates to `return false` or cancels the default action. At click time, a scriptable browser executes the event handler and ignores the `href` attribute; a non-scriptable browser ignores the event handler and follows the link. See the discussion of the generic `onclick` event handler in Chapter 26 for more details.

Properties

charset

Value: String Read/Write

Compatibility: WinIE6+, MacIE6+, NN6+, Moz+, Safari+, Opera+, Chrome+

The charset property represents the HTML 4 charset attribute of an a element. It advises the browser of the character set used by the document to which the href attribute points. The value is a string of one of the character set codes from the registry at http://www.iana.org/assignments/character-sets. The most commonly used character set on the Web is called ISO-8859-5.

coords
shape

Value: Strings Read/Write

Compatibility: WinIE6+, MacIE6+, NN6+, Moz+, Safari+, Opera+, Chrome+

HTML 4 provides specifications for a elements that accommodate different shapes (rect, circle, and poly) and coordinates when the link surrounds an image. Although the coords and shape properties are present for a element objects in all W3C DOM–compatible browsers, active support for the feature is not present in NN6.

hash
host
hostname
pathname
port
protocol
search

Value: Strings Read/Write

Compatibility: WinIE3+, MacIE3+, NN2+, Moz+, Safari+, Opera+, Chrome+

This large set of properties is identical to the same-named properties of the location object (see Chapter 28, "Location and History Objects"). All properties are components of the URL that is assigned to the link object's href attribute. Although none of these properties appears in the W3C DOM specification for the a element object, the properties survive in modern browsers for backward compatibility. If you want to script the change of the destination for a link, try modifying the value of the object's href property, rather than modifying individual components of the URL.

Related Item: location object

href

Value: String Read/Write

Compatibility: WinIE3+, MacIE3+, NN2+, Moz+, Safari+, Opera+, Chrome+

The href property (included in the W3C DOM) is the URL of the destination of an a element equipped to act as a link. URLs can be relative or absolute.

In W3C DOM–compatible browsers, you can turn an anchor object into a link object by assigning a value to the href property, even if the a element has no href attribute in the HTML that loads from the server. Naturally, this conversion is temporary, and it lasts only as long as the page is loaded in the browser. When you assign a value to the href property of an a element that surrounds text, the text assumes the appearance of a link (either the default appearance or whatever style you assign to links).

Related Item: location object

hreflang

Value: String Read/Write

Compatibility: WinIE6+, MacIE6+, NN6+, Moz+, Safari+, Opera+, Chrome+

The `hreflang` property advises the browser (if the browser takes advantage of it) about the written language used for the content to which the `a` element's `href` attribute points. Values for this property must be in the form of the standard language codes (for example, `en-us` for U.S. English).

Methods

Value: String Read/Write

Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-, Opera-, Chrome-

The `Methods` property (note the uppercase *M*) represents the HTML 4 `methods` attribute for an `a` element. Values for this attribute and property serve as advisory instructions to the browser about which HTTP method(s) to use for accessing the destination document. This is a rare case in which an HTML 4 attribute is not echoed in the W3C DOM. In any case, although IE4+ supports the property, the IE browsers do nothing special with the information.

mimeType

Value: String Read-Only

Compatibility: WinIE4+, MacIE-, NN-, Moz-, Safari-, Opera-, Chrome-

This property is used to obtain the MIME type of the document linked to by the `a` element. The HTML 4 and W3C DOM specifications define a `type` attribute and `type` property, instead of `mimeType`. The property is a read-only property and, therefore, has no control over the MIME type of the destination document. Interestingly, the IE8 MSDN documentation for the `a` object does not list this attribute, although IE8 does continue to support it.

Related Item: `a.type` property

name

Value: String Read/Write

Compatibility: WinIE3+, MacIE3+, NN2+, Moz+, Safari+, Opera+, Chrome+

Although a `name` attribute is optional for an `a` element serving solely as a link object, it is required for an anchor object. This value is exposed to scripting via the `name` property. Although it is unlikely that you will need to change the value by scripting, you can use this property as a way to identify a link object from among the `document.links` arrays in a repeat loop. For example:

```
for (var i = 0; i < document.links.length; i++)
{
   if (document.links[i].name == "bottom"
   {
      // statements dealing with the link named "bottom"
   }
}
```

If this code makes it inside the `if` clause, you know you've found a link with the name `bottom`.

nameProp

Value: String Read-Only

Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-

The IE-specific `nameProp` property is a convenience property that retrieves the segment of the `href` to the right of the rightmost forward slash character of the URL. Most typically, this value is the name of the file from a URL. But if the URL also includes a port number, that number is returned as part of the `nameProp` value.

rel
rev

Value: String Read/Write

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

The `rel` and `rev` properties define relationships in the forward and back directions with respect to the destination document of the `a` element. In other words, you're describing how a link relates to the document to which it points, as well as how the document relates back. For example, in a table-of-contents page, each link to a chapter might have its `rel` attribute set to `chapter`, whereas its `rev` attribute might be set to `contents`. Browsers have yet to exploit most of the potential of these attributes and properties.

A long list of values is predefined for these properties, based on the corresponding attribute values specified in HTML 4. If the browser does nothing with a particular value, the value is ignored. You can string together multiple values in a space-delimited list inside a single string. Accepted values are as follows:

alternate	contents	index	start
appendix	copyright	next	stylesheet
bookmark	glossary	prev	subsection
chapter	help	section	

target

Value: String Read/Write

Compatibility: WinIE3+, MacIE3+, NN2+, Moz+, Safari+, Opera+, Chrome+

An important property of the link object is the `target`. This value reflects the window name supplied to the `target` attribute in the `a` element.

You can temporarily change the target for a link. But as with most transient object properties, the setting does not survive soft reloads. Rather than alter the target this way, you can safely force the target change by letting the `href` attribute call a `javascript:functionName()` psuedo-URL in which the function assigns a document to the desired `window.location`. If you have done extensive HTML authoring before, you will find it hard to break the habit of relying on the `target` attribute.

Another drawback to the `target` attribute is the fact that it isn't supported by the strict XHTML DTD. So if you develop XHTML pages that must validate with the strict DTD, you will not be able

to include a `target` attribute in your `<a>` tags. Instead, use the page's `onload` event handler or the `a` element's `onclick` event handler to invoke a function that assigns the desired value to the `target` property. In this case, you are using a JavaScript property to sidestep a limitation associated with an HTML attribute.

Related Item: `document.links` property

type

Value: String　　　　Read/Write

Compatibility: WinIE6+, MacIE6+, NN6+, Moz+, Safari+, Opera+, Chrome+

The `type` property represents the HTML 4 `type` attribute, which specifies the MIME type for the content of the destination document to which the element's `href` attribute points. This is primarily an advisory property for browsers that wish to, say, display different cursor styles based on the anticipated type of content at the other end of the link. Thus far, browsers do not take advantage of this feature. However, you can assign MIME type values to the attribute (for example, `video/mpeg`) and let scripts read those values for making style changes to the link text after the page loads. IE4+ also implements a similar property in the `mimeType` property.

Related Item: `a.mimeType` property

urn

Value: String　　　　Read/Write

Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-, Opera-, Chrome-

The `urn` property represents the IE-specific `URN` attribute, which enables authors to use a URN (Uniform Resource Name) for the destination of the `a` element. (See `http://www.ietf.org/rfc/rfc2141.txt` for information about URNs.) This property is not in common use.

CHAPTER

31

Image, Area, Map, and Canvas Objects

IN THIS CHAPTER

How to pre-cache images

Swapping images after a document loads

Creating interactive, client-side image maps

Drawing vector graphics with a canvas

For modern web browsers, images and areas — those items created by the `<img>` and `<area>` tags — are first-class objects that you can script for enhanced interactivity. You can swap the image displayed in an `<img>` tag with other images, perhaps to show the highlighting of an icon button when the cursor rolls atop it. And with scriptable client-side area maps, pages can be smarter about how they respond to users' clicks on image regions.

One further benefit afforded scripters is that they can preload images into the browser's image cache as the page loads. With cached images, the user experiences no delay when the first swap occurs. The need for this capability has diminished slightly with higher bandwidth connections, but it still isn't a bad idea for those users who still rely on connections with speed limitations.

New on the graphical JavaScript scene is the notion of a canvas, which is a graphical region that you can use to carry out graphics operations via JavaScript code. A few browsers already support canvases, so you can get started tinkering with them now.

Image and img Element Objects

For HTML element properties, methods, and event handlers, see Chapter 26, "Generic HTML Element Objects."

Properties	Methods	Event Handlers
`align`		`onabort`
`alt`		`onerror`
`border`		`onload`
`complete`		

Properties	Methods	Event Handlers
dynsrc		
fileCreatedDate		
fileModifiedDate		
fileSize		
fileUpdatedDate		
height		
href		
hspace		
isMap		
loop		
longDesc		
lowsrc		
mimeType		
name		
nameProp		
naturalHeight		
naturalWidth		
protocol		
src		
start		
useMap		
vspace		
width		
x		
y		

Syntax

Creating an `Image` object:

```
imageObject = new Image([pixelWidth, pixelHeight]);
```

Accessing `img` element and `Image` object properties or methods:

```
(NN3+/IE4+)   [window.]document.imageName. property | method([parameters])
(NN3+/IE4+)   [window.]document.images[index]. property | method([parameters])
(NN3+/IE4+)   [window.]document.images["imageName"]. property |
              method([parameters])
(IE4+)        [window.]document.all.elemID.property | method([parameters])
(IE5+/W3C)    [window.]document.getElementById("elemID").property |
              method([parameters])
```

Compatibility: WinIE4+, MacIE4+, NN3+, Moz+, Safari+, Opera+, Chrome+

About this object

Before getting into detail about images as objects, it's important to understand the distinction between instances of the static `Image` object and `img` element objects. The former exist only in the browser's memory without showing anything to the user; the latter are the elements on the page generated via the `<img>` (or nonsanctioned, but accepted, `<image>`) tag. Scripts use instances of the `Image` object to pre-cache images for a page, but `Image` object instances obviously have fewer applicable properties, methods, and event handlers because they are neither visible on the page nor influenced by tag attributes.

The primary advantage of treating `img` elements as objects is that scripts can change the image that occupies the `img` object's space on the page, even after the document has loaded and displayed an initial image. The key to this scriptability is the `src` property of an image.

In a typical scenario, a page loads with an initial image. That image's tags specify any of the extra attributes, such as `height` and `width` (which help speed the rendering of the page), and specify whether the image uses a client-side image map to make it interactive. (See the `area` object later in this chapter.) As the user spends time on the page, the image can then change (perhaps in response to user action or some timed event in the script), replacing the original image with a new one in the same space. In legacy browsers that support the `img` element object, the height and width of the initial image that loads into the element establishes a fixed-sized rectangular space for the image. Attempts to fit an image of another size into that space forces the image to scale (up or down, as the case may be) to fit the rectangle. But in modern browsers (IE4+/Moz/W3C), a change in the image's size is reflected by an automatic reflow of the page content around the different size. Of course, you might not consider this desirable and choose to pre-size your images to the same size to avoid the automatic reflow of the page content.

The benefit of separate instances of the `Image` object is that a script can create a virtual image to hold a preloaded image. (The image is loaded into the image cache but the browser does not display the image.) The hope is that one or more unseen images will load into memory while the user is busy reading the page or waiting for the page to download. Then, in response to user action on the page, an image can change instantaneously rather than forcing the user to wait for the image to load on demand.

To preload an image, begin by assigning a new, empty `Image` object to a global variable. The new image is created via the constructor function available to the `Image` object:

```
var imageVariable = new Image(width, height);
```

You help the browser allocate memory for the image if you provide the pixel height and width of the pre-cached image as parameters to the constructor function. All that this statement does is create an object in memory whose properties are all empty. To force the browser to load the image into the cache, assign an image file URL to the object's `src` property:

```
var oneImage = new Image(55,68);
oneImage.src = "neatImage.gif";
```

As this image loads, you see the progress in the status bar, just like you would with any image. Later, assign the `src` property of this stored image to the `src` property of the `img` element object that appears on the page:

```
document.images["someImage"].src = oneImage.src;
```

Depending on the type and size of image, you will be amazed at the speedy response of this kind of loading. With small-palette graphics, the image displays instantaneously.

A popular user-interface technique is to change the appearance of an image that represents a clickable button when the user rolls the mouse pointer atop the art. This action assumes that a mouse event fires on an element associated with the object. Image rollovers are most commonly accomplished in two different image states: normal and highlighted. But you may want to increase the number of states to more closely simulate the way clickable buttons work in application programs. In some instances, a third state signifies that the button is switched on. For example, if you use rollovers in a frame for navigational purposes, and the user clicks a button to navigate to the Products area, that button stays selected but in a different style than the rollover highlights. Some designers go one step further by providing a fourth state that appears briefly when the user mouses down an image. Each one of these states requires the download of yet another image, so you have to gauge the effect of the results against the delay in loading the page.

The speed with which image swapping takes place may lead you to consider using this approach for animation. Though this approach may be practical for brief bursts of animation, the many other ways of introducing animation to your web page (such as via GIF89a-standard images, Flash animations, Java applets, and a variety of plug-ins) produce animation that offers better speed control. In fact, swapping preloaded JavaScript image objects for some cartoon-like animations may be too fast. You can build a delay mechanism around the `setInterval()` method, but the precise timing between frames varies with client processor performance.

All browsers that implement the `img` element object also implement the `document.images` array. You can (and probably should) use the availability of this array as a conditional switch before any script statements that work with the `img` element or `Image` object. The construction to use is as follows:

```
if (document.images)
{
   // statements working with images as objects
}
```

Earlier browsers treat the absence of this array as the equivalent of `false` in the `if` clause's conditional statement.

Most of the properties discussed here mirror attributes of the `img` HTML element. For more details on the meanings and implications of attribute values on the rendered content, consult the HTML 4.01 specification (`http://www.w3.org/TR/html4/`), the HTML 5 specification (`http://www.w3.org/TR/html5/`), and Microsoft's extensions for IE (`http://msdn.microsoft.com/en-us/library/ms535259%28VS.85%29.aspx`).

Properties

align

Value: String Read/Write

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

The `align` property defines how the image is oriented in relation to surrounding text content. It is a double-duty property because you can use it to control the vertical or horizontal alignment, depending on the value (and whether the image is influenced by a `float` style attribute). Values are string constants, as follows:

```
absbottom    middle
absmiddle    right
baseline     texttop
bottom       top
left
```

The default alignment for an image is `bottom`. Increasingly, element alignment is handed over to style sheet control. (This is reflected in the corresponding `alignment` attribute's deprecation in HTML 4.01 and nonexistence in HTML 5.) In modern web pages, designers are encouraged to use style sheets, as opposed to element attributes, for presentation details such as alignment.

Listing 31-1 enables you to choose from the different `align` property values as they influence the layout of an image whose HTML is embedded inline with some other text. Resize the window to see different perspectives on word-wrapping on a page and their effects on the alignment choices. Not all browsers provide distinctive alignments for each choice, so experiment in multiple supported browsers.

LISTING 31-1

Testing an Image's align Property

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>img align Property</title>
      <script type="text/javascript">
      function setAlignment(sel)
      {
         document.getElementById("myIMG").align =
            sel.options[sel.selectedIndex].value;
      }
      </script>
   </head>
   <body>
      <h1>img align Property</h1>
      <hr />
      <form>
         Choose the image alignment: <select onchange="setAlignment(this)">
            <option value="absbottom">absbottom</option>
            <option value="absmiddle">absmiddle</option>
```

continued

LISTING 31-1 *(continued)*

```
          <option value="baseline">baseline</option>
          <option value="bottom" selected="selected">bottom</option>
          <option value="left">left</option>
          <option value="middle">middle</option>
          <option value="right">right</option>
          <option value="texttop">texttop</option>
          <option value="top">top</option>
        </select>
      </form>
      <hr />
      <p>Lorem ipsum dolor sit amet, consectetaur adipisicing elit, sed do
        eiusmod tempor incididunt ut labore et dolore magna aliqua.
        <img alt="image" id="myIMG" src="desk1.gif" height="90" width="120" />
        Ut enim adminim veniam, quis nostrud exercitation ullamco laboris nisi ut
        aliquip ex ea commodo consequat.</p>
    </body>
  </html>
```

Related Items: `text-align`, `float` style sheet attributes

alt

Value: String Read/Write

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

The `alt` property enables you to set or modify the text that the browser displays in the image's rectangular space (if height and width are specified in the tag) before the image downloads to the client. Also, if a browser has images turned off (or is incapable of displaying images), the `alt` text helps users identify what is normally displayed in that space. In IE, the `alt` text is displayed when you mouse over the image. You can modify this `alt` text even after the page loads.

Example

Use The Evaluator (Chapter 4, "JavaScript Essentials") to assign a string to the `alt` property of the `document.myIMG` image on the page. First, assign a nonexistent image to the `src` property to remove the existing image:

```
document.myIMG.src = "fred.gif"
```

Scroll down to the image, and you can see that the original image has been replaced with a space for the new, nonexistent, image. Now, assign a string to the `alt` property:

```
document.myIMG.alt = "Fred\'s face"
```

The extra backslash is required to escape the apostrophe inside the string. Scroll down to see the new `alt` text in the image space. This simple test in The Evaluator will work in most browsers. In some browsers, such as Firefox, it will appear to not work. Browsers such as Firefox support

the properties in question, but since the new image is nonexistent, it doesn't replace the original image; because the original image isn't broken, you won't see the effect of changing the alt property.

Related Item: title property

border

Value: Integer Read/Write

Compatibility: WinIE4+, MacIE4+, NN3+, Moz+, Safari+, Opera+, Chrome+

The border property defines the thickness in pixels of a border around an image. Remember that if you wrap an image inside an a element to make use of the mouse events (for rollovers and such), be sure to set the border=0 attribute of the <img> tag to prevent the browser from generating the usual link type of border around the image. Even though the default value of the attribute is zero, surrounding the image with an a element or attaching the image to a client-side image map puts a border around the image.

As with the alignment property, an element's border is increasingly handed over to style sheet control. (This is reflected in the corresponding border attribute's nonexistence in HTML 5.) In modern web pages, designers are encouraged to use style sheets as opposed to element attributes for presentation details such as borders.

Example

Feel free to experiment with the document.myIMG.border property for the image in The Evaluator (Chapter 4, "JavaScript Essentials") by assigning different integer values to the property.

Related Items: isMap, useMap properties

complete

Value: Boolean Read-Only

Compatibility: WinIE4+, MacIE4+, NN3+, Moz+, Safari+, Opera+, Chrome+

Sometimes you may want to make sure that an image is not still in the process of loading before allowing another process to take place. This situation is different from waiting for an image to load before triggering some other process (which you can do via the Image object's onload event handler). To verify that the img object displays a completed image, check for the Boolean value of the complete property. To verify that a particular image file has loaded, first find out whether the complete property is true; then compare the src property against the desired filename.

An image's complete property switches to true even if only the specified lowsrc image has finished loading. Do not rely on this property alone for determining whether the src image has loaded if both src and lowsrc attributes are specified in the <img> tag.

One of the best ways to use this property is in an if construction's conditional statement:

```
if (document.myImage.complete)
{
    // statements that work with document.myImage
}
```

***imageObject*.complete**

To experiment with the `image.complete` property, run Listing 31-2. Click the Load NASA Image button. As the image loads, click the Is "it loaded yet?" button to see the status of the `complete` property for the `Image` object. The value is `false` until the loading finishes; then, the value becomes `true`. We deliberately chose to load a large image directly from the NASA web site so that you would have plenty of time to click the "Is it loaded yet?" button several times in order to see the behavior of the `complete` property. You may have to quit and relaunch your browser between trials to clear the image from the cache (or empty the browser's memory cache) if you've been visiting the NASA web site's image gallery.

LISTING 31-2

Scripting image.complete

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>The Complete Property</title>
      <script type="text/javascript">
      function loadIt()
      {
         document.getElementById("result").value = "";
         document.getElementById("theImage").src =
           "http://www.nasa.gov/images/content/402217main_ssc2008-01b_full.jpg" ;
      }
      function checkLoad()
      {
         document.getElementById("result").value =
            document.getElementById("theImage").complete;
      }
      </script>
   </head>
   <body>
      <img id="theImage" alt="image" width="120" height="90" />
      <form>
         <input type="button" value="Load NASA image"
                onclick="loadIt();" />
         <p><input type="button" value="Is it loaded yet?"
                   onclick="checkLoad();" />
            <input id="result" type="text" />
         </p>
         <input type="reset" />
      </form>
   </body>
</html>
```

Note

The property assignment event handling technique in the previous example is a deliberate simplification to make the code more readable. It is generally better to use the more modern approach of binding events using the `addEventListener()` (NN6+/Moz/W3C) or `attachEvent()` (IE5+) methods. A modern cross-browser event handling technique is explained in detail in Chapter 32, "Events Objects." ■

Related Items: `img.src`, `img.lowsrc`, `img.readyState` properties; `onload` event handler

dynsrc

Value: URL string Read/Write

Compatibility: WinIE4-6, MacIE4+, NN-, Moz-, Safari-, Opera-, Chrome-

The `dynsrc` property is a URL to a video source file, which (in IE) you can play through an `img` element. You can turn a space devoted to a static image into a video viewer by assigning a URL of a valid video source (for example, an `.avi` or `.mpg` file) to the `dynsrc` property of the image element object. Unlike the `src` property of image objects, assigning a URL to the `dynsrc` property does not pre-cache the video.

You may experience buggy behavior in various IE versions when you assign a value to an image's `dynsrc` property after the `img` element renders a `.gif` or `.jpg` image. In WinIE5, the status bar indicates that the video file is still downloading, even though the download is complete. Clicking the Stop button has no effect. WinIE5.5+ may not even load the video file, leaving a blank space, although the property does store the value of the URL. MacIE5 changes between static and motion images with no problems, but playing the video file multiple times causes the `img` element to display black space beyond the element's rectangle.

Related Items: `img.loop`, `img.start` properties

fileCreatedDate
fileModifiedDate
fileUpdatedDate
fileSize

Value: String, Integer (`fileSize`) Read-Only

Compatibility: WinIE4+, MacIE5+, NN-, Moz-, Safari-, Opera-, Chrome-

These four IE-specific properties return information about the file displayed in the `img` element (whether it's a still or animated image). Three of the properties reveal the dates on which the current image's file was created, modified, and updated. For an unmodified file, its creation and modified dates are the same. The updated date of an image is the date on which the image file was last uploaded to the server; the `fileUpdatedDate` property is only supported on WinIE5.5+ and MacIE5. The `fileSize` property reveals the number of bytes of the file.

Date values returned for the first two properties are formatted differently between IE4 and IE5. The former provides a full readout of the day and date; the latter returns a format similar to mm/dd/yyyy. Note, however, that the values contain only the date and not the time. In any case, you can use the values as the parameter to a `new Date()` constructor function. This enables you to then use date calculations for such information as the number of days between the current day and the most recent modification.

Not all servers provide the proper date or size information about a file or in a format that IE can interpret. Test your implementation on the deployment server to ensure compatibility.

Also, be aware that these properties can be read-only for a file that is loaded in the browser. JavaScript by itself cannot get this information about files on the server.

Note

All of these file-related properties are present in the Mac version of IE, but the values are empty. ■

Example

These properties are similar to the same-named properties of the `document` object. You can see these properties in action in Listing 29-4. Make a copy of that listing, and supply an image before modifying the references from the `document` object to the `Image` object to see how these properties work with the `img` element object.

Or, just test them out one at a time using an existing image in the Evaluator (Chapter 4, "JavaScript Essentials"):

```
document.getElementById("myIMG").fileSize
```

Related Items: None

height
width

Value: Integer Read/Write (see text)

Compatibility: WinIE4+, MacIE4+, NN3+, Moz+, Safari+, Opera+, Chrome+

The `height` and `width` properties return and control the pixel height and width of an image object. The property is read/write in all modern browsers that support the `img` element object. However, the net effect of changing these properties varies from browser to browser. For example, if you adjust the `height` property of an image in Mozilla, the browser automatically scales the image within the same proportions as the original. But adjusting the `width` property has no effect on the `height` property. In IE7, the opposite effect is true in regard to `width` and `height`. Any time an image is scaled dynamically, unwanted pixelation can occur in the image, so modify an image's size with extreme care.

Example

Use The Evaluator (Chapter 4, "JavaScript Essentials") to experiment with the `height` and `width` properties. Begin retrieving the default values by entering the following two statements into the top text box:

```
document.myIMG.height
document.myIMG.width
```

Increase the height of the image from its default 90 to 180:

```
document.myIMG.height = 180
```

Next, exaggerate the width:

```
document.myIMG.width = 400
```

View the resulting image.

Related Items: `hspace`, `vspace` properties

`href`

(See `src` property)

`hspace`
`vspace`

Value: Integer Read/Write

Compatibility: WinIE4+, MacIE4+, NN3+, Moz+, Safari+, Opera+, Chrome+

The `hspace` and `vspace` properties control the pixel width of a transparent margin surrounding an image. Specifically, `hspace` controls the margins at the left and right of the image; `vspace` controls the top and bottom margins. Images, by default, have margins of zero pixels.

Example

Use The Evaluator (Chapter 4, "JavaScript Essentials") to experiment with the `hspace` and `vspace` properties. Begin by noticing that the image near the bottom of the page has no margins specified for it and is flush left with the page. Now assign a horizontal margin spacing of 30 pixels:

```
document.myIMG.hspace = 30
```

The image has shifted to the right by 30 pixels. An invisible margin also exists to the right of the image.

Related Items: `height`, `width` properties

`isMap`

Value: Boolean Read/Write

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+ Opera+, Chrome+

The `isMap` property enables you to set whether the image should act as a server-side image map. When set as a server-side image map, pixel coordinates of the click are passed as parameters to whatever link `href` surrounds the image. For client-side image maps, see the `useMap` property later in this chapter.

Example

The image in The Evaluator page is not defined as an image map. Thus, if you type the following statement into the top text box, the property returns `false`:

```
document.myIMG.isMap
```

Related Item: `img.useMap` property

`longDesc`

Value: URL string Read/Write

Compatibility: WinIE6+, MacIE5+, NN6+, Moz+, Safari+, Opera+, Chrome+

The `longDesc` property is a URL of a file that is intended to provide a detailed description of the image associated with the `img` element. Current browsers recognize this property, but do not do anything special with the information — whether specified by script or the `longdesc` attribute.

Related Item: `alt` property

loop

Value: Integer Read/Write

Compatibility: WinIE4-6, MacIE4+, NN-, Moz-, Safari-, Opera-, Chrome-

The `loop` property represents the number of times a video clip playing through the `img` element object should run. After the video plays that number of times, only the first frame of the video appears in the image area. The default value is `1`; but if you set the value to `-1`, the video plays continuously. Unfortunately, setting the property to `0` prior to assigning a URL to the `dynsrc` property does not prevent the movie from playing at least once (except on the Mac, as noted in the `dynsrc` property discussion earlier in this chapter).

Related Item: `dynsrc` property

lowsrc
lowSrc

Value: URL string Read/Write

Compatibility: WinIE4+, MacIE4+, NN3+, Moz-, Safari-, Opera-, Chrome-

For image files that take several seconds to load, modern browsers enable you to specify a lower-resolution image, or some other quick-loading placeholder, to stand in while the big image crawls to the browser. You assign this alternate image via the `lowsrc` attribute in the `<img>` tag. The attribute is reflected in the `lowsrc` property of an image object. While all modern browsers enable you to specify this property or its corresponding HTML attribute, only IE actually displays the specified image.

All compatible browsers recognize the all-lowercase version of this property. NN6 also recognizes a camelCase "S" version of the property, `lowSrc`.

Be aware that in some browsers if you assign a URL to the `lowsrc` attribute, the `complete` property switches to `true` and the `onLoad` event handler fires when the alternate file finishes loading: The browser does not wait for the main `src` file to load.

Example

See Listing 31-4 for the `Image` object's `onload` event handler to see how the source-related properties affect event processing.

Related Items: `img.src`, `img.complete` properties

mimeType

Value: String Read-Only

Compatibility: WinIE6+, MacIE-, NN-, Moz-, Safari-, Opera-, Chrome-

The `mimeType` property returns a plain-language description of the MIME type for the image, such as `JPEG Image` or `GIF Image`.

Example

You can use the `mimeType` property in Internet Explorer to determine the format of an image, as the following example demonstrates:

```
if (document.myIMG.mimeType.indexOf("JPEG") != -1) {
   // Carry out JPEG-specific processing
}
```

In this example, the `indexOf()` method is used to check for the presence of the phrase `"JPEG"` anywhere in the MIME type string. This works because the string returned in the `mimeType` property for JPEG images is `JPEG Image`.

Related Items: None

`name`

Value: Identifier string Read/Write

Compatibility: WinIE3+, MacIE3+, NN2+, Moz+, Safari+ Opera+, Chrome+

The `name` property returns the value assigned to the `name` attribute of an `img` element. Modern browsers allow you to use the ID of the element (`id` attribute) to reference the `img` element object via `document.all` (IE) and `document.getElementById()`. But references in the form of `document.imageName` and `document.images[imageName]` must use only the value assigned to the `name` attribute.

In some designs, it may be convenient to assign numerically sequenced names to `img` elements, such as `img1`, `img2`, and so on. As with any scriptable identifier, the name cannot begin with a numeric character. Rarely, if ever, will you need to change the name of an `img` element object.

Example

You can use The Evaluator (Chapter 4, "JavaScript Essentials") to examine the value returned by the `name` property of the image on that page. Enter the following statement into the top text box:

```
document.myIMG.name
```

Of course, this is redundant because the name is part of the reference to the object.

Related Item: `id` property

`nameProp`

Value: Filename string Read-Only

Compatibility: WinIE5+, MacIE-, NN-, Moz-, Safari-, Opera-, Chrome-

Unlike the `src` property, which returns a complete URL in IE, the IE `nameProp` property returns only the filename, exclusive of protocol and path. If your image-swapping script needs to read the name of the file currently assigned to the image (to determine which image to show next), the `nameProp` property makes it easier to get the actual filename without having to perform extensive parsing of the URL.

Example

You can use The Evaluator Sr. (Chapter 4, "JavaScript Essentials") to compare the results of the `src` and `nameProp` properties in WinIE5+. Enter each of the following statements into the top text box:

```
document.myIMG.src
document.myIMG.nameProp
```

Related Item: `img.src` property

naturalHeight
naturalWidth

Value: Integer Read-Only

Compatibility: WinIE-, MacIE-, NN6+, Moz+, Safari+, Opera-, Chrome+

The `naturalHeight` and `naturalWidth` properties return the unscaled height and width of the image, in pixels. These properties are useful in situations where script code or `img` element attributes have scaled an image and you wish to know the image's original size.

Example

Use The Evaluator (Chapter 4, "JavaScript Essentials") to experiment with the `naturalHeight` and `naturalWidth` properties in a Mozilla-based or WebKit-based browser. Begin retrieving the default values by entering the following statement into the top text box:

```
document.myIMG.width
```

Increase the width of the image from its default 120 to 200:

```
document.myIMG.width = 200
```

If you scroll down to the image, you see that the image has scaled in proportion. You can now find out the natural width of the original image by taking a look at the `naturalWidth` property:

```
document.myIMG.naturalWidth
```

The Evaluator will reveal 120 as the natural image width even though the image is currently scaled to 200.

Related Items: `img.height`, `img.width` properties

protocol

Value: String Read-Only

Compatibility: WinIE4+, MacIE5+, NN-, Moz-, Safari-, Opera-, Chrome-

The IE `protocol` property returns only the protocol portion of the complete URL returned by the `src` property. This allows your script, for example, to see if the image is sourced from a local hard drive or a web server. Values returned are not the actual protocol strings; rather, they are descriptions thereof: `HyperText Transfer Protocol` or `File Protocol`.

Example

You can use The Evaluator Sr. (Chapter 4, "JavaScript Essentials") to examine the `protocol` property of the image on the page. Enter the following statement into the top text box:

```
document.myIMG.protocol
```

Related Items: `img.src`, `img.nameProp` properties

src

Value: URL string Read/Write

Compatibility: WinIE4+, MacIE4+, NN3+, Moz+, Safari+, Opera+, Chrome+

The `src` property is the gateway to precaching images (in instances of the `Image` object that are stored in memory) and performing image swapping (in `img` element objects). Assigning a URL to the `src` property of an image object in memory causes the browser to load the image into the browser's cache (provided the user has the cache turned on). Assigning a URL to the `src` property of an `img` element object causes the element to display the new image. To take advantage of this powerful combination, you preload alternate versions of swappable images into image objects in memory and then assign the `src` property of the `Image` object to the `src` property of the desired `img` element object.

In legacy browsers, the size of the image defined by the original `img` element governs the rectangular space devoted to that image. An attempt to assign an image of a different size to that `img` element object causes the image to rescale to fit the rectangle (usually resulting in a distorted image). In all modern browsers, however, the `img` element object resizes itself to accommodate the image, and the page content reflows around the new size.

Note that when you read the `src` property, it returns a fully formed URL of the image file, including protocol and path. This often makes it inconvenient to let the name of the file guide your script in swapping images in a sequence of your choice. Some other mechanism (such as storing the current filename in a global variable) may be easier to work with (also see the WinIE5+ `nameProp` property).

Example

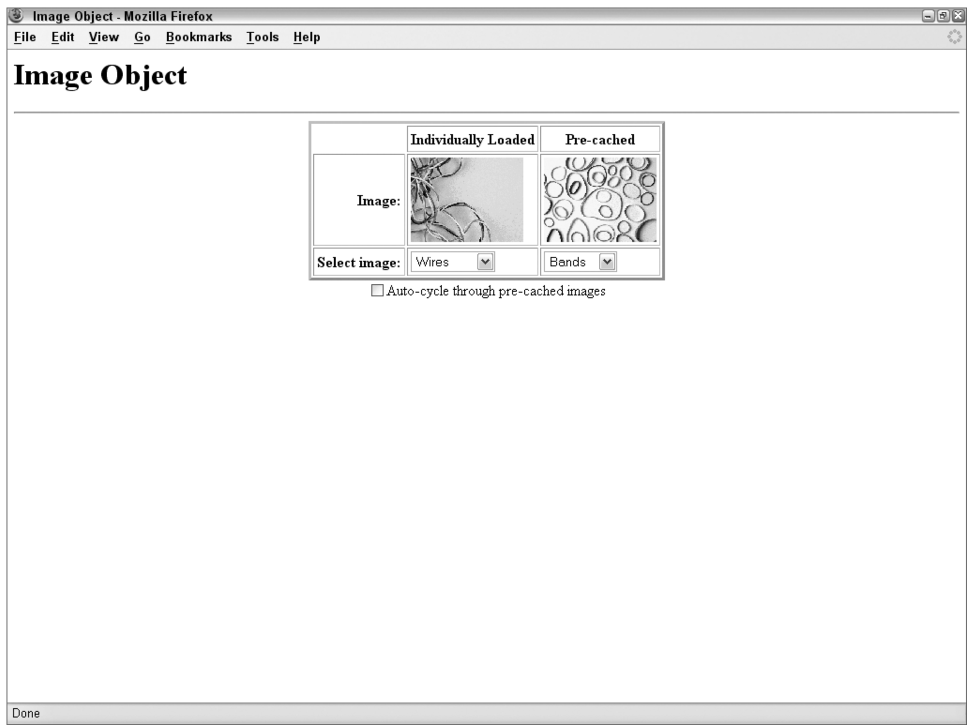
In the following example (see Listing 31-3), you see a few applications of image objects. Of prime importance is a comparison of how pre-cached versus images rendered on-the-fly feel to the user. As a bonus, you see an example of how to set a timer to automatically change the images displayed in an image object. This feature is a popular request among sites that display advertising banners or slide shows.

As the page loads, a global variable is handed an array of image objects. Entries of the array are assigned string names as index values (`"desk1"`, `"desk2"`, and so on). The intention is that these names ultimately will be used as addresses to the array entries. Each image object in the array has a URL assigned to it, which pre-caches the image.

The page (see Figure 31-1) includes two `img` elements: one that displays noncached images and one that displays cached images. Under each image is a `select` element that you can use to select one of four possible image files for each element. The `onchange` event handler for each `select` list invokes a different function to change the noncached (`loadIndividual()`) or cached (`loadCached()`) images. Both of these functions take as their single parameter a reference to the form that contains the `select` elements.

FIGURE 31-1

The image object demonstration page.



To cycle through images at five-second intervals, the `checkTimer()` function looks to see if the timer check box is selected. If so, the `selectedIndex` property of the cached image `select` control is copied and incremented (or reset to zero if the index is at the maximum value). The `select` element is adjusted, so you can now invoke the `loadCached()` function to read the currently selected item and set the image accordingly.

For some extra style points, the `<body>` tag includes an `onunload` event handler that invokes the `resetSelects()` function. This general-purpose function loops through all forms on the page and all elements within each form. For every `select` element, the `selectedIndex` property is reset to zero. Thus, if a user reloads the page, or returns to the page via the Back button, the images start in their original sequence. An `onload` event handler makes sure that the images are in sync with the `select` choices and the `checkTimer()` function is invoked with a five-second delay. Unless the timer check box is checked, however, the cached images don't cycle.

LISTING 31-3

A Scripted Image Object and Rotating Images

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Image Object</title>
      <script type="text/javascript">
      // global declaration for 'desk' images array
      var imageDB;
      // pre-cache the 'desk' images
      if (document.images)
      {
         // list array index names for convenience
         var deskImages = new Array("desk1", "desk2", "desk3", "desk4");
         // build image array and pre-cache them
         imageDB = new Array(4);
         for (var i = 0; i < imageDB.length ; i++)
         {
            imageDB[deskImages[i]] = new Image(120,90);
            imageDB[deskImages[i]].src = deskImages[i] + ".gif";
         }
      }
      // change image of 'individual' image
      function loadIndividual(form)
      {
         if (document.images)
         {
            var gifName =
                form.individual.options[form.individual.selectedIndex].value;
            document.getElementById("thumbnail1").src = gifName + ".gif";
         }
      }
      // change image of 'cached' image
      function loadCached(form)
      {
         if (document.images)
         {
            var gifIndex =
                form.cached.options[form.cached.selectedIndex].value;
            document.getElementById("thumbnail2").src = imageDB[gifIndex].src;
         }
      }
      // if switched on, cycle 'cached' image to next in queue
      function checkTimer()
```

continued

imageObject.src

LISTING 31-3 *(continued)*

```
   {
      if (document.images & document.Timer.timerBox.checked)
      {
         var gifIndex = document.selections.cached.selectedIndex;
            if (++gifIndex > imageDB.length - 1)
            {
               gifIndex = 0;
            }
         document.selections.cached.selectedIndex = gifIndex;
         loadCached(document.selections);
         var timeoutID = setTimeout("checkTimer()",5000);
      }
   }
   // reset form controls to defaults on unload
   function resetSelects()
   {
      for (var i = 0; i < document.forms.length; i++)
      {
         for (var j = 0; j < document.forms[i].elements.length; j++)
         {
            if (document.forms[i].elements[j].type == "select-one")
            {
               document.forms[i].elements[j].selectedIndex = 0;
            }
         }
      }
   }
   // get things rolling
   function init()
   {
      loadIndividual(document.selections);
      loadCached(document.selections);
      setTimeout("checkTimer()",5000);
   }
   </script>
</head>
<body onload="init()" onunload="resetSelects ()">
   <h1>Image Object</h1>
   <hr />
   <center>
      <table border="3" cellpadding="3">
         <tr>
            <th></th>
            <th>Individually Loaded</th>
            <th>Pre-cached</th>
         </tr>
         <tr>
            <td align="right"><b>Image:</b></td>
            <td><img alt="image" src="cpu1.gif" id="thumbnail1"
               height="90" width="120" /></td>
```

```
            <td><img alt="image" src="desk1.gif" id="thumbnail2"
               height="90" width="120" /></td>
         </tr>
         <tr>
            <td align="right"><b>Select image:</b></td>
            <form name="selections">
            <td><select name="individual"
               onchange="loadIndividual(this.form)">
                  <option value="cpu1">Wires</option>
                  <option value="cpu2">Keyboard</option>
                  <option value="cpu3">Discs</option>
                  <option value="cpu4">Cables</option>
               </select></td>
            <td><select name="cached" onchange="loadCached(this.form)">
               <option value="desk1">Bands</option>
               <option value="desk2">Clips</option>
               <option value="desk3">Lamp</option>
               <option value="desk4">Erasers</option>
            </select></td>
            </form>
         </tr>
      </table>
      <form name="Timer">
         <input type="checkbox" name="timerBox"
         onclick="checkTimer()" />Auto-cycle through pre-cached images
      </form>
    </center>
  </body>
</html>
```

Related Items: `img.lowsrc`, `img.nameProp` properties

start

Value: String Read/Write

Compatibility: WinIE4-6, MacIE4+, NN-, Moz-, Safari-, Opera-, Chrome-

The `start` property works in conjunction with video clips viewed through the `img` element in IE4+. By default, a clip starts playing (except on the Macintosh) when the image file opens. This follows the default setting of the `start` property: `"fileopen"`. Another recognized value is `"mouseover"`, which prevents the clip from running until the user rolls the mouse pointer atop the image.

Related Items: `img.dynsrc`, `img.loop` properties

useMap

Value: Identifier string Read/Write

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

The `useMap` property represents the `usemap` attribute of an `img` element, pointing to the name assigned to the `map` element in the page (see Listing 31-6). This `map` element contains the details

about the client-side image map (described later in this chapter). The value for the `useMap` property must include the hash mark that defines an internal HTML reference on the page. If you need to switch between two or more image maps for the same `img` element (for example, you swap images or the user is in a different mode), you can define multiple `map` elements with different names. Then, change the value of the `useMap` property for the `img` element object to associate a different map with the image.

Related Item: `isMap` property

vspace

(See `hspace`)

width

(See `height`)

x
y

Value: Integer Read-Only

Compatibility: WinIE-, MacIE-, NN4, Moz1+, Safari1+, Opera-, Chrome+

A script can retrieve the *x* and *y* coordinates of an `img` element (the top-left corner of the rectangular space occupied by the image) via the `x` and `y` properties. These properties are read-only. They are generally not used, in favor of the `offsetLeft` and `offsetTop` properties of any element, which are also supported in IE.

Related Items: `img.offsetLeft`, `img.offsetTop` properties; `img.scrollIntoView()`, `window.scrollTo()` methods

Event handlers

onabort
onerror

Compatibility: WinIE4+, MacIE4+, NN3+, Moz-, Safari-, Opera-, Chrome-

Your scripts may need to be proactive when a user clicks the Stop button while an image loads or when a network or server problem causes the image transfer to fail. Use the `onabort` event handler to activate a function in the event of a user clicking the Stop button; use the `onerror` event handler for the unexpected transfer snafu. Be aware that while `onerror` works in all browsers, `onabort` does not.

In practice, these event handlers don't supply all the information you may like to have in a script, such as the filename of the image that's loading at the time. If such information is critical to your scripts, the scripts need to store the name of a currently loading image to a variable before they set the image's `src` property. You also don't know the nature of the error that triggers an error event. You can treat such problems by forcing a scripted page to reload or by navigating to an entirely different spot in your web site.

Example

Listing 31-4 includes an `onabort` event handler. If the images already exist in the cache, you must quit and relaunch the browser to try to stop the image from loading. In that example, we provide a

reload option for the entire page. How you handle the exception depends a great deal on your page design. Do your best to smooth over any difficulties that users may encounter.

`onload`

Compatibility: WinIE3+, MacIE3+, NN2+, Moz+, Safari+, Opera+, Chrome+

An `img` object's `onload` event handler fires when one of three actions occurs: an image's `lowsrc` image finishes loading; in the absence of a `lowsrc` image specification, the `src` image finishes loading; or when each frame of an animated GIF (GIF89a format) appears.

It's important to understand that if you define a `lowsrc` file inside an `<img>` tag, the `img` object receives no further word about the `src` image having completed its loading. If this information is critical to your script, verify the current image file by checking the `src` property of the `Image` object.

Be aware, too, that an `img` element's `onload` event handler may fire before the other elements on the page have completed loading. If the event handler function refers to other elements on the page, the function should verify the existence of other elements prior to addressing them.

Quit and restart your browser, or clear your cache, to get the most from Listing 31-4. As the document first loads, the `lowsrc` image file (the picture of pencil erasers) loads ahead of the NASA image. (Recall that the `lowsrc` property is not supported by all browsers, so you won't necessarily see the pencil erasers loaded ahead of the NASA image.) Again, we deliberately chose to load a large image directly from the NASA web site so that you would have more time to see the behavior of the `onload` event handler. When the erasers are loaded, the `onload` event handler writes "done" to the text field, even though the main image (the NASA image) is not loaded yet.

LISTING 31-4

The Image onload Event Handler

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>The Image onload Event Handler</title>
      <script type="text/javascript">
      function loadIt(theImage,form)
      {
         if (document.images)
         {
            form.result.value = "";
            document.images[0].lowsrc = "desk1.gif";
            document.images[0].src = theImage;
         }
      }
      function checkLoad(form)
      {
         if (document.images)
         {
            form.result.value = document.images[0].complete;
         }
```

continued

LISTING 31-4 *(continued)*

```
      }
      function signal()
      {
         if(confirm("You have stopped the image from loading. Do you want to try again?"))
         {
            location.reload();
         }
      }
      </script>
   </head>
   <body>
      <h1>The Image onload Event Handler</h1>
      <h3>Click the browser's STOP button to invoke the onabort event handler</h3>
      <img alt="image" width="120" height="90" lowsrc="desk4.gif"
           src="http://www.nasa.gov/images/content/406570main_s129e007575_hires_full.jpg"
           onload="if (document.forms[0].result) document.forms[0].result.value='done'"
           onabort="signal()" />
      <form>
         <p><input type="button" value="Is it loaded yet?"
                   onclick="checkLoad(this.form)" />
            <input type="text" name="result" />
            <input type="hidden" />
         </p>
      </form>
   </body>
</html>
```

Related Items: `img.src`, `img.lowsrc` properties

area Element Object

For HTML element properties, methods, and event handlers, see Chapter 26, "Generic HTML Element Objects."

Properties	Methods	Event Handlers
`alt`		
`coords`		
`hash`		
`host`		
`hostname`		
`href`		

Properties	Methods	Event Handlers
noHref		
pathname		
port		
protocol		
search		
shape		
target		

Syntax

Accessing `area` element object properties:

```
(NN3+/IE4+)     [window.]document.links[index].property
(IE4+)          [window.]document.all.elemID.property | method([parameters])
(IE4+)          [window.]document.all.MAPElemID.areas[index].property |
                method([parameters])
(IE5+/W3C)      [window.]document.getElementById("MAPElemID").areas
                [index].property | method([parameters])
(IE5+/W3C)      [window.]document.getElementById("elemID").property |
                method([parameters])
```

Compatibility: WinIE4+, MacIE4+, NN3+, Moz+, Safari+, Opera+, Chrome+

About this object

Document object models treat an image map area object as one of the link (`a` element) objects in a document (see the anchor object in Chapter 30, "Link and Anchor Objects"). When you think about it, such treatment is not illogical at all because clicking a map area generally leads the user to another document or anchor location in the same document — a hyperlinked reference.

Although the HTML definitions of links and map areas differ greatly, the earliest scriptable implementations of both kinds of objects had nearly the same properties and event handlers. Starting with IE4, NN6/Moz, and W3C-compatible browsers, all `area` element attributes are accessible as scriptable properties. Moreover, you can change the makeup of client-side image map areas by way of the `map` element object. The `map` element object contains an array of `area` element objects nested inside. You can remove, modify, or add to the `area` elements inside the `map` element.

Client-side image maps are fun to work with, and they have been well documented in HTML references since Netscape Navigator 2 introduced the feature. Essentially, you define any number of areas within the image, based on shape and coordinates. Many graphics tools can help you capture the coordinates of images that you need to enter into the `coords` attribute of the `<area>` tag.

Tip

If one gotcha exists that trips up most HTML authors, it's the tricky link between the `<img>` and `<map>` tags. You must assign a name to the `<map>`; in the `<img>` tag, the `usemap` attribute requires a hash symbol (#) and the map name. If you forget the hash symbol, you can't create a connection between the image and its map. ■

Listing 31-5 contains an example of a client-side image map that allows you to navigate through different geographical features of the Middle East. As you drag the mouse around an aerial image, certain regions cause the mouse pointer to change, indicating that there is a link associated with the region. Clicking a region results in an alert box indicating which region you clicked.

LISTING 31-5

A Simple Client-Side Image Map

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Image Maps</title>
      <script type="text/javascript">
      function show(msg)
      {
         window.status = msg;
         return true;
      }
      function go(where)
      {
         alert("We're going to " + where + "!");
      }
      function clearIt()
      {
         window.status = "";
         return true;
      }
      </script>
   </head>
   <body>
      <h1>Sinai and Vicinity</h1>
      <img alt="image" src="nile.gif" width="320" height="240"
           usemap="#sinai" />
      <map id="sinai" name="sinai" >
         <area href="javascript:go('Cairo')" coords="12,152,26,161"
               shape="rect" onmouseover="return show('Cairo')"
               onmouseout="return clearIt()" />
         <area href="javascript:go('the Nile River')"
               shape="poly" onmouseover="return show('Nile River')"
               onmouseout="return clearIt()"
               coords="1,155,6,162,0,175,3,201,61,232,109,227,167,238,274,239,292,220,307,
               220,319,230,319,217,298,213,282,217,267,233,198,228,154,227,107,
               221,71,225,21,199,19,165,0,149" />
         <area href="javascript:go('Israel')" coords="95,69,201,91"
               shape="rect" onmouseover="return show('Israel')"
               onmouseout="return clearIt()" />
         <area href="javascript:go('Saudi Arabia')"
               coords="256,57,319,121"
               shape="rect" onmouseover="return show('Saudi Arabia')"
               onmouseout="return clearIt()" />
         <area href="javascript:go('the Mediterranean Sea')"
```

```
            coords="1,55,26,123" shape="rect"
            onmouseover="return show('Mediterranean Sea')"
            onmouseout="return clearIt()" />
      <area href="javascript:go('the Mediterranean Sea')"
            coords="27,56,104,103" shape="rect"
            onmouseover="return show('Mediterranean Sea')"
            onmouseout="return clearIt()" />
    </map>
  </body>
</html>
```

Properties

`alt`

Value: String Read/Write

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

The `alt` property represents the `alt` attribute of an `area`. Internet Explorer displays the `alt` text in a tiny pop-up window (tool tip) above an area when you pause (hover) the mouse pointer over it. There is debate among web developers about Microsoft's usage of tool tips for `alt` text, both in image maps and regular images.

Future browsers may implement this attribute to provide additional information about the link associated with the `area` element. For the time being, Internet Explorer is the only mainstream browser to use the `alt` property in any noticeable way, although all the browsers recognize it.

Related Item: `title` property

`coords`
`shape`

Value: String Read/Write

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

The `coords` and `shape` properties control the location, size, and shape of the image hot spot governed by the `area` element. Shape values that you can use for this property control the format of the `coords` property values, as follows:

Shape	Coordinates	Example
`circ`	center-x, center-y, radius	`"30, 30, 20"`
`circle`	center-x, center-y, radius	`"30, 30, 20"`
`poly`	x1, y1, x2, y2, . . .	`"0, 0, 0, 30, 15, 30, 0, 0"`
`polygon`	x1, y1, x2, y2, . . .	`"0, 0, 0, 30, 15, 30, 0, 0"`
`rect`	left, top, right, bottom	`"10, 20, 60, 40"`
`rectangle`	left, top, right, bottom	`"10, 20, 60, 40"`

The default shape for an `area` is a `rectangle`.

Related Items: None

hash
host
hostname
href
pathname
port
protocol
search
target

(See corresponding properties of the `link` object in Chapter 30, "Link and Anchor Objects.")

noHref

Value: Boolean Read/Write

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

The `noHref` property is used to enable or disable a particular area within a map. The values of the property may be `true`, which means an area is enabled, or `false` to prevent the area from serving as a link within the map. The actual behavior in any of the browsers may be unpredictable, so test carefully.

shape

(See `coords`)

map Element Object

For HTML element properties, methods, and event handlers, see Chapter 26, "Generic HTML Element Objects."

Properties	Methods	Event Handlers
areas[]		onscroll†
name		

†See Chapter 29

Syntax

Accessing `map` element object properties:

```
(IE4+)        [window.]document.all.elemID.property | method([parameters])
(IE5+/W3C)    [window.]document.getElementById("elemID").property |
              method([parameters])
```

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

About this object

The map element object is an invisible HTML container for all area elements, each of which defines a “hot” region for an image. Client-side image maps associate links (and targets) to rectangular, circular, or polygonal regions of the image.

By far, the most important properties of a map element object are the areas array and, to a lesser extent, the name. It is unlikely that you will change the name of a map. (It is better to define multiple map elements with different names, and then assign the desired name to an img element object’s useMap property.) But you can use the areas array to change the makeup of the area objects inside a given client-side map.

Properties

areas[]

Value: Array of area element objects Read/Write

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

Use the areas array to iterate through all area element objects within a map element. Whereas Mozilla-based and WebKit-based browsers adhere closely to the document node structure of the W3C DOM, IE4+ provides more direct access to the area element objects nested inside a map. If you want to rewrite the area elements inside a map, you can clear out the old ones by setting the length property of the areas array to zero. Then assign area element objects to slots in the array to build that array.

Listing 31-6 demonstrates how to use scripting to replace the area element objects inside a map element. The scenario is that the page loads with one image of a computer keyboard. This image is linked to the keyboardMap client-side image map, which specifies details for three hot spots on the image. If you then switch the image displayed in that img element, scripts change the useMap property of the img element object to point to a second map that has specifications more suited to the desk lamp in the second image. Roll the mouse pointer atop the images, and view the URLs associated with each area in the status bar (for this example, the URLs do not lead to other pages).

Another button on the page, however, invokes the makeAreas() function (not supported by MacIE5), which creates four new area element objects and (through DOM-specific pathways) adds those new area specifications to the image. If you roll the mouse atop the image after the function executes, you can see that the URLs now reflect those of the new areas. Also note the addition of a fourth area, whose status bar message appears in Figure 31-2.

LISTING 31-6

Modifying area Elements On-the-Fly

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>map Element Object</title>
      <script type="text/javascript">
      // generate area elements on-the-fly
      function makeAreas()
```

continued

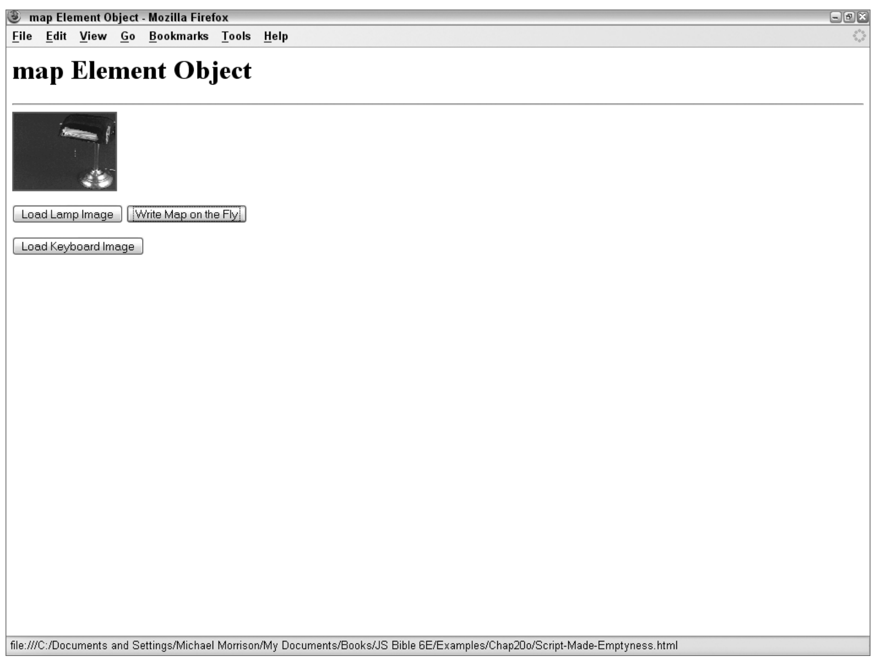
LISTING 31-6 *(continued)*

```
   {
      document.getElementById("myIMG").src = "desk3.gif";
      // build area element objects
      var area1 = document.createElement("area");
      area1.href = "Script-Made-Shade.html";
      area1.shape = "polygon";
      area1.coords = "52,28,108,35,119,29,119,8,63,0,52,28";
      var area2 = document.createElement("area");
      area2.href = "Script-Made-Base.html";
      area2.shape = "rect";
      area2.coords = "75,65,117,87";
      var area3 = document.createElement("area");
      area3.href = "Script-Made-Chain.html";
      area3.shape = "polygon";
      area3.coords = "68,51,73,51,69,32,68,51";
      var area4 = document.createElement("area");
      area4.href = "Script-Made-Emptyness.html";
      area4.shape = "rect";
      area4.coords = "0,0,50,120";
      // stuff new elements into MAP child nodes
      var mapObj = document.getElementById("lamp_map");
      while (mapObj.childNodes.length)
      {
         mapObj.removeChild(mapObj.firstChild);
      }
      mapObj.appendChild(area1);
      mapObj.appendChild(area2);
      mapObj.appendChild(area3);
      mapObj.appendChild(area4);
      // workaround NN6 display bug
      document.getElementById("myIMG").style.display = "inline";
   }
   function changeToKeyboard()
   {
      document.getElementById("myIMG").src = "cpu2.gif";
      document.getElementById("myIMG").useMap = "#keyboardMap";
   }
   function changeToLamp()
   {
      document.getElementById("myIMG").src = "desk3.gif";
      document.getElementById("myIMG").useMap = "#lampMap";
   }
   </script>
</head>
<body>
   <h1>map Element Object</h1>
   <hr />
   <img alt="image" id="myIMG" src="cpu2.gif" width="120" height="90"
       usemap="#keyboardMap" />
   <map id="keyboardMap" name="keyboardMap">
      <area href="AlpaKeys.htm" shape="rect" coords="0,0,26,42" />
```

```
      <area href="ArrowKeys.htm" shape="polygon"
            coords="48,89,57,77,69,82,77,70,89,78,84,89,48,89" />
      <area href="PageKeys.htm" shape="circle" coords="104,51,14" />
   </map>
   <map name="lampMap" id="lamp_map">
      <area href="Shade.htm" shape="polygon"
            coords="52,28,108,35,119,29,119,8,63,0,52,28" />
      <area href="Base.htm" shape="rect" coords="75,65,117,87" />
      <area href="Chain.htm" shape="polygon"
            coords="68,51,73,51,69,32,68,51" />
   </map>
   <form>
      <p><input type="button" value="Load Lamp Image"
                onclick="changeToLamp()" />
         <input type="button" value="Write Map on-the-Fly"
                onclick="makeAreas()" />
      </p>
      <p><input type="button" value="Load Keyboard Image"
                onclick="changeToKeyboard()" />
      </p>
   </form>
  </body>
</html>
```

FIGURE 31-2

Scripts created a special client-side image map for the image.



Related Items: `area` element object

name

Value: String — Read/Write

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

A relationship between the image map and the image needs to be created. We do this with the value of the name property for an image map: it matches the value of the corresponding image's `useMap` property.

Related Item: `useMap` property of the corresponding image

canvas Element Object

For HTML element properties, methods, and event handlers, see Chapter 26, "Generic HTML Element Objects."

Properties	Methods	Event Handlers
fillStyle	arc()	
globalAlpha	arcTo()	
globalCompositeOperation	bezierCurveTo()	
lineCap	beginPath()	
lineJoin	clearRect()	
lineWidth	clip()	
miterLimit	closePath()	
shadowBlur	createLinearGradient()	
shadowColor	createPattern()	
shadowOffsetX	createRadialGradient()	
shadowOffsetY	drawImage()	
strokeStyle	fill()	
	fillRect()	
	getContext()	
	lineTo()	
	moveTo()	
	quadraticCurveTo()	
	rect()	
	restore()	

Properties	Methods	Event Handlers
	rotate()	
	save()	
	scale()	
	stroke()	
	strokeRect()	
	translate()	

Syntax

Accessing `canvas` element object properties:

```
(W3C)     [window.]document.getElementById("canvasID").property |
              method([parameters])
```

Compatibility: WinIE-, MacIE-, NN-, Moz1.8+, Safari1.3+, Opera+, Chrome+

About this object

A *canvas* is a relatively new construct that enables you to create a rectangular region on a page that can be drawn to programmatically through JavaScript. Relative to a web page, a canvas appears as an image since it occupies a rectangular space. Unlike images, however, canvas content is generated programmatically using a series of methods defined on the `canvas` object. Support for canvases first appeared in the Safari browser in version 1.3, and then spread to Mozilla browsers in Mozilla version 1.8, which corresponds to Firefox 1.5. While the current version of IE does not natively support this object, there are third-party libraries that allow you to use this object in IE.

Canvas objects are created and positioned on the page using the `<canvas>` tag, which supports only two unique attributes: `width` and `height`. After the canvas is created, the remainder of the work associated with creating a canvas graphic falls to JavaScript code. You'll typically want to create a special draw function that takes on the task of drawing to the canvas, upon the page loading. The job of the draw function is to use the methods of the `canvas` object to render the canvas graphic.

Listing 31-7 contains a skeletal page for placing a basic canvas with a border. Notice that there is a `draw()` function that is ready to receive the code for rendering the canvas graphic.

LISTING 31-7

A Skeletal Canvas

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>canvas Object</title>
      <script type="text/javascript">
```

continued

LISTING 31-7 *(continued)*

```
      function draw()
      {
         // Draw some stuff
      }
      </script>
      <style type="text/css">
         canvas { border: 1px solid black; }
      </style>
   </head>
   <body onload="draw();">
      <h1>canvas Object</h1>
      <hr />
      <canvas width="350" height="250"></canvas>
   </body>
</html>
```

Listing 31-8 contains a more interesting canvas example that builds on the skeletal page by adding some actual canvas drawing code. The properties and methods in the `canvas` object provide you with the capability to do some amazing things, so consider this example a rudimentary scratching of the canvas surface.

LISTING 31-8

A Canvas Containing a Simple Chart

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>canvas Object</title>
      <script type="text/javascript">
      function draw()
      {
         var canvas = document.getElementById('chart');
            if (canvas.getContext)
            {
               var context = canvas.getContext('2d');
               context.lineWidth = 20;
               // First bar
               context.strokeStyle = "red";
               context.beginPath();
               context.moveTo(20, 90);
               context.lineTo(20, 10);
               context.stroke();
               // Second bar
               context.strokeStyle = "green";
               context.beginPath();
               context.moveTo(50, 90);
```

```
            context.lineTo(50, 50);
            context.stroke();
            // Third bar
            context.strokeStyle = "yellow";
            context.beginPath();
            context.moveTo(80, 90);
            context.lineTo(80, 25);
            context.stroke();
            // Fourth bar
            context.strokeStyle = "blue";
            context.beginPath();
            context.moveTo(110, 90);
            context.lineTo(110, 75);
            context.stroke();
         }
      }
      </script>
      <style type="text/css">
         canvas { border: 1px solid black; }
      </style>
   </head>
   <body onload="draw();">
      <h1>canvas Object</h1>
      <hr />
      <canvas id="chart" width="130" height="100"></canvas>
   </body>
</html>
```

Figure 31-3 shows this canvas example in action, which involves the display of a simple bar chart. The thing to keep in mind is that this bar chart is being rendered programmatically using vector graphics, which is quite powerful.

This example reveals a few more details about how canvases work. Notice that you must first obtain a context in order to perform operations on the canvas. (In reality, you perform graphics operations on a canvas context, not the canvas element itself.) The job of a context is to provide you with a virtual surface on which to draw. You obtain a canvas context by calling the `getContext()` method on the canvas object and specifying the type of context; currently, only the `2d` (two-dimensional) context is supported in browsers.

When you have a context, canvas drawing operations are carried out relative to the context. You are then free to tinker with stroke and fill colors, create paths and fills, and do most of the familiar graphical things that go along with vector drawing.

You can find a more complete tutorial and examples dedicated to drawing with the canvas element at `http://developer.mozilla.org/en/docs/Canvas_tutorial`.

Properties

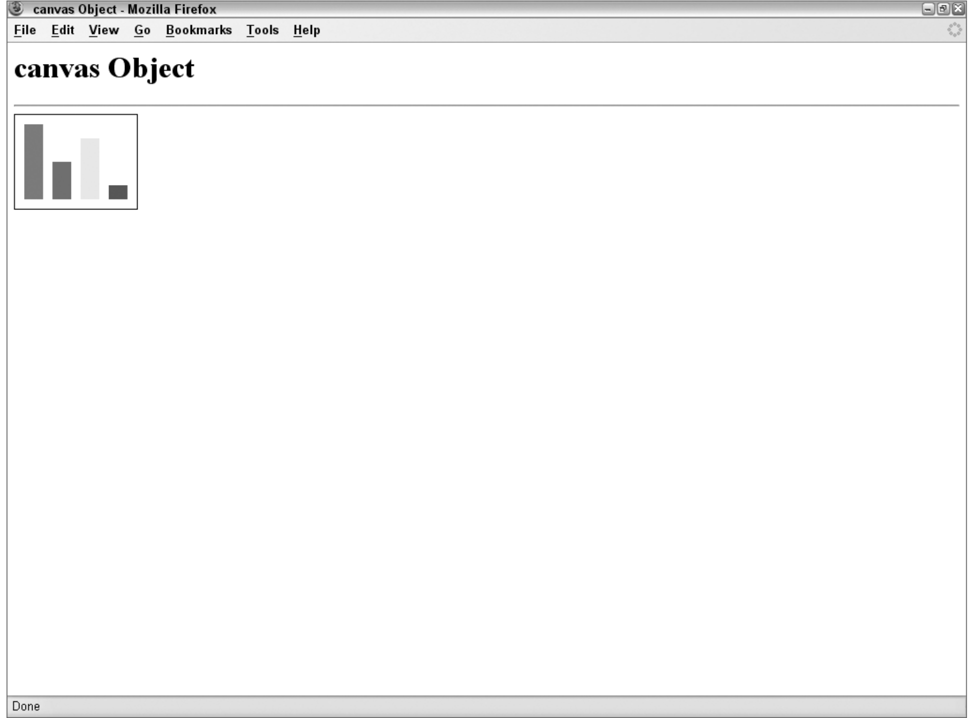
`fillStyle`

Value: String Read/Write

Compatibility: WinIE-, MacIE-, NN-, Moz1.8+, Safari1.3+, Opera+, Chrome+

FIGURE 31-3

A simple bar chart created using a vector canvas.



The `fillStyle` property sets the fill for the brush used in filling a region of a canvas is with a color or pattern. Although you can create gradients and other interesting patterns for use in filling shapes, the most basic usage of the `fillStyle` property is creating a solid color fill by setting the property to an HTML-style color (`#RRGGBB`). All fill operations that follow the `fillStyle` setting will use the new fill style.

Example

Setting a fill color simply involves assigning an HTML-style color to the `fillStyle` property:

```
context.fillStyle = "#FF00FF";
```

Related Item: `strokeStyle` property

globalAlpha

Value: Float Read/Write

Compatibility: WinIE-, MacIE-, NN-, Moz1.8+, Safari1.3+, Opera+, Chrome+

The `globalAlpha` property is a floating-point property that establishes the transparency (or opacity, depending on how you think about it) of the content drawn on a canvas. Acceptable

values for this property range from 0.0 (fully transparent) to 1.0 (fully opaque). The default setting is 1.0, which means that all canvases initially have no transparency.

Example

To set the transparency of a canvas to 50 percent transparency, set the `globalAlpha` property to 0.5:

```
context.globalAlpha = 0.5;
```

Related Item: None

globalCompositeOperation

Value: String — Read/Write

Compatibility: WinIE-, MacIE-, NN-, Moz1.8+, Safari1.3+, Opera+, Chrome+

The `globalCompositeOperation` property determines how the canvas appears in relation to background content on a web page. This is a powerful property because it can dramatically affect the manner in which canvas content appears with respect to any underlying web page content. The default setting is `source-over`, which means opaque areas of the canvas are displayed, but transparent areas are not. Other popular settings include `copy`, `lighter`, and `darker`, among others.

Example

If you want a canvas to always fully cover the background web page, regardless of any transparent areas it may have, you should set the `globalCompositeOperation` property to `copy`:

```
context.globalCompositeOperation = "copy";
```

Related Item: None

lineCap
lineJoin
lineWidth

Value: String, Float (`lineWidth`) — Read/Write

Compatibility: WinIE-, MacIE-, NN-, Moz1.8+, Safari1.3+, Opera+, Chrome+

These properties all impact the manner in which lines are drawn on a canvas. The `lineCap` property determines the appearance of line end points (`butt`, `round`, or `square`). Similar to `lineCap` is the `lineJoin` property, which determines how lines are joined to each other (`bevel`, `miter`, or `round`). By default, lines terminate with no special end point (`butt`) and are joined cleanly with no special joint graphic (`miter`).

The `lineWidth` property establishes the width of lines and is expressed as an integer value greater than 0 in the canvas coordinate space. When a line is drawn, its width appears centered over the line coordinates.

Example

You'll often want to change the `lineWidth` property to get different effects when assembling a canvas graphic. Here's an example of setting a wider line width (`10` in this case):

```
context.lineWidth = 10;
```

Related Items: `miterLimit`, `strokeStyle` properties

miterLimit

Value: Float Read/Write

Compatibility: WinIE-, MacIE-, NN-, Moz1.8+, Safari1.3+, Opera+, Chrome+

The `miterLimit` property is a floating-point value that determines more specifically how lines are joined together. The `miterLimit` property works in conjunction with the `lineJoin` property to cleanly and consistently join lines in a path.

Related Item: `lineJoin` property

shadowBlur
shadowColor
shadowOffsetX
shadowOffsetY

Value: Integer, String (`shadowColor`) Read/Write

Compatibility: WinIE-, MacIE-, NN-, Moz1.8+, Safari1.3+, Opera+, Chrome+

These properties all work in conjunction to establish a shadow around canvas content. The `shadowBlur` property determines the width of the shadow itself, whereas `shadowColor` sets the color of the shadow as an HTML-style RGB value (#RRGGBB). Finally, the `shadowOffsetX` and `shadowOffsetY` properties specify exactly how far the shadow is offset from a graphic. The `shadowBlur`, `shadowOffsetX`, and `shadowOffsetY` properties are all expressed in units of the canvas coordinate space.

Example

The following code creates a shadow that is 5 units wide, light gray in color (#BBBBBB), and offset 3 units in both the X and Y directions:

```
context.shadowBlur = 5;
context.shadowColor = "#BBBBBB";
context.shadowOffsetX = 3;
context.shadowOffsetY = 3;
```

Related Item: None

strokeStyle

Value: String Read/Write

Compatibility: WinIE-, MacIE-, NN-, Moz1.8+, Safari1.3+, Opera+, Chrome+

The `strokeStyle` property controls the style of strokes used to draw on the canvas. You are free to set the stroke style to a gradient or pattern using more advanced canvas features, but the simpler approach is to just set a solid colored stroke as an HTML-style color value (#RRGGBB). The default stroke style is a solid black stroke.

Example

To change the stroke style to a solid green brush, set the `strokeStyle` property to the color green:

```
context.strokeStyle = "#00FF00";
```

Related Item: `fillStyle` property

Methods

arc(*x, y, radius, startAngle, endAngle, clockwise*)
arcTo(*x1, y1, x2, y2, radius*)
bezierCurveTo(*cp1x, cp1y, cp2x, cp2y, x, y*)
quadraticCurveTo(*cpx, cpy, x, y*)

Returns: Nothing

Compatibility: WinIE-, MacIE-, NN-, Moz1.8+, Safari1.3+, Opera+, Chrome+

These methods are all responsible for drawing curves in one way or another. If you have any experience with drawing vector graphics, then you're probably familiar with the difference between arcs, Bezier curves, and quadratic curves.

The `arc()` method draws a curved line based upon a center point, a radius, and start and end angles. You can think of this method as tracing the curve of a circle from one angle to another. The angles are expressed in radians, not degrees, so you'll probably need to convert degrees to radians:

```
var radians = (Math.PI / 180) * degrees;
```

The last argument to `arc()` is a Boolean value that determines whether or not the arc is drawn in the clockwise (`true`) or counterclockwise (`false`) direction.

The `arcTo()` method draws an arc along a curve based upon tangent lines of a circle. This method is not implemented in Mozilla until version 1.8.1.

Finally, the `bezierCurveTo()` and `quadraticCurveTo()` methods draw a curved line based upon arguments relating to Bezier and quadratic curves, respectively, which are a bit beyond this discussion. To learn more about Bezier and quadratic curves, check out `en.wikipedia.org/wiki/Bézier_curve`.

beginPath()
closePath()

Returns: Nothing

Compatibility: WinIE-, MacIE-, NN-, Moz1.8+, Safari1.3+, Opera+, Chrome+

These two methods are used to manage paths. Call the `beginPath()` method to start a new path, into which you can then add shapes. When you're finished, call `closePath()` to close up the path. The only purpose of `closePath()` is to close a subpath that is still open, meaning that you want to finish connecting an open shape back to its start. If you've created a path that is already closed,

there is no need to call the `closePath()` method. Listing 31-8 contains a good example of where the `closePath()` method is unnecessary because the bar shapes don't need to be closed.

`clip()`

Returns: Nothing

Compatibility: WinIE-, MacIE-, NN-, Moz1.8+, Safari1.3+, Opera+, Chrome+

The `clip()` method recalculates the clipping path based upon the current path and the clipping path that already exists. Subsequent drawing operations rely on the newly calculated clipping path.

`createLinearGradient(`*`x1, y1, x2, y2`*`)`
`createRadialGradient(`*`x1, y1, radius1, x2, y2, radius2`*`)`
`createPattern(`*`image, repetition`*`)`

Returns: Gradient object reference, pattern object reference (`createPattern()`)

Compatibility: WinIE-, MacIE-, NN-, Moz1.8+, Safari1.3+, Opera+, Chrome+

These methods are used to create special patterns and gradients for fill operations. A linear gradient is specified as a smooth color transition between two coordinates, whereas a radial gradient is specified based upon two circles with similar radii. You set the actual color range of a gradient by calling the `addColorStop()` method on the gradient object returned from the `createLinearGradient()` and `createRadialGradient()` methods. The `addColorStop()` method accepts a floating-point offset and a string color value as its only two arguments.

The `createPattern()` method is used to create a fill pattern based upon an image. You provide an image object and a repetition argument for how the image is tiled when filling a region. Repetition options include `repeat`, `repeat-x`, `repeat-y`, and `no-repeat`.

`drawImage(`*`image, x, y`*`)`
`drawImage(`*`image, x, y, width, height`*`)`
`drawImage(`*`image, srcX, srcY, srcWidth, srcHeight, destX, destY, destWidth, destHeight`*`)`

Returns: Nothing

Compatibility: WinIE-, MacIE-, NN-, Moz1.8+, Safari1.3+, Opera+, Chrome+

These `drawImage()` methods all draw an image to the context. The difference between them has to do with if and how the image is scaled as it is drawn. The first version draws the image at a coordinate with no scaling, whereas the second version scales the image to the specified target width and height. Finally, the third version enables you to draw a portion of the image to a target location with a scaled width and height.

`fill()`
`fillRect(`*`x, y, width, height`*`)`
`clearRect(`*`x, y, width, height`*`)`

Returns: Nothing

Compatibility: WinIE-, MacIE-, NN-, Moz1.8+, Safari1.3+, Opera+, Chrome+

These methods are used to fill and clear areas. The `fill()` method fills the area within the current path, whereas the `fillRect()` method fills a specified rectangle independent of the current path. The `clearRect()` method is used to clear (erase) a rectangle.

getContext(*contextID*)

Returns: Context object reference

Compatibility: WinIE-, MacIE-, NN-, Moz1.8+, Safari1.3+, Opera+, Chrome+

The `getContext()` method operates on a canvas element object, not a context, and is used to obtain a context for further graphical operations. You must call this method to obtain a context before you can draw to a canvas since all of the canvas drawing methods are actually called relative to a context, not a canvas. For the standard two-dimensional canvas context, specify `"2d"` as the sole parameter to the method.

lineTo(*x, y*)
moveTo(*x, y*)

Returns: Nothing

Compatibility: WinIE-, MacIE-, NN-, Moz1.8+, Safari1.3+, Opera+, Chrome+

These two methods are used to draw lines and adjust the stroke location. The `moveTo()` method simply moves the current stroke location without adding anything to the path. The `lineTo()` method, on the other hand, draws a line from the current stroke location to the specified point. It's worth pointing out that drawing a line with the `lineTo()` method only adds a line to the current path; the line doesn't actually appear on the canvas until you call the `stroke()` method to carry out the actual drawing of the path.

rect(*x, y, width, height*)

Returns: Nothing

Compatibility: WinIE-, MacIE-, NN-, Moz1.8+, Safari1.3+, Opera+, Chrome+

The `rect()` method adds a rectangle to the current path. Similar to other drawing methods, the rectangle is actually just added to the current path, which isn't truly visible until you render it using the `stroke()` method.

restore()
save()

Returns: Nothing

Compatibility: WinIE-, MacIE-, NN-, Moz1.8+, Safari1.3+, Opera+, Chrome+

It's possible to save and restore the state of the graphic context, in which case you can make changes and then return to a desired state. The context state includes information such as the clip region, line width, fill color, and so forth. Call the `save()` and `restore()` methods to save and restore the context state.

rotate(*angle*)
scale(*x*, *y*)
translate(*x*, *y*)

Returns: Nothing

Compatibility: WinIE-, MacIE-, NN-, Moz1.8+, Safari1.3+, Opera+, Chrome+

All of the drawing operations that you perform on a canvas are expressed in units relative to the coordinate system of the canvas. These three methods enable you to alter the canvas coordinate system by rotating, scaling, or translating its origin. Rotating the coordinate system affects how angles are expressed in drawing operations that involve angles. Scaling the coordinate system impacts the relative size of units expressed in the system. And finally, translating the coordinate system alters the location of the origin, which affects where positive and negative values intersect on the drawing surface.

stroke()
strokeRect(*x*, *y*, *width*, *height*)

Returns: Nothing

Compatibility: WinIE-, MacIE-, NN-, Moz1.8+, Safari1.3+, Opera+, Chrome+

Most of the drawing operations on a canvas impact the current path, which you can think of as a drawing you've committed to memory but have yet to put on paper. You render a path to the canvas by calling the `stroke()` method. If you want to draw a rectangle to the canvas without dealing with the current path, call the `strokeRect()` method.

CHAPTER

32

Event Objects

IN THIS CHAPTER

The "life" of an event object

Event support in different browser generations

Retrieving information from an event

Prior to version 4 browsers, user and system actions — events — were captured predominantly by event handlers defined as attributes inside HTML tags. For instance, when a user clicked a button, the `clickevent` triggered the `onclick` event handler in the tag. That handler may invoke a separate function or perform some inline JavaScript script. Even so, the events themselves were rather dumb: Either an event occurred or it didn't. Where an event occurred (that is, the screen coordinates of the pointer at the moment the mouse button was clicked) and other pertinent event tidbits (for example, whether a keyboard modifier key was pressed at the same time) were not part of the equation — until version 4 browsers, that is.

While remaining fully backward compatible with the event handler mechanism of old, version 4 browsers had the first event model that turn events into first-class objects whose properties automatically carry a lot of relevant information about the event when it occurs. These properties are fully exposed to scripts, allowing pages to respond more intelligently about what the user does with the page and its elements.

Another new aspect of version 4 event models was the notion of "event propagation." It was possible to have an event processed by an object higher up the element containment hierarchy whenever it made sense to have multiple objects share one event handler. That the event being processed carried along with it information about the intended target, plus other golden information nuggets, made it possible for event handler functions to be smart about processing the event without requiring an event handler call to pass all kinds of target-specific information.

Unfortunately, the joy of this newly found power is tempered by the forces of object model incompatibility. Event object models are clearly divided along two fronts:

- The IE4+ model
- The model adopted by the W3C DOM Level 2, as implemented in NN6+/Moz and Gecko-, WebKit-, and Presto-based browsers

Many of these distinctions are addressed in the overviews of the object models in Chapter 26, "Generic HTML Element Objects." In this chapter, you find out more about the actual event objects that contain all the "goodies." Where possible, cross-browser concerns are addressed.

Why "Events"?

Graphical user interfaces are more difficult to program than the "old-fashioned" command-line interface. With a command-line or menu-driven system, users were intentionally restricted in the types of actions they could take at any given moment. The world was very modal, primarily as a convenience to programmers who led users through rigid program structures.

That all changes in a graphical user interface, such as Windows, Mac OS, XWindow System, or any other derived from the pioneering work of the Xerox Star system. The challenge for programmers is that a good user interface in this realm must make it possible for users to perform all kinds of actions at any given moment: roll the mouse, click a button, type a key, select text, choose a pull-down menu item, and so on. To accommodate this, a program (or, better yet, the operating system) must be on the lookout for any possible activity coming from all input ports, whether it be the mouse, keyboard, or network connection.

A common methodology to accomplish this at the operating system level is to look for any kind of event, whether it comes from user action or some machine-generated activity. The operating system or program then looks up how it should process each kind of event. Such events, however, must have some smarts about them so that the program knows what, and where on the screen, the event is.

What an event knows (and when it knows it)

Although the way to reference an event object varies a bit among the three event models, the one concept they all share is that an event object is created the instant the event action occurs. For instance, if you click a button, an event object is created in the browser's memory. As the object is created, the browser assigns values to the object's properties — properties that reflect numerous characteristics of that specific event. For a `click` event, that information includes the coordinates of the click, and which mouse button was used to generate the event. To be even more helpful, the browser does some quick calculations to determine that the coordinates of the `click` event coincide with the rectangular space of a button element on the screen. Therefore, the event object has as one of its properties a reference to the "screen thing" that you clicked on.

Most event object properties are read-only (in some even models, all the properties are), because an event object is like a snapshot of an event action. If the event model were to allow modification of event properties, performing both potentially useful and potentially unfriendly actions would be possible. For example, how frustrating would it be to a user to attempt to type into a text box, only for a keystroke to be modified after the actual key press and a totally different character to appear in the text box? On the other hand, perhaps it may be useful in some situations to make sure that anything typed into a text box is converted to uppercase characters, no matter what is typed. Each event model brings its own philosophy to the table in this regard. For example, the IE4+ event model allows keyboard character events to be modified by script; the W3C DOM event model does not.

Perhaps the most important aspect of an event object to keep in mind is that it exists only as long as scripts process the event. An event can trigger an event handler — usually a function. That function, of course, can invoke other functions. As long as statements are still executing in response to the event handler, the event object and all its properties are still "alive" and available to your scripts. But after the last script statement runs, the event object reverts to an empty object.

The reason an event object has such a brief life is that there can be only one event object at a time. In other words, no matter how complex your event handler functions are, or how rapidly events fire, they are executed serially (for experienced programmers: there is one execution thread). The operating system buffers events that start to bunch up on each other. Except in rare cases in which the buffer gets full and events are not recorded, event handlers are executed in the order in which the events occur.

The static Event object

Up to this point, the discussion has been about the event object (with a lowercase “e”), which is one instance of an event, with all the properties associated with that specific event action. In the W3C DOM event model, there is also a static `Event` object (with an uppercase “E”) that includes additional subcategories within it. These subcategories are all covered later in this chapter, but they are introduced here to illustrate the contrast between the event and `Event` objects. The former, as you’ve seen, is a transient object with details about a specific event action; the latter serves primarily as a holder of event-related constant values that scripts can use. The static `Event` object is always available to scripts inside any window or frame. For a list of all `Event` object properties in NN6+/Moz and Gecko-, WebKit-, and Presto-based browsers, use The Evaluator (see Chapter 4, ‘JavaScript Essentials’): enter **Event** into the bottom text box (also check out the `KeyEvent` object in NN6+/Moz).

The static `Event` object also turns out to be the object from which event objects are cloned. Thus, the static `Event` object has a number of properties and methods that apply to (are inherited by) the event objects created by event actions. These relationships are more important in the W3C DOM event model, which builds upon the DOM’s object-oriented tendencies to implement the event model.

Event Propagation

Prior to version 4 browsers, an event fired on an object. If an event handler was defined for that event and that object, the handler executed; if there was no event handler, the event just disappeared into the ether. Newer browsers, however, send events on a longer ride, causing them to propagate through the document object models. As you know by now, two propagation models exist, one for each of the event models in use today: IE4+ and W3C DOM, as implemented in NN6+/Moz and Gecko-, WebKit-, and Presto-based browsers. It’s also worth mentioning the event model that is unique to NN4, which served as a third model prior to NN4 succumbing to modern browsers. The NN4 event model has historical relevance because it aids in understanding the latter two models. Conceptually, the NN4 and IE4+ propagation models are diametrically opposite each other — any NN4 event propagates inward toward the target, whereas an IE event starts at the target and propagates outward. But the W3C DOM model manages to implement both models simultaneously, albeit with all new syntax so as not to step on the older models.

At the root of all three models is the notion that every event has a target. For user-initiated actions, this is fairly obvious. If you click a button or type in a text box, that button is the target of your mouse-related event; the text box is the target of your keyboard event. System-generated events are not so obvious, such as the `onload` event that fires after a page finishes loading. In all event models, this event fires on the `window` object. What distinguishes the event propagation models is how an event reaches its target, and what, if anything, happens to the event after it finishes executing the event handler associated with the target.

NN4-only event propagation

Although NN4 has given way to newer browsers, its propagation model initiated some concepts that are found in the modern W3C DOM event propagation model. The name for the NN4 model is *event capture*.

In NN4, all events propagate from the top of the document object hierarchy (starting with the `window` object) downward to the target object. For example, if you click a button in a form, the `click` event passes through the `window` and `document` (and, if available, `layer`) objects before reaching the button (the `form` object is not part of the propagation path). This propagation happens instantaneously, so that there is no performance penalty by this extra journey.

The event that passes through the `window`, `document`, and `layer` objects is a fully formed event object, complete with all properties relevant to that event action. Therefore, if the event were processed at the window level, one of the event object's properties is a reference to the target object, so that the event handler scripts at the window level can find out information, such as the name of the button, and even get a reference to its enclosing form.

By default, event capture is turned off. To instruct the `window`, `document`, or `layer` object levels to process that passing click object, requires turning on event capture for the `window`, `document`, and/or `layer` object.

Enabling NN4 event capture

All three objects just mentioned — `window`, `document`, and `layer` — have a `captureEvents()` method. You use this method to enable event capture at any of those object levels. The method requires one or more parameters, which are the event types (as supplied by `Event` object constants) that the object should capture, while letting all others pass untouched. For example, if you want the `window` object to capture all `keypress` events, you include the following statement in a script that executes as the page loads:

```
window.captureEvents(Event.KEYPRESS);
```

Defining event handlers in the intended targets is also a good idea, even if they are empty (for example, `onkeypress=""`) to help NN4 generate the event in the first place. If you want the window to capture multiple event types, string the event type constants together, separated by the pipe character:

```
window.captureEvents(Event.KEYPRESS | Event.CLICK);
```

Now you must assign an action to the event, at the window's level, for each event type. More than likely, you have defined functions to execute for the event. Assign a function reference to the event handler by setting the handler property of the `window` object:

```
window.onkeypress = processKeyEvent;
window.onclick = processClickEvent;
```

Hereafter, if a user clicks a button or types into a field inside that window, the events are processed by their respective window-level event handler functions.

Turning off event capture

As soon as you enable event capture for a particular event type in a document, that capture remains in effect until the page unloads or you specifically disable the capture. You can turn off event capture for each event via the `window`, `document`, or `layer releaseEvents()` method. The `releaseEvents()` method takes the same kind of parameters — `Event` object type constants — as the `captureEvents()` method.

The act of releasing an event type simply means that events go directly to their intended targets without stopping elsewhere for processing, even if an event handler for the higher-level object

is still defined. And because you can release individual event types based on parameters set for the `releaseEvents()` method, other events being captured are not affected by the release of others.

Passing events toward their targets

If you capture a particular event type in NN4, your script may need to perform some limited processing on that event before letting it reach its intended target. For example, perhaps you want to do something special if a user clicks an element with the Shift meta key pressed. In that case, the function that handles the event at the document level inspects the event's `modifiers` property to determine if the Shift key was pressed at the time of the event. If the Shift key was not pressed, you want the event to continue on its way to the element that the user clicked.

To let an event pass through the object hierarchy to its target, you use the `routeEvent()` method, passing as a parameter the event object being handled in the current function. A `routeEvent()` method does not guarantee that the event will reach its intended destination, because another object in between may have event capturing for that event type turned on and will intercept the event. That object, too, can let the event pass through with its own `routeEvent()` method.

In some cases, your scripts need to know if an event that is passed onward by the `routeEvent()` method activated a function that returns a value. This knowledge is especially valuable if your event must return a `true` or `false` value to let an object know if it should proceed with its default behavior (for example, whether a link should activate its `href` attribute URL or cancel after the event handler evaluates to `return true` or `return false`). When a function is invoked by the action of a `routeEvent()` method, the return value of the destination function is passed back to the `routeEvent()` method. That value, in turn, can be returned to the object that originally captured the event.

Event traffic cop

The last scenario is one in which a higher-level object captures an event and directs the event to a particular object elsewhere in the hierarchy. For example, you could have a document-level event handler function direct every `click` event whose `modifiers` property indicates that the Alt key was pressed, to a Help button object whose own `onclick` event handler displays a help panel (perhaps showing an otherwise hidden layer).

You can redirect an event to any object via the `handleEvent()` method. This method works differently from the others described in this chapter, because the object reference of this method is the reference of the object to handle the event (with the event object being passed as a parameter, such as one of the other methods). As long as the target object has an event handler defined for that event, it will process the event as if it had received the event directly from the system (even though the event object's target property may be some other object entirely).

IE4+ event propagation

IE's event propagation model is called *event bubbling*, since events "bubble" upward from the target object through the HTML element containment hierarchy. It's important to distinguish between the old-fashioned document object hierarchy (followed in the NN4 event capture model) and the more modern notion of HTML element containment — a concept that carries over to the W3C DOM as well.

A good way to demonstrate the effect of event bubbling — a behavior that is turned on by default — is to populate a simple document with lots of event handlers to see which ones fire and in

what order. Listing 32-1 has `onclick` event handlers defined for a button inside a form, the form itself, and other elements and objects, all the way up the hierarchy to the window.

LISTING 32-1

Event Bubbling Demonstration

```
<!DOCTYPE html>
<html onclick="alert('Event is now at the HTML element.')">
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Event Bubbles</title>
      <script type="text/javascript">
      function init()
      {
         window.onclick = winEvent
         document.onclick = docEvent;
         document.body.onclick = docBodEvent;
      }
      function winEvent()
      {
         alert("Event is now at the window object level.");
      }
      function docEvent()
      {
         alert("Event is now at the document object level.");
      }
      function docBodEvent()
      {
         alert("Event is now at the BODY element.");
      }
      </script>
   </head>
   <body onload="init()">
      <h1>Event Bubbles</h1>
      <hr />
      <form onclick="alert('Event is now at the FORM element.')">
         <input type="button" value="Button 'main1'" name="main1"
                onclick="alert('Event started at Button: ' + this.name)" />
      </form>
   </body>
</html>
```

You can try this listing in IE4+ and even NN6+/Moz , Safari, Opera, or Chrome because W3C DOM browsers also observe event bubbling. But you will notice differences in the precise propagation among WinIE4+, MacIE4+, and W3C DOM browsers. But notice that after you click the button in Listing 32-1, the event first fires at the target: the button. Then the event bubbles upward through the HTML containment to fire at the enclosing `form` element, next to the enclosing `body` element, and so on. Where the differences occur are after the `body` element. Table 32-1 shows the objects for which event handlers are defined in Listing 32-1, and to which objects the `click` event bubbles in the three classes of browsers.

Despite the discrepancies in Table 32-1, events do bubble through the most likely HTML containers that come to mind. The object level with the most global scope, and that works in all browser categories shown in the table, is the `document` object.

Preventing IE event bubbling

Because bubbling occurs by default, there are times when you may prefer to prevent an event from bubbling up the hierarchy. For example, if you have one handler at the `document` level whose job is to deal with the `click` event from a related series of buttons, any other object that receives `click` events will allow those events to bubble upward to the `document` level unless the bubbling is cancelled. Having the event bubble up could conflict with the document-level event handler.

TABLE 32-1

Event Bubbling Variations for Listing 32-1

Event Handler Location	WinIE4+/Opera	MacIE4+	NN6+/Moz/Safari/Chrome
`button`	Yes	Yes	Yes
`form`	Yes	Yes	Yes
`body`	Yes	Yes	Yes
`HTML`	Yes	No	Yes
`document`	Yes	Yes	Yes
`window`	No	No	Yes

Each `event` object in IE has a property called `cancelBubble`. The default value of this property is `false`, which means that the event bubbles to the next outermost container that has an event handler for that event. But if, in the execution of an event handler, that property is set to `true`, the event does not bubble up any higher when the processing of that handler finishes its job. Therefore, to stop an event from bubbling beyond the current event handler, include the following statement somewhere in the handler function:

```
event.cancelBubble = true;
```

You can prove this to yourself by modifying the page in Listing 32-1 to cancel bubbling at any level. For example, if you change the event handler of the `form` element to include a statement that cancels bubbling, the event goes no further than the `form` in IE (while the WebKit engine and recent versions of the Mozilla/Gecko engine support some of the IE syntax, the syntax is different for NN6+/Moz browsers, as discussed later in this chapter):

```
<form onclick="alert('Event is now at the form element.');
   event.cancelBubble=true">
```

Preventing IE event default action

In the days when events were almost always bound to elements by way of attributes in tags, the technique to block the event's default action was to make sure the event handler evaluated to

`return false`. This is how, for instance, a `form` element's `onsubmit` event handler could prevent the form from carrying out the submission if client-side form validation failed.

To enhance that capability — especially when events are bound by other means, such as object element properties — IE's `event` object includes a `returnValue` property. Assign `false` to this property in the event handler function to block the element's default action to the event:

```
event.returnValue = false;
```

This way of blocking default actions in IE is often more effective than the old `return false` technique.

Redirecting events

Starting with IE5.5, you can redirect an event to another element, but with some limitations. The mechanism that makes this possible is the `fireEvent()` method of all HTML element objects (see Chapter 26). This method isn't so much redirecting an event as causing a brand-new event to be fired. But you can pass most of the properties of the original `event` object with the new event by specifying a reference to the old `event` object as the optional second parameter to the `fireEvent()` method.

The big limitation in this technique, however, is that the reference to the target element gets lost in this hand-off to the new event. The `srcElement` property of the old event gets overwritten with a reference to the object that is the target of the call to `fireEvent()`. For example, consider the following `onclick` event handler function for a button inside a `form` element:

```
function buttonEvent() {
   event.cancelBubble = true;
   document.body.fireEvent("onclick", event);
}
```

By cancelling event bubbling, the event does not propagate upward to the enclosing `form` element. Instead, the event is explicitly redirected to the `body` element, passing the current `event` object as the second parameter. When the event handler function for the `body` element runs, its `event` object has information about the original event, such as the mouse button used for the click and the coordinates. But the `event.srcElement` property points to the `document.body` object. As the event bubbles upward from the `body` element, the `srcElement` property continues to point to the `document.body` object. You can see this at work in Listing 32-2 for IE5.5+.

LISTING 32-2

Cancelling and Redirecting Events in IE5.5+

```
<!DOCTYPE html>
<html onclick="revealEvent('HTML', event)">
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Event Cancelling & Redirecting</title>
      <script type="text/javascript">
         // display alert with event object info
         function revealEvent(elem, evt)
```

```
        {
           var msg = "Event (from " + evt.srcElement.tagName + " at ";
           msg += event.clientX + "," + event.clientY + ") is now at the ";
           msg += elem + " element.";
           alert(msg);
        }
        function init()
        {
           document.onclick = docEvent;
           document.body.onclick = docBodEvent;
        }
        function docEvent()
        {
           revealEvent("document", event);
        }
        function docBodEvent()
        {
           revealEvent("BODY", event);
        }
        function buttonEvent(form)
        {
           revealEvent("BUTTON", event);
           // cancel if checked (IE4+)
           event.cancelBubble = form.bubbleCancelState.checked;
           // redirect if checked (IE5.5+)
              if (form.redirect.checked)
              {
                 document.body.fireEvent("onclick", event);
              }
        }
     </script>
  </head>
  <body onload="init()">
     <h1>Event Cancelling &amp; Redirecting</h1>
     <hr />
     <form onclick="revealEvent('FORM', event)">
        <p><button name="main1" onclick="buttonEvent(this.form)">Button
           'main1'</button>
        </p>
        <p><input type="checkbox" name="bubbleCancelState"
                  onclick="event.cancelBubble=true" />Cancel Bubbling at BUTTON
           <br />
           <input type="checkbox" name="redirect"
                  onclick="event.cancelBubble=true" />Redirect Event to BODY
        </p>
     </form>
  </body>
</html>
```

Listing 32-2 is a modified version of Listing 32-1. Major additions are enhanced event handlers at each level so that you can see the tag name of the event that is regarded as the `srcElement` of the event, as well as the coordinates of the `click` event. With both check boxes unchecked, events bubble upward from the button, and the `button` element is then shown to be the original target all the

way up the bubble hierarchy. If you check the Cancel Bubbling check box, the event goes no further than the button element, because that's where event bubbling is turned off. If you then check the "Redirect Event to Body" check box, the original event is cancelled at the button level, but a new event is fired at the body element. But notice that by passing the old event object as the second parameter, the click location properties of the old event are applied to the new event directed at the body. This event then continues to bubble upward from the body.

As a side note, if you uncheck the Cancel Bubbling check box but leave the Redirect Event box checked, you can see how the redirection is observed at the end of the button's event handler, and that something special goes on. The original event is held aside by the browser while the redirected event bubbles upward. As soon as that event-processing branch finishes, the original bubbling propagation carries on with the form. Notice, though, that the event object still knows that it was targeted at the button element, and the other properties are intact. This means that for a time, two event objects were in the browser's memory, but only one is "active" at a time. While the redirected event is propagating, the window.event object refers to that event object only.

Applying event capture

WinIE 5 and later also provide a kind of event capture, which overrides all other event propagation. Intended primarily for temporary capture of mouse events, it is controlled not through the event object but via the setCapture() and releaseCapture() methods of all HTML element objects (described in Chapter 26).

When you engage capture mode, all mouse events are directed to the element object that invoked the setCapture() method, regardless of the actual target of the event. This action facilitates such activities as element dragging, so that mouse events that might fire outside of the intended target (for example, when dragging the cursor too fast for the animation to track) continue to go to the target. When the drag mode is no longer needed, invoke the releaseCapture() method to allow mouse events to propagate normally.

W3C event propagation

Yielding to arguments in favor of both NN4's event capture and IE's event bubbling, the W3C DOM group managed to assemble an event model that employs both propagation systems. Although forced to use new syntax so as not to conflict with older browsers, the W3C DOM propagation model works like the NN4 model for capture and like IE4+ for bubbling. In other words, an event bubbles by default, but you can also turn on event capture if you want. Thus, an event first trickles down the element containment hierarchy to the target; then it bubbles up through the reverse path.

Event bubbling is on by default, just as in IE4+. To enable capture, you must apply a W3C DOM event listener to an object at some higher container. Use the addEventListener() method (see Chapter 26) for any visible HTML element or node. One of the parameters of the addEventListener() method determines whether the event listener function should be triggered while the event is bubbling or while it is captured.

Listing 32-3 is a simplified example for NN6+/Moz/W3C that demonstrates how a click event aimed at a button can be both captured and allowed to bubble. Most event handling functions are assigned inside the init() function. Borrowing code from Listing 32-1, event handlers are assigned to the window, document, and body objects as property assignments. These are automatically treated as bubble-type event listeners. Next, two objects — document and form — are given capture-type event listeners for the click event. The document object event listener invokes the same function as the bubble-type event handler (the alert text includes some asterisks to remind you

that it is the same alert being displayed in both the capture and bubble phases of the event). For the form object, however, the capture-type event listener is directed to one function, while a bubble-type listener for the same object is directed at a separate function. In other words, the form object invokes one function as the event trickles down to the target, and another function when the event starts bubbling back up. Many of the event handler functions dynamically read the eventPhase property of the event object to reveal which phase of event propagation is in force at the instance the event handler is invoked (although an apparent bug reports the incorrect phase at the document object during event capture).

LISTING 32-3

W3C Event Capture and Bubble

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>W3C DOM Event Propagation</title>
      <script type="text/javascript">
         function init()
         {
            // using old syntax to assign bubble-type event handlers
            window.onclick = winEvent;
            document.onclick = docEvent;
            document.body.onclick = docBodEvent;
            // turn on click event capture for two objects
            document.addEventListener("click", docEvent, true);
            document.forms[0].addEventListener("click", formCaptureEvent, true);
            // set event listener for bubble
            document.forms[0].addEventListener("click", formBubbleEvent, false);
         }
         function winEvent(evt)
         {
            alert("Event is now at the window object level
                 (" + getPhase(evt) + ").");
         }
         function docEvent(evt)
         {
            alert("Event is now at the **document** object level
                 (" + getPhase(evt) + ").");
         }
         function docBodEvent(evt)
         {
            alert("Event is now at the BODY level (" + getPhase(evt) + ").");
         }
         function formCaptureEvent(evt)
         {
            alert("This alert triggered by FORM only on CAPTURE.");
         }
         function formBubbleEvent(evt)
         {
            alert("This alert triggered by FORM only on BUBBLE.");
```

continued

LISTING 32-3 *(continued)*

```
         }
         // reveal event phase of current event object
         function getPhase(evt)
         {
            switch (evt.eventPhase)
            {
               case 1:
                  return "CAPTURING";
                  break;
               case 2:
                  return "AT TARGET";
                  break;
               case 3:
                  return "BUBBLING";
                  break;
               default:
                  return "";
            }
         }
      </script>
   </head>
   <body onload="init()">
      <h1>W3C DOM Event Propagation</h1>
      <hr />
      <form>
         <input type="button" value="Button 'main1'" name="main1"
                onclick="alert('Event is now at the button object level ('
                               + getPhase(event)
                               + ').')" />
      </form>
   </body>
</html>
```

If you want to remove event capture after it has been enabled, use the `removeEventListener()` method on the same object as the event listener that was originally added (see Chapter 26). And, because multiple event listeners can be attached to the same object, specify the exact same three parameters to the `removeEventListener()` method that were applied to the `addEventListener()` method.

Preventing W3C event bubbling or capture

Corresponding to the `cancelBubble` property of the IE4+ `event` object is an event object method in the W3C DOM. The method that prevents propagation in any event phase is the `stopPropagation()` method. Invoke this method anywhere within an event listener function. The current function executes to completion, but the event propagates no further.

Listing 32-4 extends the example of Listing 32-3 to include two check boxes that let you stop propagation at the `form` element in your choice of the capture or bubble phase.

LISTING 32-4

Preventing Bubble and Capture

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>W3C DOM Event Propagation</title>
      <script type="text/javascript">
         function init()
         {
            // using old syntax to assign bubble-type event handlers
            window.onclick = winEvent;
            document.onclick = docEvent;
            document.body.onclick = docBodEvent;
            // turn on click event capture for two objects
            document.addEventListener("click", docEvent, true);
            document.forms[0].addEventListener("click", formCaptureEvent, true);
            // set event listener for bubble
            document.forms[0].addEventListener("click", formBubbleEvent, false);
         }
         function winEvent(evt)
         {
            if (evt.target.type == "button")
            {
               alert("Event is now at the window object level ↩
                    (" + getPhase(evt) + ").");
            }
         }
         function docEvent(evt)
         {
            if (evt.target.type == "button")
            {
               alert("Event is now at the **document** object level ↩
                    (" + getPhase(evt) + ").");
            }
         }
         function docBodEvent(evt)
         {
            if (evt.target.type == "button")
            {
               alert("Event is now at the BODY level (" + getPhase(evt) + ").");
            }
         }
         function formCaptureEvent(evt)
```

continued

LISTING 32-4 *(continued)*

```
      {
         if (evt.target.type == "button")
         {
            alert("This alert triggered by FORM only on CAPTURE.");
            if (document.forms[0].stopAllProp.checked)
            {
               evt.stopPropagation();
            }
         }
      }
      function formBubbleEvent(evt)
      {
         if (evt.target.type == "button")
         {
            alert("This alert triggered by FORM only on BUBBLE.");
            if (document.forms[0].stopDuringBubble.checked)
            {
               evt.stopPropagation();
            }
         }
      }
      // reveal event phase of current event object
      function getPhase(evt)
      {
         switch (evt.eventPhase)
         {
            case 1:
               return "CAPTURING";
               break;
            case 2:
               return "AT TARGET";
               break;
            case 3:
               return "BUBBLING";
               break;
            default:
               return "";
         }
      }
   </script>
</head>
<body onload="init()">
   <h1>W3C DOM Event Propagation</h1>
   <hr />
   <form>
      <input type="checkbox" name="stopAllProp" />Stop all propagation at FORM
      <br />
```

```
         <input type="checkbox" name="stopDuringBubble" />Prevent bubbling past FORM
         <hr />
         <input type="button" value="Button 'main1'" name="main1"
                onclick="alert('Event is now at the button object level ('
                        + getPhase(event)
                        + ').')" />
      </form>
   </body>
</html>
```

In addition to the W3C DOM `stopPropagation()` method, NN6+, Moz, Safari, Opera, and Chrome also support IE's `cancelBubble` property for syntactical convenience.

Preventing W3C event default action

The W3C DOM counterpart to IE's `returnValue` property is the event object's `preventDefault()` method. Invoke this method in an event handler function when you wish to block the element's default action to the event:

```
evt.preventDefault();
```

Redirecting W3C DOM events

The mechanism for sending an event to an object outside the normal propagation pattern in W3C is similar to that of IE4+, although with different syntax and an important requirement. In place of the IE4+ `fireEvent()` method, NN6+/Moz and Gecko-, WebKit-, and Presto-based browsers use the W3C DOM `dispatchEvent()` method. The sole parameter of the method is an event object, but it cannot be an event object that is already propagating through the element hierarchy. Instead, you must create a new event object via a W3C DOM event object constructor (described later in this chapter). Listing 32-5 is the same as the IE4+ Listing 32-2, but with just a few modifications to run in the W3C event model. Notice that the `dispatchEvent()` method passes a newly created event object as its sole parameter.

LISTING 32-5

Cancelling and Redirecting Events in the W3C DOM

```
<!DOCTYPE html>
<html onclick="revealEvent('HTML', event)">
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Event Cancelling & Redirecting</title>
      <script type="text/javascript">
         // display alert with event object info
         function revealEvent(elem, evt)
         {
            var msg = "Event (from " + evt.target.tagName + " at ";
            msg += evt.clientX + "," + evt.clientY + ") is now at the ";
            msg += elem + " element.";
            alert(msg);
```

continued

LISTING 32-5 *(continued)*

```
        }
        function init()
        {
           document.onclick = docEvent;
           document.body.onclick = docBodEvent;
        }
        function docEvent(evt)
        {
           revealEvent("document", evt);
        }
        function docBodEvent(evt)
        {
           revealEvent("BODY", evt);
        }
        function buttonEvent(form, evt)
        {
           revealEvent("BUTTON", evt);
           // redirect if checked
              if (form.redirect.checked)
              {
                 document.body.dispatchEvent(evt);
              }
           // cancel if checked
              if (form.bubbleCancelState.checked)
              {
                 evt.stopPropagation();
              }
        }
     </script>
  </head>
  <body onload="init()">
     <h1>Event Cancelling & Redirecting</h1>
     <hr />
     <form onclick="revealEvent('FORM', event)">
        <p><button name="main1" onclick="buttonEvent(this.form, event)">Button
             'main1'</button>
        </p>
        <p><input type="checkbox" name="bubbleCancelState"
                  onclick="event.stopPropagation()" />Cancel Bubbling at BUTTON
           <br />
           <input type="checkbox" name="redirect"
                  onclick="event.stopPropagation()" /> Redirect Event to BODY
        </p>
     </form>
  </body>
</html>
```

Referencing the event Object

Just as there are two different event object models in today's browsers, the way your scripts access those objects is divided into two camps: the IE way and the W3C way. We start with the simpler, IE way.

In IE4+, the `event` object is accessible as a property of the `window` object:

```
window.event
```

But, as you are well aware, the `window` part of references is optional, so your scripts can treat the `event` object as if it were a global reference:

```
event.propertyName
```

Thus, any statement in an event handler function can access the `event` object without any special preparation or initializations.

The situation is a bit more complicated in the W3C event model. In some cases you must explicitly pass the event object as a parameter to an event handler function, whereas in other cases, the event object is delivered as a parameter automatically. The difference depends on how the event handler function is bound to the object.

Binding Events

Perhaps the most important facet of event handling in any script is binding an event to an element on the page. There are several different ways that you can carry out this event binding, and as you might expect, they aren't all compatible across different browsers. Furthermore, some of the techniques are considered passé in the sense that they have been improved upon by more modern approaches. Following are the four main techniques that can be used to bind events to elements:

- Assignment through tag attributes
- Assignment through object properties
- Attachment in IE
- Event listeners in NN/Moz/W3C

The following sections explore these event binding options in more detail, with an emphasis on showing you how to craft a modern, cross-browser event binding function based upon the last two techniques listed.

Binding events through tag attributes

Dating back to some of the earliest JavaScript-powered browsers, the original way of binding event handlers to objects is through an attribute in the element's tag. To bind an event in this manner, you simply assign inline JavaScript code in the attribute of an element, as in the following:

```
<input type="button" value="Click Me" onclick="handleClick();" />
```

The attribute name is the name of the event being handled, and its value is inline JavaScript code that is executed upon the event firing. You can include multiple statements in the event attribute, as this code reveals:

```
<input type="button" value="Click Me"
onclick="doSomething(this); doSomethingElse(this.form);" />
```

For modern browsers that support the W3C event model (NN6+/Moz and Gecko, WebKit- and Presto-based browsers), if you intend to inspect properties of the event within the event handler function, you must specify the event object as a parameter by passing `event` as a parameter, as in:

```
<input type="button" value="Click Me" onclick="handleClick(event);" />
```

This is the only time in the W3C model that you see an explicit reference to the `event` (lowercase `e`) object as if it were a global reference. This reference does not work in any other context — only as a parameter to an event handler function. If you have multiple parameters, the `event` reference can go in any order, but we tend to put it last:

```
<input type="button" value="Click Me" onclick="doSomething(this, event);" />
```

The function definition that is bound to the element should therefore have a parameter variable in place to catch the event object parameter:

```
function doSomething(widget, evt) {...}
```

You have no restrictions on how you name this parameter variable. In some examples of this book, you may see the variable assigned as `event` or, more commonly, `evt`. When working with cross-browser scripts, avoid using `event` as a parameter variable name so as not to interfere with the Internet Explorer `window.event` property.

The good news is that binding an event through an event tag attribute works well across all browsers. The bad news is that it goes against the prevailing trend in web design, which is to separate HTML content from the code that makes it interactive. In other words, there is a concerted effort among web developers to clearly delineate JavaScript code from HTML code.

This concept is closely related to the notion of separating content from presentation, which is afforded by style sheets. In this way, you could think of a web page as having three distinct components: HTML content, CSS, and JavaScript code. Keeping these three components as compartmentalized as possible results in cleaner, more manageable code.

Note

You see many examples of event tag attribute binding throughout this book during demonstrations of various objects, properties, and methods. The usage is intentional because it is generally easier to understand the concepts under discussion when the events are bound closely to the elements. ■

The trick to maintaining a clean separation between JavaScript event binding and HTML code is to bind the events purely within script code as opposed to within attributes of HTML elements. The latter three event binding approaches mentioned earlier all offer this separation.

Binding events through object properties

Dating back as far as NN3 and IE4, element objects have event properties that can be used to bind events by assignment. For every event that an element is capable of receiving and responding to, there is a suitably named property, in all lowercase. For example, the `button` element object has a property named `onclick` that corresponds to the `onclick` event. You can bind an event handler to a `button` element by assigning a function reference to the `onclick` property:

```
document.forms[0].myButton.onclick = handleClick;
```

Note

Although event properties should be specified in all lowercase (`onclick`), some browsers also recognize mixed case event names (`onClick`). ■

One catch to binding events as object properties is that at first glance it doesn't appear to be possible to pass your own parameters to the invoked handler functions. W3C browsers pass an event object as the only parameter to event handler functions, and this doesn't exactly leave room for you to include your own parameters. Without any further trickery, this means that your functions should receive the passed event object in a parameter variable:

```
function doSomething(evt) {...}
```

Recall that the `event` object contains a reference to the object that was the target of the event. From that, you can access any properties of that object, such as the `form` object that contains a form control object.

It is in fact perfectly possible to pass along your own parameters; it just takes an intermediary anonymous function to do the go-between work. For example, the following code demonstrates how to pass a single custom parameter along with the standard `event` object:

```
document.forms[0].myButton.onclick =
   function(evt) {doSomething("Cornelius", evt);};
```

In this example, a name string is passed along as the first parameter to the event handler, whereas the `event` object (automatically passed to the anonymous function as its sole parameter, and assigned to the parameter variable `evt`) is routed along as the second parameter. The actual handler code would look something like this:

```
function doSomething(firstName, evt) {...}
```

The `evt` parameter variable in the `doSomething()` event handler function acts as a reference to the `event` object for statements within the function. If you need to invoke other functions from there, you can pass the `event` object reference further along as needed. The `event` object retains its properties as long as the chain of execution triggered by the event action continues.

Binding events through IE attachments

In IE5 Microsoft set out to establish a new means of binding events to elements through attachments, which were originally intended for use with IE behaviors (see Chapter 51, "Internet Explorer Behaviors," on the CD-ROM). Eventually, the attachment approach to event binding expanded beyond

behaviors and became the de facto IE standard for event binding. Seeing as how IE (as of version 8) still does not support the W3C approach to binding events, which you see in the next section, you should consider attachments the preferred way of handling events in IE for the foreseeable future.

IE event attachments are managed through the `attachEvent()` and `detachEvent()` methods, which are supported by all element objects that are capable of receiving events. By using both of these methods, you can bind and unbind events throughout the course of an application as needed.

The `attachEvent()` method takes the following form:

```
elementReference.attachEvent("event", functionReference);
```

To put this form in perspective, the following is an example of binding an event using the `attachEvent()` method in IE:

```
document.getElementById("myButton").attachEvent("onclick", doSomething);
```

The first parameter to the `attachEvent()` is the string name of the event, including the `on` prefix, as in `"onclick"`. The second parameter is a reference to the event handler function for the event.

One new power afforded by IE event attachment is the ability to attach the same event to the same element multiple times (presumably pointing to different event handler functions). Just remember that if you choose to bind multiple events of the same type to the same element, they will be processed in the reverse order that they were assigned. This means the first event added is processed last.

Since the IE event model is predicated on the `event` object, which is a property of the `window` object, there is no `event` object passed into the event handler function. To access event properties, you simply access the window's `event` object using either `window.event` or just `event`. The latter approach works because the `window` object is always assumed in client-side scripting. The upcoming section "event Object Compatibility" shows how to reconcile the IE `window.event` property and the W3C `event` event handler parameter.

The IE event binding approach also offers the ability to unbind an event, which means the targeted element will no longer receive event notifications. You unbind an IE event by calling the `detachEvent()` method on the element, like this:

```
document.getElementById("myButton").detachEvent("onclick", doSomething);
```

This example reveals how the `detachEvent()` method relies on the exact same syntax as `attachEvent()`.

Binding events through W3C listeners

The W3C approach to binding events is logically similar to IE event attachment in that it revolves around two methods: `addEventListener()` and `removeEventListener()`. These two methods give elements the ability to listen for events and then respond accordingly. Also similarly to the IE `attachEvent()` and `detachEvent()` methods, `addEventListener()` and `removeEventListener()` work as a pair for adding and removing event listeners, respectively.

The `addEventListener()` method takes the following form:

```
elementReference.addEventListener("eventType", functionReference,
   captureSwitch);
```

The following example should help to reveal the practical usage of the method:

```
document.getElementById("myButton").addEventListener("click", doSomething,
   false);
```

Note how the event name is specified without the `on` prefix, which is different from the name used in IE event attachments. The other notable difference in W3C event listeners as compared to IE event attachments involves the third parameter to `addEventListener()`, `captureSwitch`, which determines whether the element should listen for the event during the capture phase of event propagation. Later in the chapter you learn about event propagation and how this parameter might be used to tweak the propagation of an event. For now, just know that the parameter is typically set to `false`.

Similar to IE event attachments, you can add the same event listener to the same element multiple times. Unlike the IE approach, however, is the fact that W3C events added in this manner are processed in the same order that they were assigned. This means that the first event added is processed first.

Another similarity the W3C event model has to IE event handling is the ability to unbind an event from an element. The W3C version of event unbinding involves the `removeEventListener()` method, which is demonstrated in this example:

```
document.getElementById("myButton").removeEventListener("click", doSomething,
   false);
```

This example shows how the `removeEventListener()` method accepts the same parameters as `addEventListener()`.

A cross-browser event binding solution

Pulling together what you've learned about modern event handling, you know it must be possible to reconcile the IE and W3C approaches to event binding. In fact, it doesn't take all that much extra code to bind events in a manner that cleanly attempts to use the latest event binding techniques while still gracefully falling back on an older technique (object properties) for legacy browsers.

Following is a cross-browser function you can use to add an event binding to an element:

```
function addEvent(elem, evtType, func)
{
   if (elem.addEventListener)
   {
      elem.addEventListener(evtType, func, false);
   }
   else if (elem.attachEvent)
   {
      elem.attachEvent("on" + evtType, func);
   }
   else
   {
      elem["on" + evtType] = func;
   }
}
```

Parameters for the function are a reference to the element, a string of the event type (that is, the version without the `on` prefix), and a reference to the function to be invoked when the event fires

on the element. The addEvent() function first attempts to use the addEventListener() method on the supplied element, which satisfies modern W3C browsers (NN6+/Mozilla/Safari/Opera/Chrome). If that fails, attachEvent() is tried, which accommodates modern IE browsers (IE5+). If that's a bust, the function falls back on simply assigning the event handler function to the event object property, which works on the vast majority of browsers.

Note

You could easily extend the addEvent() **function to allow for the** captureSwitch **parameter of the** addEventListener() **method by adding a fourth parameter and passing it to** addEventListener()**, instead of passing** false**.** ■

Of course, the addEvent() function has to get called in order to bind events for a page. The onload event provides a great opportunity for binding events but, as you know, it's not a good idea to just call the addEvent() function in the onload HTML attribute. That would go against everything you've just learned. The trick is to first add an anonymous event handler for the onload event, and then carry out your other event bindings within that function. Here's an example of how you might do this:

```
addEvent(window, "load", function()
{
   addEvent(document.getElementById("myButton"), "click", handleClick);
   addEvent(document.body, "mouseup",
      function(evt) {handleClick(evt);});
});
```

In case you need to unbind an event, here is a suitable cross-browser function for unbinding events:

```
function removeEvent(elem, evtType, func)
{
   if (elem.removeEventListener)
   {
      elem.removeEventListener(evtType, func, false);
   }
   else if (elem.detachEvent)
   {
      elem.detachEvent("on" + evtType, func);
   }
   else
   {
      elem["on" + evtType] = null;
   }
}
```

event Object Compatibility

Despite the incompatible ways that W3C DOM and IE event objects arrive at an event handler function, you can easily stuff the object into one variable that both browser types can use. For example, the following function fragment receives a W3C DOM event object but also accommodates the IE event object:

```
function doSomething(evt)
```

```
{
   evt = (evt) ? evt : ((window.event) ? window.event : null);
   if (evt)
   {
      // browser has an event to process
      ...
   }
}
```

If an `event` object arrives as a parameter, it continues to be available as `evt`; but if not, the function makes sure that a `window.event` object is available and assigns it to the `evt` variable; finally, if the browser doesn't know about an `event` object, the `evt` variable is made `null`. Processing continues only if `evt` contains an event object.

That's the easy part. The madness comes in the details: reading properties of the event object when the property names can vary widely across the two event object models. Sections later in this chapter provide specifics of each property and method of both event object models, but seeing an overview of the property terminology on a comparative basis is helpful. Table 32-2 lists the common information bits and actions you are likely to want from an event object, and the property or method names used in the event object models.

TABLE 32-2

Common event Object Properties and Methods

Property/Action	IE4+	W3C DOM
Target element	`srcElement`	`target`
Event type	`type`	`type`
X coordinate in element	`offsetX`	n/a†
Y coordinate in element	`offsetY`	n/a†
X coordinate on page	n/a†	`pageX`††
Y coordinate on page	n/a†	`pageY`††
X coordinate in window	`clientX`	`clientX`
Y coordinate in window	`clientY`	`clientY`
X coordinate on screen	`screenX`	`screenX`
Y coordinate on screen	`screenY`	`screenY`
Mouse button	`button`	`button`
Keyboard key	`keyCode`	`keyCode`
Shift key pressed	`shiftKey`	`shiftKey`
Alt key pressed	`altKey`	`altKey`
Ctrl key pressed	`ctrlKey`	`ctrlKey`

TABLE 32-2 *(continued)*

Property/Action	IE4+	W3C DOM
Previous Element	fromElement	relatedTarget
Next Element	toElement	relatedTarget
Cancel bubbling	cancelBubble	stopPropagation()
Prevent default action	returnValue	preventDefault()

†Value can be derived through calculations with other properties.
††Not an official W3C DOM property, but is supported in Mozilla, Safari, Opera, and Chrome.

As you can see in Table 32-2, properties for the IE4+ and W3C event objects have a lot in common. Perhaps the most important incompatibility to overcome is referencing the element that is the intended target of the event. This, too, can be branched in your code to achieve a common variable that references the element. For example, embedded within the previous function fragment can be a statement, such as the following:

```
var elem = (evt.target) ? evt.target : ((evt.srcElement) ?
   evt.srcElement : null);
```

Each event model has additional properties that are not shared by the other. Details about these are covered in the rest of this chapter.

Dueling Event Models

Despite the sometimes widely divergent ways event object models treat their properties, accommodating a wide range of browsers for event manipulation is not difficult. In this section, you see two scripts that examine important event properties. The first script reveals which, if any, modifier keys are held down during an event; the second script extracts the codes for both mouse buttons and keyboard keys. Both scripts work with all modern browsers that have event objects.

Cross-platform modifier key check

Listing 32-6 demonstrates branching techniques for examining the modifier key(s) being held down while an event fires. You can find details of the event object properties, such as `modifiers` and `altKey`, later in this chapter. To see the page in action, click a link, type into a text box, and click a button while holding down any combination of modifier keys. A series of four check boxes representing the four modifier keys is at the bottom. As you click or type, the check box(es) of the pressed modifier key(s) become checked.

LISTING 32-6

Checking Events for Modifier Keys

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
```

```
    <title>Event Modifiers</title>
    <script type="text/javascript">
       function checkMods(evt)
       {
          evt = (evt) ? evt : ((window.event) ? window.event : null);
             if (evt)
             {
                var elem = (evt.target) ? evt.target : evt.srcElement;
                var form = document.output;
                form.modifier[0].checked = evt.altKey;
                form.modifier[1].checked = evt.ctrlKey;
                form.modifier[2].checked = evt.shiftKey;
                form.modifier[3].checked = false;
             }
          return false;
       }

       // bind the event handlers
       function addEvent(elem, evtType, func)
       {
          if (elem.addEventListener)
          {
             elem.addEventListener(evtType, func, false);
          }
          else if (elem.attachEvent)
          {
             elem.attachEvent("on" + evtType, func);
          }
          else
          {
             elem["on" + evtType] = func;
          }
       }

       addEvent(window, "load", function()
       {
          addEvent(document.getElementById("link"), "mousedown",
             function(evt) {return checkMods(evt);});
          addEvent(document.getElementById("text"), "keyup",
             function(evt) {checkMods(evt);});
          addEvent(document.getElementById("button"), "click",
             function(evt) {checkMods(evt);});
       });
    </script>
 </head>
 <body>
    <h1>Event Modifiers</h1>
    <hr />
    <p>Hold one or more modifier keys and click on
       <a id="link" href="javascript:void(0)">this link</a>
       while looking at the checkboxes to see which keys you are holding.
    </p>
    <form name="output">
```

continued

LISTING 32-6 *(continued)*

```
         <p>Enter some text with uppercase and lowercase letters
            (while looking at the checkboxes):
            <input id="text" type="text" size="40" />
         </p>
         <p>
            <input id="button" type="button"
            value="Click Here With Modifier Keys" />
         </p>
         <p>
            <input type="checkbox" name="modifier" />Alt
            <input type="checkbox" name="modifier" />Control
            <input type="checkbox" name="modifier" />Shift
            <input type="checkbox" name="modifier" />Meta
         </p>
      </form>
   </body>
</html>
```

The script checks the event object property for each of three modifiers to determine which, if any, modifier keys are being pressed.

Cross-platform key capture

To demonstrate keyboard events in both event capture models, Listing 32-7 captures the key character being typed into a text box, as well as the mouse button used to click a button.

LISTING 32-7

Checking Events for Key and Mouse Button Pressed

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Button and Key Properties</title>
      <script type="text/javascript">
         function checkWhich(evt)
         {
            evt = (evt) ? evt : ((event) ? event : null);
               if (evt)
               {
                  var thingPressed = "";
                  var elem = (evt.target) ? evt.target : evt.srcElement;
                     if (elem.type == "textarea")
                     {
                        thingPressed = (evt.charCode) ? evt.charCode : evt.keyCode;
                     }
                     else if (elem.type == "button")
                     {
                        thingPressed = (typeof evt.button !=
```

```
                    "undefined") ? evt.button : "n/a";
                 }
              alert(thingPressed);
           }
        return false;
     }

     // bind the event handlers
     function addEvent(elem, evtType, func)
     {
        if (elem.addEventListener)
        {
           elem.addEventListener(evtType, func, false);
        }
        else if (elem.attachEvent)
        {
           elem.attachEvent("on" + evtType, func);
        }
        else
        {
           elem["on" + evtType] = func;
        }
     }

     addEvent(window, "load", function()
     {
        addEvent(document.getElementById("button"), "mousedown",
                 function(evt) {checkWhich(evt);});
        addEvent(document.getElementById("text"), "keypress",
                 function(evt) {checkWhich(evt);});
     });
  </script>
</head>
<body>
  <h1>Button and Key Properties</h1>
  <hr />
  <form>
     <p>Mouse down atop this
        <input id="button" type="button" value="Button" />
        with either mouse button (if you have more than one).
     </p>
     <p>Enter some text with uppercase and lowercase letters:
        <textarea id="text" cols="40" rows="4" wrap="virtual"></textarea>
     </p>
  </form>
</body>
</html>
```

The codes displayed for each keyboard event are equivalent to the ASCII values of character keys. If you need the codes of other keys, the `onkeydown` and `onkeyup` event handlers provide Unicode values for any key that you press on the keyboard. See the `charCode` and `keyCode` property listings for event objects later in this chapter for more details.

Event Types

Although browsers prior to version 4 did not have an accessible event object, this is a good time to summarize the evolution of what in today's browsers is known as the `type` property. The `type` property reveals the kind of event that generates an event object (the event handler name minus the "on"). Object models in IE4+ and NN6+/W3C provide event handlers for virtually every HTML element, so that it's possible, for example, to define an `onclick` event handler for not only a clickable button, but also for a `p` or even an arbitrary `span` element.

Older Browsers

Earlier browsers tended to limit the number of event handlers for any particular element to just those that made sense for the kind of element it was. Even so, many scripters wanted more event handlers on more objects. But until that became a reality in IE4+ and NN6+/W3C, authors had to know the limits of the object models. Table 32-4 shows the event handlers available for objects within three generations of early browsers. Each column represents the version in which the event type was introduced. For example, the `window` object started out with four event types, and gained three more when NN4 was released. In contrast, the `area` object was exposed as an object for the first time in NN3, which is where the first event types for that object are listed.

With the exception of the NN4 `layer` object, all objects shown in Table 32-4 have survived into the newer browsers, so that you can use these event handlers with confidence. Again, keep in mind that of the browsers listed in Table 32-4, only NN4 has an `event` object of any kind exposed to scripts.

TABLE 32-3

Event Types through the Early Ages

Object	NN2/IE3	NN3	NN4
`window`	`blur`		`dragdrop`
	`focus`		`move`
	`load`		`resize`
	`unload`		
`layer`			`blur`
			`focus`
			`load`
			`mouseout`
			`mouseover`
			`mouseup`

Object	NN2/IE3	NN3	NN4
link	click	mouseout	dblclick
	mouseover		mousedown
			onmouseup
area		mouseout	click
		mouseover	
image		abort	
		error	
		load	
form	submit	reset	
text, textarea, password			
	blur		keydown
	change		keypress
	focus		keyup
	select		
all buttons	click		mousedown
			mouseup
select	blur		
	change		
	focus		
fileupload		blur	
		focus	
		select	

Event types in IE4+ and NN6+/W3C

By now you should have at least scanned the list of event handlers defined for elements in common, as shown in Chapter 26. This list of event types is enormous. A sizable number of the event types are unique to IE4, IE5, and IE5.5+, and in some cases, just the Windows version at that.

If you compose pages for both IE4+ and NN6+/W3C, however, you need to know which event types these browser families and generations have in common. Event types for NN6+/Moz and Gecko-, WebKit-, and Presto-based browsers, are based primarily on the W3C DOM Level 2 specification, although they also include keyboard events, whose formal standards are still under development for DOM Level 3. Table 32-4 lists a common denominator of event types for modern browsers and the

TABLE 32-4

IE4+ and W3C DOM Event Types in Common

Event Type	Applicable Elements
abort	object
blur	window, button, text, password, label, select, textarea
change	text, password, textarea, select
click	All elements
error	window, frameset, object
focus	window, button, text, password, label, select, textarea
keydown	text, password, textarea
keypress	text, password, textarea
keyup	text, password, textarea
load	window, frameset, object
mousedown	All elements
mousemove	All elements
mouseout	All elements
mouseover	All elements
mouseup	All elements
reset	form
resize	window
scroll	window
select	text, password, textarea
submit	form
unload	window, frameset
IE4+ event Object	
altKey	
altLeft	
behaviorCookie	
behaviorPart	

IE4+ event Object
bookmarks
boundElements
button
cancelBubble
clientX
clientY
contentOverflow
ctrlKey
ctrlLeft
dataFld
dataTransfer
fromElement
keyCode
nextPage
offsetX
offsetY
propertyName
qualifier
reason
recordset
repeat
returnValue
saveType
screenX
screenY
shiftKey
shiftLeft
srcElement

continued

TABLE 32-4 *(continued)*

IE4+ event Object
srcFilter
srcUrn
toElement
type
wheelData
X
Y

objects that support them. Although not as long as the IE event list, the event types in Table 32-4 are the basic set you should get to know for all browsers.

Syntax

Accessing IE4+ `event` object properties:

```
[window.]event.property
```

Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-, Opera-, Chrome-

About this object

The IE4+ `event` object is a property of the `window` object. Its basic operation is covered earlier in this chapter.

You can see a little of what the `event` object is about with the help of The Evaluator (see Chapter 4, "JavaScript Essentials"). If you type `event` into the bottom text box, you can examine the properties of the `event` object for the event that triggers the function that displays the `event` object properties. If you press the Enter key in the text box, you see properties of the `keypress` event that caused the internal script to run; click the List Properties button to see the properties of the `click` event fired at the button. Hold down some of the modifier keys while clicking to see how this affects some of the properties.

As you review the properties for the `event` object, make special note of the compatibility rating for each property. The list of properties for this object has grown over the evolution of the IE4+ event object model. Also, most properties are listed here as being read-only, which they were in IE4. But for IE5+, these properties are also Read/Write if the event is created artificially via methods, such as IE5.5+'s `document.createEventObject()` method. Event objects that are created by user or system action have very few properties that can be modified on the fly (to prevent your scripts from altering user actions). Notice, too, that some properties are the same as for the W3C DOM event object, as revealed in the compatibility ratings.

Properties

altKey
ctrlKey
shiftKey

Value: Boolean Read-Only

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

When an `event` object is created in response to a user or system action, these three properties are set based on whether their corresponding keys were being held down at the time — a Shift-click, for example. If the key was held down, the property is assigned a value of `true`; otherwise the value is `false`.

Most commonly, you use expressions consisting of this property as `if` construction conditional statements. Because these are Boolean values, you can combine multiple properties in a single condition. For example, if you have a branch of a function that is to execute only if the event occurred with both the Shift and Control keys held down, the condition looks as the following:

```
if (event.shiftKey && event.ctrlKey)
{
   // statements to execute
}
```

Conversely, you can take a more user-friendly approach to provide special processing if the user holds down any one of the three modifier keys:

```
if (event.shiftKey || event.ctrlKey || event.altKey)
{
   // statements to execute
}
```

The rationale behind this approach is to offer perhaps some shortcut operation for users, but not force them to memorize a specific modifier key combination.

Example

See Listing 32-6, where the values of these three properties are used to set the `checked` properties of corresponding check boxes for a variety of event types.

Related Items: `altLeft`, `ctrlLeft`, `shiftLeft` properties

altLeft
ctrlLeft
shiftLeft

Value: Boolean Read-Only

Compatibility: WinIE5.5+, MacIE-, NN-, Moz-, Safari-, Opera-, Chrome-

Some versions of Windows allow events to be modified by only the left-hand Alt, Ctrl, and Shift keys when using IE5.5+. For these modifiers to be recorded by the `event` object, focus must be on the document (body), and not in any form control. If the left-key version is `false` and the regular version is `true`, then your script knows that the right-hand key had been held down during the event.

Related Items: `altKey`, `ctrlKey`, `shiftKey` properties

behaviorCookie
behaviorPart

Value: Integer Read-Only

Compatibility: WinIE6+, MacIE-, NN-, Moz-, Safari-, Opera-, Chrome-

These two properties are related to a Windows technology that Microsoft calls *rendering behaviors*. Unlike the behaviors discussed under the `addBehavior()` method in Chapter 26, rendering behaviors are written in C++ and provide services for custom drawing on your web page. For more details, consult the document "Implementing Rendering Behaviors" at `http://msdn.microsoft.com/en-us/library/aa753628%28VS.85%29.aspx`.

bookmarks
boundElements
dataFld
qualifier
reason
recordset

Value: See text Read-Only

Compatibility: WinIE6+, MacIE-, NN-, Moz-, Safari-, Opera-, Chrome-

TABLE 32-5

ADO-Related event Object Properties

Property	Value	First Implemented	Description
`bookmarks`	Array	IE4	Array of ADO bookmarks (saved positions) for records within a recordset associated with the object that received the event.
`boundElements`	Array	IE5	Array of element references for all elements bound to the same data set that was touched by the current event.
`dataFld`	String	IE5	Name of the data source column that is bound to a table cell that receives a `cellchange` event.
`qualifier`	String	IE5	Name of the data member associated with a data source that receives a data-related event. Available only if the data source object (DSO) allows multiple-named data members or a qualifier has been explicitly set via the `datasrc` attribute of the bound element. Read-write in IE5+.
`reason`	Integer	IE4	Set only from `onDataSetComplete` event, provides the result code of the data set loading (`0`=successful; `1`=transfer aborted; `2`=other error).
`recordset`	Object	IE4	Reference to the current recordset in a data source object.

This group of `event` object properties is tied to using Data Binding in Windows versions of IE4+. Extensive details of Data Binding lie outside the scope of this book, but Table 32-5 provides a summary of these `event` object properties within that context (much of the terminology is used in Data Binding, but doesn't affect other scripting). For more details, search for ActiveX Data Objects (ADO) at `http://msdn.microsoft.com/en-us/library/default.aspx`.

Note

Although still supported in IE, Microsoft's original ADO technology has given way to ADO.NET, which is designed for tighter integration with Microsoft's .NET architecture. To learn more about the differences between the two technologies, visit `http://msdn.microsoft.com/library/en-us/dndotnet/html/adonetprogmsdn.asp`. ■

button

Value: Integer Read-Only

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

The `button` property reveals which button or buttons were pressed to activate a mouse event. If no mouse button is pressed to generate an event, this property is zero in IE. But integers 1 through 7 reveal single and multiple button presses, including the actions of three-button mice when they are recognized by the operating system. Integer values in IE correspond to buttons according to the following scheme:

Value	Description
0	No button
1	Left (primary) button
2	Right button
3	Left and right buttons together
4	Middle button
5	Left and middle buttons together
6	Right and middle buttons together
7	Left, middle, and right buttons together

Mouse buttons other than the primary one are easier to look for in `mousedown` or `mouseup` events rather than in `onclick` events. Be aware that as the user works toward pressing multiple buttons, each press fires a `mousedown` event. Therefore, if the user presses the left button first, the `mousedown` event fires, with the `event.button` property bearing the `1` value; as soon as the right button is pressed, the `mousedown` event fires again, but this time with an `event.button` value of `3`. If your script intends to perform special action with both buttons pressed, it should ignore and not perform any action for a single mouse button, because that one-button event will very likely fire in the process, disturbing the intended action.

Exercise caution when scripting the `event.button` property for both IE4+ and NN6+/Moz/W3C. The W3C DOM event model defines different button values for mouse buttons (`0`, `1`, and `2` for left, middle, and right) and no values for multiple buttons.

Example

See Listing 32-7, where the `event.button` property is revealed in the status bar. Try pressing individual mouse buttons on, for example, the screen button. Then try combinations, watching the results very closely in the status bar.

Related Items: None

cancelBubble

Value: Boolean Read/Write

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

The `cancelBubble` property (which sounds more as if it should be a method name) determines whether the current `event` object bubbles up any higher in the element containment hierarchy of the document. By default, this property is `false`, meaning that if the event is supposed to bubble, it will do so automatically.

To prevent event bubbling for the current event, set the property to `true` anywhere within the event handler function. As an alternative, you can cancel bubbling directly in an element's event handler attribute, as in the following:

```
onclick="doButtonClick(this); event.cancelBubble = true"
```

Cancelling event bubbling works only for the current event. The very next event to fire will have bubbling enabled (provided the event bubbles).

Example

See Listing 32-2 to see the `cancelBubble` property in action. Even though that listing has some features that apply to IE5.5+, the bubble cancelling demonstration works all the way back to IE4.

Related Item: `returnValue` property

clientX
clientY
offsetX
offsetY
screenX
screenY
x
y

Value: Integer Read/Write

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

An IE `event` object provides coordinates for an event in as many as four coordinate spaces: the element itself, the parent element of the event's target, the viewable area of the browser window, and the entire video screen. Unfortunately, misleading values can be returned by some of the properties that correspond to these coordinate spaces, as discussed in this section. Note that no properties provide the explicit position of an event relative to the entire page, in case the user has scrolled the window.

Starting with the innermost space — that of the element that is the target of the event — the `offsetX` and `offsetY` properties should provide pixel coordinates within the target element. This is how, for example, you could determine the click point on an image, regardless of whether the

image is embedded in the body or floating around in a positioned div. Windows versions through IE8 produce the correct values in most cases. But for some elements that are child elements of the body element, the vertical (y) value may be relative to the viewable window, rather than just the element itself. You can see an example of this when you work with Listing 32-8 and click the h1 or p elements near the top of the page. This problem does not affect MacIE, but there is another problem on Mac versions: If the page is scrolled away from its normal original position, the scrolled values are subtracted from the clientX and clientY values. This is an incompatibility bug, and you must take this error into account if you need click coordinates inside an element for a potentially scrolled page. This error correction must be done only for the Mac, because in Windows it works as is.

Extending scope to the offset parent element of the event's target, the x and y properties in IE5+ for Windows should return the coordinates for the event relative to the target's offset parent element (the element that can be found via the offsetParent property). For most non-positioned elements, these values are the same as the clientX and clientY properties because, as discussed in a moment, the offset parent element has a zero offset with *its* parent, the body. Observe an important caution about the x and y properties: In WinIE4 and through MacIE5, the properties do not take into account any offset parent locations other than the body. Even in WinIE5+, this property can give false readings in some circumstances. By and large, these two properties should not be used.

The next set of coordinates, clientX and clientY, are relative to the visible document area of the browser window. When the document is scrolled all the way to the top (or the document doesn't scroll at all), these coordinates are the same as the coordinates on the entire page. But because the page can scroll "underneath" the viewable window, the coordinates on the page can change if the page scrolls. Also, in the Windows versions of IE, you can actually register mouse events that are up to 2 pixels outside of the body element, which is weird, but true. Therefore, in WinIE, if you click the background of the body, the event fires on the body element, but the clientX/clientY values will be 2 pixels greater than offsetX/offsetY (they're equal in MacIE). Despite this slight discrepancy, you should rely on the clientX and clientY properties if you are trying to get the coordinates of an event that may be in a positioned element, but have those coordinates relative to the entire viewable window, rather than just the positioning context.

Taking the page's scrolling into account for an event coordinate is often important. After all, unless you generate a fixed-size window for a user, you don't know how the browser window will be oriented. If you're looking for a click within a specific region of the page, you must take page scrolling into account. The scrolling factor can be retrieved from the document.body.scrollLeft and document.body.scrollTop properties. When reading the clientX and clientY properties, be sure to add the corresponding scroll properties to get the position on the page:

```
var coordX = event.clientX + document.body.scrollLeft;
var coordY = event.clientY + document.body.scrollTop;
```

Do this in your production work without fail.

Finally, the screenX and screenY properties return the pixel coordinates of the event on the entire video screen. These properties would be more useful if IE provided more window dimension properties. In any case, because mouse events fire only when the cursor is somewhere in the content region of the browser window, don't expect to get the screen values of any place outside this region.

If these descriptions seem confusing to you, you are not alone. Throw in a few bugs, and it may seem like quite a mess. But think how you may use event coordinates in scripts. By and large, you want to know one of two types of mouse event coordinates: within the element itself and within the page. Use the offsetX/offsetY properties for the former; use clientX/clientY (plus the scroll property values) for the latter.

Although the coordinate properties are used primarily for mouse events, there is a little quirk that may let you determine if the user has resized the window via the maximize icon in the title bar

(on the Mac, this is called the zoom box) or the resize handle at the bottom-right corner of the screen. Mouse event coordinates are recorded in the `event` object for a `resize` event. In the case of the maximize icon, the `clientY` coordinate is a negative value (above the client space) and the `clientX` coordinate is within about 45 pixels of the previous width of the window (`document.body.clientWidth`). This, of course, happens after the window has resized, so it is not a way to prevent window resizing.

Example

Listing 32-8 provides readings of all event coordinate properties in an interactive way. An `onmousedown` event handler triggers all event handling, and you can click the mouse anywhere on the page to see what happens. You see the tag of the element targeted by the mouse event to help you visualize how some of the coordinate properties are determined. An image is encased inside a positioned `div` element to help you see what happens to some of the properties when the event is targeted inside a positioned element.

LISTING 32-8

IE4+ Event Coordinate Properties

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>X and Y Event Properties (IE4+ Syntax)</title>
      <script type="text/javascript">
         function checkCoords(evt)
         {
            evt = (evt) ? evt : ((window.event) ? window.event : null);
               if (evt)
               {
                  var elem = (evt.target) ? evt.target : evt.srcElement;
                  var form = document.forms[0];
                  form.srcElemTag.value = "<" + elem.tagName + ">";
                  form.clientCoords.value = evt.clientX + "," + evt.clientY;
                     if (typeof document.body.scrollLeft != "undefined")
                     {
                        form.pageCoords.value = (evt.clientX +
                        document.body.scrollLeft) +
                           "," + (evt.clientY + document.body.scrollTop);
                     }
                  form.offsetCoords.value = evt.offsetX + "," + evt.offsetY;
                  form.screenCoords.value = evt.screenX + "," + evt.screenY;
                  form.xyCoords.value = evt.x + "," + evt.y;
                     if (elem.offsetParent)
                     {
                        form.parElem.value = "<" + elem.offsetParent.tagName + ">";
                     }
                  return false;
               }
         }
         function handleSize(evt)
```

```
    {
       evt = (evt) ? evt : ((window.event) ? window.event : null);
          if (evt)
          {
             document.forms[0].resizeCoords.value =
                 evt.clientX + "," + evt.clientY;
          }
    }

    // bind the event handlers
    function addEvent(elem, evtType, func)
    {
       if (elem.addEventListener)
       {
          elem.addEventListener(evtType, func, false);
       }
       else if (elem.attachEvent)
       {
          elem.attachEvent("on" + evtType, func);
       }
       else
       {
          elem["on" + evtType] = func;
       }
    }
    addEvent(window, "load", function()
    {
       addEvent(document.body, "mousedown",
                function(evt) {checkCoords(evt);});
       addEvent(document.body, "resize",
                function(evt) {handleSize(evt);});
    });
  </script>
</head>
<body>
  <h1>X and Y Event Properties (IE4+ Syntax)</h1>
  <hr />
  <p>Click on any element to see the coordinate values
    for the event object.
  </p>
  <form name="output">
    <table>
       <tr>
          <td colspan="2">IE Mouse Event Coordinates:</td>
       </tr>
       <tr>
          <td align="right">srcElement:</td>
          <td>
             <input type="text" name="srcElemTag" size="10" />
          </td>
       </tr>
       <tr>
          <td align="right">clientX, clientY:</td>
```

continued

***eventObject*.clientX**

LISTING 32-8 *(continued)*

```
            <td>
               <input type="text" name="clientCoords" size="10" />
            </td>
            <td align="right">...With scrolling:</td>
            <td>
               <input type="text" name="pageCoords" size="10" />
            </td>
         </tr>
         <tr>
            <td align="right">offsetX, offsetY:</td>
            <td>
               <input type="text" name="offsetCoords" size="10" />
            </td>
         </tr>
         <tr>
            <td align="right">screenX, screenY:</td>
            <td>
               <input type="text" name="screenCoords" size="10" />
            </td>
         </tr>
         <tr>
            <td align="right">x, y:</td>
            <td>
               <input type="text" name="xyCoords" size="10" />
            </td>
            <td align="right">...Relative to:</td>
            <td>
               <input type="text" name="parElem" size="10" />
            </td>
         </tr>
         <tr>
            <td align="right">
               <input type="button" value="Click Here" />
            </td>
         </tr>
         <tr>
            <td colspan="2"><hr /></td>
         </tr>
         <tr>
            <td colspan="2">Window Resize Coordinates:</td>
         </tr>
         <tr>
            <td align="right">clientX, clientY:</td>
            <td>
               <input type="text" name="resizeCoords" size="10" />
            </td>
         </tr>
      </table>
   </form>
```

```
      <div id="display" style="position:relative; left:100">
         <img alt="image" src="nile.gif" width="320" height="240" border="0" />
      </div>
   </body>
</html>
```

Here are some tasks to try in IE with the page that loads from Listing 32-8 to help you understand the relationships among the various pairs of coordinate properties:

1. Click the dot above the "i" on the "Click Here" button label. The target element is the button (`input`) element, whose `offsetParent` is a table cell element. The `offsetY` value is very low because you are near the top of the element's own coordinate space. The client coordinates (and `x` and `y`), however, are relative to the viewable area in the window. If your browser window is maximized in Windows, the `screenX` and `clientX` values will be the same; the difference between `screenY` and `clientY` is the height of all the window chrome above the content region. With the window not scrolled at all, the client coordinates are the same with and without scrolling taken into account.
2. Jot down the various coordinate values and then scroll the page down slightly (clicking the scrollbar fires an event), and click the dot on the button again. The `clientY` value shrinks because the page has moved upward relative to the viewable area, making the measure between the top of the area smaller with respect to the button. The Windows version does the right thing with the offset properties, by continuing to return values relative to the element's own coordinate space; the Mac, unfortunately, subtracts the scrolled amount from the offset properties.
3. Click the large image. The client properties perform as expected for both Windows and Mac, as do the screen properties. For Windows, the `x` and `y` properties correctly return the event coordinates relative to the `img` element's `offsetParent`, which is the `div` element that surrounds it. Note, however, that the browser "sees" the `div` as starting 10 pixels to the left of the image. In WinIE5.5+, you can click within those 10 transparent pixels to the left of the image to click the `div` element. This padding is inserted automatically and impacts the coordinates of the `x` and `y` properties. A more reliable measure of the event inside the image is the offset properties. The same is true in the Macintosh version, as long as the page isn't scrolled, in which case the scroll, just as in Step 2, affects the values above.
4. Click the top `hr` element under the heading. It may take a couple of tries to actually hit the element (you've made it when the `hr` element shows up in the `srcElement` box). This is to reinforce the way the client properties provide coordinates within the element itself (again, except on the Mac when the page is scrolled). Clicking at the very left end of the rule, you eventually find the 0,0 coordinate.

Finally, if you are a Windows user, here are two examples to try to see some of the unexpected behavior of coordinate properties.

1. With the page not scrolled, click anywhere along the right side of the page, away from any text, so that the `body` element is `srcElement`. Because the `body` element theoretically fills the entire content region of the browser window, all coordinate pairs except for the screen coordinates should be the same. But offset properties are 2 pixels less than all the others. By and large, this difference won't matter in your scripts, but you should be aware of this potential discrepancy if precise positioning is important. For inexplicable reasons, the offset properties are measured in a space that is inset 2 pixels from the left and top of the window. This is not the case in the Macintosh version, where all value pairs are the same from the `body` perspective.

2. Click the text of the `h1` or `p` elements (just above and below the long horizontal rule at the top of the page). In theory, the offset properties should be relative to the rectangles occupied by these elements (they're block elements, after all). But instead, they're measured in the same space as the client properties (plus the 2 pixels). This unexpected behavior doesn't have anything to do with the cursor being a text cursor, because if you click inside any of the text box elements, their offset properties are properly relative to their own rectangles. This problem does not afflict the Macintosh version.

Many of these properties are also in the W3C DOM and are therefore supported in W3C DOM browsers. Unsupported properties display their values as `undefined` when you run Listing 32-8 in those browsers.

You can see further examples of important event coordinate properties in action in the discussion of dragging elements around the IE page in Chapter 43, "Positioned Objects."

Related Items: `fromElement`, `toElement` properties

contentOverflow

Value: Boolean Read-Only

Compatibility: WinIE5.5+, MacIE-, NN-, Moz-, Safari-, Opera-, Chrome-

The value indicates whether page content has overflowed the current layout rectangle. This property is primarily used during printing as the basis for overflowing content from one page to another.

dataTransfer

Value: Object Read-Only

Compatibility: WinIE5+, MacIE-, NN-, Moz-, Safari-, Opera-, Chrome-

The `dataTransfer` property is a reference to the `dataTransfer` object. Use this object in drag-and-drop operations (that is, with drag-and-drop-related events) to control not only the data that gets transferred from the source to the target, but also to control the look of the cursor along the way.

Table 32-6 lists the properties and methods of the `dataTransfer` object.

The `dataTransfer` object acts as a conduit and controller of the data that your scripts need to transfer from one element to another in response to a user's drag-and-drop action. You need to adhere to a well-defined sequence of actions triggered by a handful of event handlers. This means that the object is invoked on different instances of the `event` object as different events fire in the process of dragging and dropping.

The sequence begins at the source element, where an `ondragstart` event handler typically assigns a value to the `dropEffect` property and uses the `getData()` method to explicitly capture whatever data regarding the source object gets transferred to the eventual target. For example, if you drag an image, the information being transferred may simply be the URL of the image — data that is extractable from the `event.srcElement.src` property of that event (the `src` property of the image, that is).

At the target element(s), three event handlers must be defined: `ondragenter`, `ondragover`, and `ondrop`. Most commonly, the first two event handlers do nothing more than mark the element for a particular `dropEffect` (which must match the `effectAllowed` set at the source during the drag's start) and set `event.returnValue` to false so the desired cursor is displayed. These actions are

also carried out in the `ondrop` event handler, but that is also the handler that does the processing of the destination action at the target element. This is when the `dataTransfer` object's `getData()` method is invoked to pick up the data that has been "stored" away by `getData()` at the start of the drag. If you also want to make sure that the data is not picked up accidentally by another event, invoke the `clearData()` method to remove that data from memory.

TABLE 32-6

dataTransfer object Properties and Methods

Property/Method	Returns	Description
`dropEffect`	String	An element that is a potential recipient of a drop action can use the `ondragenter`, `ondragover`, or `ondrop` event handler to set the cursor style to be displayed when the cursor is atop the element. Before this can work, the source element's `ondragstart` event handler must assign a value to the `event.effectAllowed` property. Possible string values for both properties are `copy`, `link`, `move`, or `none`. These properties correspond to the Windows system cursors for the operations users typically do with files and in other documents. You must also cancel the default action (meaning set `event.returnValue` to `false`) for all of these drop element event handlers: `ondragenter`, `ondragover`, and `ondrop`.
`effectAllowed`	String	Set in response to an `ondragstart` event of the source element, this property determines which kind of drag-and-drop action will be taking place. Possible string values are `copy`, `link`, `move`, or `none`. This property value must match the `dropEffect` property value for the target element's event object. Also, cancel the default action (meaning, set `event.returnValue` to `false`) in the `ondragstart` event handler.
`clearData([format])`	Nothing	Removes data in the clipboard. If no format parameters are supplied, all data are cleared. Data formats can be one or more of the following strings: `Text`, `URL`, `File`, `HTML`, `Image`.
`getData(format)`	String	Retrieves data of the specified format from the clipboard. The format is one of the following strings: `Text`, `URL`, `File`, `HTML`, `Image`. The clipboard is not emptied after you get the data, so it can be retrieved in several sequential operations.
`setData(format, data)`	Boolean	Stores string data in the clipboard. The format is one of the following strings: `Text`, `URL`, `File`, `HTML`, `Image`. For non-text data formats, the data must be a string that specifies the path or URL to the content. Returns `true` if the transfer to the clipboard is successful.

Note that the style of dragging being discussed here is not the kind in which you see the source element actually moving on the screen (although you could script it that way). The intention is to treat drag-and-drop operations just as Windows does in, say, the Windows Explorer window or on the Desktop. To the user, the draggable component becomes encapsulated in the cursor. That's why the properties of the `dataTransfer` object control the appearance of the cursor at the drop point as a way of conveying to the user the type of action that will occur with the impending drop. Apple implements the same behavior in Safari 2.

Example

An extensive example of the `dataTransfer` property in action can be found in Listing 26-37 in the section for the `ondrag` event handler.

Related Items: `ondragend`, `ondragenter`, `ondragleave`, `ondragover`, `ondragstart`, `ondrop` event handlers

fromElement
toElement

Value: Element object — Read-Only

Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari+, Opera+, Chrome+

The `fromElement` and `toElement` properties allow an element to uncover where the cursor rolled in from or has rolled out to. These properties extend the power of the `onmouseover` and `onmouseout` event handlers by expanding their scope to outside the current element (usually to an adjacent element).

When the `onmouseover` event fires on an element, the cursor had to be over some other element just beforehand. The `fromElement` property holds a reference to that element. Conversely, when the `onmouseout` event fires, the cursor is already over some other element. The `toElement` property holds a reference to that element.

Example

Listing 32-9 provides an example of how the `fromElement` and `toElement` properties can reveal the life of the cursor action before and after it rolls into an element. When you roll the cursor to the center box (a table cell), its `onmouseover` event handler displays the text from the table cell from which the cursor arrived.

LISTING 32-9

Using the toElement and fromElement Properties

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>fromElement and toElement Properties</title>
      <style type="text/css">
         .direction {background-color:#00FFFF; width:100; height:50;
                    text-align:center;}
```

```
      #main {background-color:#FF6666; text-align:center;}
   </style>
   <script type="text/javascript">
      function showArrival(evt)
      {
         evt = (evt) ? evt : ((event) ? event : null);
         var direction = (event.fromElement.innerText) ?
            event.fromElement.innerText :
            "parts unknown";
         status = "Arrived from: " + direction;
      }
      function showDeparture(evt)
      {
         evt = (evt) ? evt : ((event) ? event : null);
         var direction = (event.toElement.innerText) ?
            event.toElement.innerText :
            "parts unknown";
         status = "Departed to: " + direction;
      }

      // bind the event handlers
      function addEvent(elem, evtType, func)
      {
         if (elem.addEventListener)
         {
            elem.addEventListener(evtType, func, false);
         }
         else if (elem.attachEvent)
         {
            elem.attachEvent("on" + evtType, func);
         }
         else
         {
            elem["on" + evtType] = func;
         }
      }
      addEvent(window, "load", function()
      {
         addEvent(document.getElementById("main"), "mouseover",
                  function(evt) {showArrival(evt);});
         addEvent(document.getElementById("main"), "mouseout",
                  function(evt) {showDeparture(evt);});
      });
   </script>
</head>
<body>
   <h1>fromElement and toElement Properties</h1>
   <hr />
   <p>Roll the mouse to the center box and look for arrival information in
      the status bar. Roll the mouse away from the center box and look for
      departure information in the status bar.</p>
   <table cellspacing="0" cellpadding="5">
      <tr>
```

continued

LISTING 32-9 *(continued)*

```
            <td></td>
            <td class="direction">North</td>
            <td></td>
         </tr>
         <tr>
            <td class="direction">West</td>
            <td id="main" onmouseover="showArrival()"
                onmouseout="showDeparture()">Roll</td>
            <td class="direction">East</td>
         </tr>
         <tr>
            <td></td>
            <td class="direction">South</td>
            <td></td>
         </tr>
      </table>
   </body>
</html>
```

This is a good example to experiment with in the browser, because it also reveals a potential limitation. The element registered as the `toElement` or `fromElement` must fire a mouse event to register itself with the browser. If not, the next element in the sequence that registers itself is the one acknowledged by these properties. For example, if you roll the mouse into the center box, and then extremely quickly roll the cursor to the bottom of the page, you may bypass the South box entirely. The text that appears in the status bar is actually the inner text of the `body` element, which is the element that caught the first mouse event to register itself as the `toElement` for the center table cell.

Related Item: `srcElement` property

keyCode

Value: Integer Read-Only

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

For keyboard events, the `keyCode` property returns an integer corresponding to the Unicode value of the character (for `onkeypress` events) or the keyboard character key (for `onkeydown` and `onkeyup` events). There is a significant distinction between these numbering code systems.

If you want the Unicode values (the same as ASCII values for the Latin character set) for the key that a user pressed, get the `keyCode` property from the `onkeypress` event handler. For example, a lowercase “a” returns 97, whereas an uppercase “A” returns 65. Non-character keys, such as arrows, page navigation, and function keys, return a `null` value for the `keyCode` property during `onkeypress` events. In other words, the `keyCode` property for `onkeypress` events is more like a character code than a key code.

To capture the exact keyboard key that the user presses, use either the `onkeydown` or `onkeyup` event handler. For these events, the `event` object captures a numeric code associated with a particular key on the keyboard. For the character keys, this varies with the language assigned as the

system language. Importantly, there is no distinction between uppercase or lowercase: The "A" key on the Latin keyboard returns a value of 65, regardless of the state of the Shift key. At the same time, however, the press of the Shift key fired its own `onkeydown` and `onkeyup` events, setting the `keyCode` value to 16. Other non-character keys — arrows, page navigation, function, and similar — have their own codes as well. This gets very detailed, and involves special key codes for the numeric keyboard keys that are different from their corresponding numbers along the top row of the alphanumeric keyboard.

Cross-Reference

Be sure to see the extensive section on keyboard events in Chapter 26 for examples of how to apply the `keyCode` **property in applications.** ■

Example

Listing 32-10 provides an additional play area to view the `keyCode` property for all three keyboard events while you type into a `textarea`. You can use this page later as an authoring tool to grab the precise codes for keyboard keys you may not be familiar with.

LISTING 32-10

Displaying keyCode Property Values

```
<html>
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>keyCode Property</title>
      <style type="text/css">
         td {text-align:center}
      </style>
      <script type="text/javascript">
         function showCode(which, evt)
         {
            evt = (evt) ? evt : ((event) ? event : null);
               if (evt)
               {
                  document.forms[0].elements[which].value = evt.keyCode;
               }
         }

         function clearEm()
         {
            for (var i = 1; i < document.forms[0].elements.length; i++)
            {
               document.forms[0].elements[i].value = "";
            }
         }

         // bind the event handlers
```

continued

LISTING 32-10 *(continued)*

```
      function addEvent(elem, evtType, func)
      {
         if (elem.addEventListener)
         {
            elem.addEventListener(evtType, func, false);
         }
         else if (elem.attachEvent)
         {
            elem.attachEvent("on" + evtType, func);
         }
         else
         {
            elem["on" + evtType] = func;
         }
      }
      addEvent(window, "load", function()
      {
         addEvent(document.getElementById("scratchpad"), "keydown",
                  function(evt) {clearEm(); showCode("down", evt);});
         addEvent(document.getElementById("scratchpad"), "keypress",
                  function(evt) {showCode("press", evt);});
         addEvent(document.getElementById("scratchpad"), "keyup",
                  function(evt) {showCode("up", evt);});
      });
   </script>
</head>
<body>
   <h1>keyCode Property</h1>
   <hr />
   <form>
      <p>
         <textarea id="scratchpad" name="scratchpad"
          cols="40" rows="5" wrap="hard"></textarea>
      </p>
      <table cellpadding="5">
         <tr>
            <th>Event</th>
            <th>event.keyCode</th>
         </tr>
         <tr>
            <td>onKeyDown:</td>
            <td><input type="text" name="down" size="3" /></td>
         </tr>
         <tr>
            <td>onKeyPress:</td>
            <td><input type="text" name="press" size="3" /></td>
         </tr>
         <tr>
            <td>onKeyUp:</td>
```

```
            <td><input type="text" name="up" size="3" /></td>
         </tr>
      </table>
   </form>
</body>
</html>
```

The following are some specific tasks to try with the page to examine key codes (if you are not using a browser set for English, and a Latin-based keyboard, your results may vary):

1. Enter a lowercase **a**. Notice how the `onkeypress` event handler shows the code to be 97, which is the Unicode (and ASCII) value for the first of the lowercase letters of the Latin alphabet. But the other two events record just the key's code: 65.
2. Type an uppercase **A** via the Shift key. If you watch closely, you see that the Shift key itself generates the code 16 for the `onkeydown` and `onkeyup` events. But the character key then shows the value 65 for all three events, because the ASCII value of the uppercase letter happens to match the keyboard key code for that letter.
3. Press and release the Down Arrow key (be sure the cursor still flashes in the `textarea`, because that's where the keyboard events are being monitored). As a non-character key, it does not fire an `onkeypress` event. But it does fire the other events, and assigns 40 as the code for this key.
4. Poke around with other non-character keys. Some may produce dialog boxes or menus, but their key codes are recorded nonetheless. Note that not all keys on a Macintosh keyboard register with MacIE.

Notice also that the `keyCode` property doesn't work properly for the `onkeypress` event in Mozilla-based browsers. This is because Mozilla uses the `charCode` property for the `onkeypress` event instead of `keyCode`. You could make the code in the listing work for all modern browsers with the following modification in the `showCode()` function:

```
if (evt)
{
   var charCode = (evt.charCode) ? evt.charCode : evt.keyCode;
   document.forms[0].elements[which].value = charCode;
}
```

Related Items: `onkeydown`, `onkeypress`, `onkeyup` event handlers

nextPage

Value: String — Read-Only

Compatibility: WinIE5.5+, MacIE-, NN-, Moz-, Safari-, Opera-, Chrome-

The `nextPage` property is applicable only if your WinIE5.5+ page uses a TemplatePrinter behavior. Values of this property are one of the following strings: `left`, `right`, or an empty string.

propertyName

Value: String — Read-Only

Compatibility: WinIE5+, MacIE-, NN-, Moz-, Safari-, Opera-, Chrome-

The `propertyName` property is filled only after an `onpropertychange` event fires.

If a script modifies a property, the `onpropertychange` event handler fires, and the string name of the property is stuffed into the `event.propertyName` property. If the property happens to be a property of the `style` object associated with the element, the `propertyName` is the full property reference, as in `style.backgroundColor`.

Example

See Listing 26-45 in the section about the `onpropertychange` event handler for an example of the values returned by this property.

Related Item: `onpropertychange` event handler (Chapter 26, "Generic HTML Element Objects")

repeat

Value: Boolean — Read-Only

Compatibility: WinIE5+, MacIE-, NN-, Moz-, Safari-, Opera-, Chrome-

The `repeat` property reveals for `onkeydown` events only whether the key is in repeat mode (as determined by the keyboard control panel settings in the system). With this information, you can prevent the automatic triggering of repeat mode from causing multiple characters from being recognized by the browser. This property can come in handy if users accidentally hold down a key too long. The following script fragment in an `onkeydown` event handler for a text box or `textarea` prevents multiple characters from appearing even if the system goes into repeat mode:

```
if (event.repeat)
{
   event.returnValue = false;
}
```

By disabling the default action while in repeat mode, no further characters reach the text box until repeat mode goes away (with the press of another key).

Related Item: `onkeydown` event handler

returnValue

Value: Boolean — Read-Only

Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari1.2+, Opera+, Chrome+

While IE4+ continues to honor the original way of preventing default action for an event handler (that is, having the last statement of the event handler evaluate to `return false`), the IE4+ event model provides a property that lets the cancellation of default action take place entirely within a function invoked by an event handler. By default, the `returnValue` property of the `event` object is `true`, meaning that the element processes the event after the scripted handler completes its job, just as if the script weren't there. Normal processing, for example, is displaying a typed character, navigating to a link's `href` URL upon being clicked, or submitting a form after the Submit button is clicked.

But you don't always want the default action to occur. For example, consider a text box that is supposed to allow only numbers to be typed in it. The `onkeypress` event handler can invoke a function that inspects each typed character. If the character is not a numeric character, it should not reach the

text box for display. The following validation function may be invoked from the `onkeypress` event handler of just such a text box:

```
function checkIt()
{
   var charCode = event.keyCode;
      if (charCode < 48 || charCode > 57)
      {
         alert("Please make sure entries are numerals only.");
         event.returnValue = false;
      }
}
```

By using this event handler, the errant character won't appear in the text box.

Note that this property is not a substitute for the `return` statement of a function. If you need a value to be returned to the invoking statement, you can use a `return` statement in addition to setting the `event.returnValue` property.

Example

You can find several examples of the `returnValue` property at work in Chapters 1 and 26. Specifically, refer to Listings 26-30, 26-33, 26-36, 26-37, 26-38, and 26-44. Moreover, many of the other examples in Chapter 26 can substitute the `returnValue` property way of cancelling the default action if the scripts were to be run exclusively on IE4+.

Related Item: `return` statement (Chapter 23, "Function Objects and Custom Objects")

saveType

Value: String Read-Only

Compatibility: WinIE5.5+, MacIE-, NN-, Moz-, Safari-, Opera-, Chrome-

The `saveType` property is assigned a value only when an `oncontentsave` event is bound to a WinIE DHTML behavior file. For more information about behaviors, see `http://msdn.microsoft.com/en-us/library/ms531079%28VS.85%29.aspx`.

Related Item: `addBehavior()` method

srcElement

Value: Element object reference Read-Only

Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari1.2+, Opera+, Chrome+

The `srcElement` property is a reference to the HTML element object that is the original target of the event. Because an event may bubble up through the element containment hierarchy and be processed at any level along the way, having a property that points back to the element from which the event originated is comforting. After you have a reference to that element, you can read or write any properties that belong to that element or invoke any of its methods.

Example

As a simplified demonstration of the power of the `srcElement` property, Listing 32-11 has but two event handlers defined for the `body` element, each invoking a single function. The idea is that the `onmousedown` and `onmouseup` events will bubble up from whatever their targets are, and the event handler functions will find out which element is the target and modify the color style of that element.

An extra flair is added to the script in that each function also checks the `className` property of the target element. If the `className` is `bold` — a class name shared by three `span` elements in the paragraph — the stylesheet rule for that class is modified so that all items share the same color. Your scripts can do even more in the way of filtering objects that arrive at the functions to perform special operations on certain objects or groups of objects.

Notice that the scripts don't have to know anything about the objects on the page to address each clicked one individually. That's because the `srcElement` property provides all of the specificity needed for acting on the target element.

LISTING 32-11

Using the srcElement Property

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>srcElement Property</title>
      <style type="text/css">
         .bold {font-weight:bold}
         .ital {font-style:italic}
      </style>
      <script type="text/javascript">
         function highlight()
         {
            var elem = event.srcElement;
               if (elem.className == "bold")
               {
                  document.styleSheets[0].rules[0].style.color = "red";
               }
               else
               {
                  elem.style.color = "#FFCC00";
               }
         }
         function restore()
         {
            var elem = event.srcElement;
               if (elem.className == "bold")
               {
                  document.styleSheets[0].rules[0].style.color = "";
               }
               else
               {
```

```
                elem.style.color = "";
             }
         }

         // bind the event handlers
         function addEvent(elem, evtType, func)
         {
            if (elem.addEventListener)
            {
               elem.addEventListener(evtType, func, false);
            }
            else if (elem.attachEvent)
            {
               elem.attachEvent("on" + evtType, func);
            }
            else
            {
               elem["on" + evtType] = func;
            }
         }
         addEvent(window, "load", function()
         {
            addEvent(document.body, "mousedown", highlight);
            addEvent(document.body, "mouseup", restore);
         });
      </script>
   </head>
   <body>
      <h1>srcElement Property</h1>
      <hr />
      <p>Click on the bolded text, then click on the unbolded text.
         What do you see happen?
      </p>
      <p>One event handler...</p>
      <ul>
         <li>Can</li>
         <li>Cover</li>
         <li>Many</li>
         <li>Objects</li>
      </ul>
      <p>Lorem ipsum dolor sit amet, consectetaur adipisicing elit,
         <span class="bold">sed do</span>
         eiusmod tempor incididunt
         <span class="ital">ut labore et</span>
         dolore magna aliqua. Ut enim adminim veniam,
         <span class="bold">quis nostrud exercitation</span>
         ullamco laboris nisi ut aliquip ex ea
         <span class="bold">commodo consequat</span>.
      </p>
   </body>
</html>
```

Related Items: `fromElement`, `toElement` properties

srcFilter

Value: String Read-Only

Compatibility: WinIE4+, MacIE-, NN-, Moz-, Safari-, Opera-, Chrome-

According to Microsoft, the `srcFilter` property should return a string of the name of the filter that was applied to trigger an `onfilterchange` event handler. While the property exists in the `event` object, its value is always `null`, at least through WinIE8.

Related Items: `onfilterchange` event handler; `style.filter` object

srcUrn

Value: String Read-Only

Compatibility: WinIE5+, MacIE-, NN-, Moz-, Safari-, Opera-, Chrome-

If an event is fired in a WinIE behavior attached to an element, and the behavior has a URN identifier defined for it, the `srcUrn` property returns the string from the URN identifier. For more information about behaviors, see `http://msdn.microsoft.com/en-us/library/ms531079%28VS.85%29.aspx`.

Related Item: `addBehavior()` method

toElement

(See `fromElement`)

type

Value: String Read-Only

Compatibility: WinIE4+, MacIE4+, NN4+, Moz+, Safari+, Opera+, Chrome+

You can find out what kind of event fired to create the current `event` object by way of the `type` property. The value is a string version of the event name — just the name of the event without the "on" prefix that is normally associated with event names in IE. This property can be helpful when you designate one event handler function to process different kinds of events. For example, both the `onmousedown` and `onclick` event handlers for an object can invoke one function. Inside the function, a branch is written for whether the type comes in as `mousedown` or `click`, with different processing for each event type. That is not to endorse such event handler function sharing, but for you to be aware of this power should your script constructions find the property helpful.

This property and its values are fully compatible with the NN6+/Moz/W3C event models.

Example

Use The Evaluator (Chapter 4, "JavaScript Essentials") to see values returned by the `type` property. Enter the following object name into the bottom text box and press Enter/Return:

```
event
```

If necessary, scroll the Results box to view the `type` property, which should read `keypress`. Now click the List Properties button. The type changes to `click`. The reason for these types is that the

`event` object whose properties are being shown here is the event that triggers the function to show the properties. From the text box, an `onkeypress` event handler triggers that process; from the button, an `onclick` event handler does the job.

Related Items: All event handlers (Chapter 26, "Generic HTML Element Objects")

wheelData

Value: Integer Read-Only

Compatibility: WinIE5.5+, MacIE-, NN-, Moz-, Safari-, Opera-, Chrome-

The `wheelData` property returns an integer indicating which direction the mouse wheel was rolled for an `onmousewheel` event. The values returned are typically either 120 or −120, with a positive value indicating that the mouse wheel was rolled toward the screen and a negative value indicating that the wheel was rolled the opposite direction.

NN6+/Moz event Object

Properties	Methods	Event Handlers
`altKey`	`initEvent()`	
`bubbles`	`initKeyEvent()`	
`button`	`initMouseEvent()`	
`cancelable`	`initMutationEvent()`	
`cancelBubble`	`initUIEvent()`	
`charCode`	`preventDefault()`	
`clientX`	`stopPropagation()`	
`clientY`		
`ctrlKey`		
`currentTarget`		
`detail`		
`eventPhase`		
`isChar`		
`isTrusted`		
`keyCode`		
`layerX`		
`layerY`		

Properties	Methods	Event Handlers
metaKey		
originalTarget		
pageX		
pageY		
screenX		
screenY		
shiftKey		
target		
timeStamp		
type		
view		

Syntax

Accessing NN6+/Moz event object properties and methods:

```
eventObject.property | method([parameters])
```

Compatibility: WinIE-, MacIE-, NN6+, Moz+, Safari+, Opera+, Chrome+

About this object

Although it is based largely on the event object as defined by the W3C DOM Level 2, the NN6+/Moz event object also carries forward several characteristics from the NN4 event object. A few properties are continued primarily for backward compatibility. But because future Mozilla development will likely forego the peculiarities of the NN4 DOM and event models, you should ignore these items (as highlighted below). Wherever possible, look forward and embrace the W3C DOM aspects of the event model. Safari, Opera, and Chrome, for example, implement a lot of the W3C DOM event model, but exclude all old NN4 properties.

Although the NN6+/Moz event model provides a bubbling event propagation model just as IE4+ does the incompatibility of referencing event objects between the event models is still there. In the W3C DOM (as in NN4), an event object is explicitly passed as a parameter to event handler (or, rather, event listener) functions. But after you have a browser-specific event object assigned to a variable inside a function, a few important properties have the same names between the IE4+ and W3C DOM event models. If Microsoft adopts more of the W3C DOM event model in future versions of IE, the compatibility situation should improve.

The event object discussed in this section is the instance of an event that is created as the result of a user or system event action. The W3C DOM includes an additional static `Event` object. Many of the properties of the static `Event` object are inherited by the event instances, so the detailed coverage of

those shared properties is in this section because it is the event object you'll be scripting for the most part.

In many code fragments in the following detail sections, you will see references that begin with the `evt` reference. This assumes that the statement(s) resides inside a function that has assigned the incoming event object to the `evt` parameter variable:

```
function myFunction(evt) {...}
```

As shown earlier in this chapter, you can equalize W3C DOM and IE4+ event object references when it is practical to do so because the scripts work on identical (or similar) event object properties. The results of this equalization are typically stored in the `evt` variable.

Properties

altKey
ctrlKey
metaKey
shiftKey

Value: Boolean Read-Only

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

When an event object is created in response to a user or system action, these four properties are set based on whether their corresponding keys were being held down at the time — a Shift-click, for example. If the key was held down, the property is assigned a value of `true`; otherwise the value is `false`. The `metaKey` property corresponds to the Command key on the Macintosh keyboard but does not register for the Windows key on Wintel computers.

Most commonly, you use expressions consisting of this property as `if` construction conditional statements. Because these are Boolean values, you can combine multiple properties in a single condition. For example, if you have a branch of a function that is to execute only if the event occurred with both the Shift and Control keys held down, the condition looks like the following:

```
if (evt.shiftKey && evt.ctrlKey)
{
   // statements to execute
}
```

Conversely, you can take a more user-friendly approach to provide special processing if the user holds down any one of the four modifier keys:

```
if (evt.shiftKey || evt.ctrlKey || evt.metaKey || evt.altKey)
{
   // statements to execute
}
```

The rationale behind this approach is to offer perhaps some shortcut operation for users, but not force them to memorize a specific modifier key combination.

Example

See Listing 32-6, where the values of these properties are used to set the `checked` properties of corresponding check boxes for a variety of event types.

Related Items: None

`bubbles`

Value: Boolean Read-Only

Compatibility: WinIE-, MacIE-, NN6+, Moz+, Safari+, Opera+, Chrome+

Not every event bubbles. For example, an `onsubmit` event propagates no further than the form object with which the event is associated. Events that do not bubble have their event object's `bubbles` property set to `false`; all others have the property set to `true`. You use this property in the rare circumstance of a single event handler function processing a wide variety of events. You may want to perform special operations only on events that can bubble, and handle the others without special treatment. For this branch, you can use the property in an `if` conditional statement:

```
if (evt.bubbles)
{
   // special processing for bubble-able events
}
```

You do not have to branch, however, just to cancel bubbling. A non-propagating event doesn't mind if you tell it not to propagate.

Related Item: `cancelBubble` property

`button`

Value: Integer Read-Only

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera8+, Chrome+

The `button` property reveals the button that was pressed to activate the mouse event. In the W3C DOM, the left (primary) button returns a value of `0`. If the mouse is a three-button mouse, the middle button returns `1`. The right button (on any multi-button mouse) returns a value of `2`.

Mouse buttons other than the primary one are easier to look for in `mousedown` or `mouseup` events, rather than `onclick` events. In the case of a user pressing multiple buttons, only the most recent button is registered.

Exercise caution when scripting the `button` property across browsers. The respective event models define different button values for mouse buttons.

Example

See Listing 32-7, where the `button` property is revealed in the status bar. Try pressing individual mouse buttons on, say, the screen button.

Related Items: None

`cancelable`

Value: Boolean Read-Only

Compatibility: WinIE-, MacIE-, NN6+, Moz+, Safari+, Opera+, Chrome+

If an event is cancelable, then its default action can be prevented from occurring with the help of a script. Although most events are cancelable, some are not. The cancelable property lets you inquire about a particular event object to see if its event type is `cancelable`. Values for the property are Booleans. You may want to perform special operations only on events that are cancelable and handle the others without special treatment. For this branch, you can use the property in an `if` conditional statement:

```
if (evt.cancelable)
{
   // special processing for cancelable events
}
```

You do not have to branch, however, just to prevent an event's default action. A non-cancelable event doesn't mind if you tell it to prevent the default action.

Related Item: `preventDefault()` method

cancelBubble

Value: Boolean Read/Write

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

The `cancelBubble` property is a rare instance of an IE4+ event property being implemented in NN6+/Moz and Gecko-, WebKit-, and Presto-based browsers, even though the property is not defined in the W3C DOM. The property operates the same as in IE4+ in that it determines whether the current event object bubbles up any higher in the element containment hierarchy of the document. By default, this property is `false`, meaning that if the event is supposed to bubble, it will do so automatically.

To prevent event bubbling for the current event, set the property to `true` anywhere within the event handler function. Cancelling event bubbling works only for the current event. The very next event to fire will have bubbling enabled (provided the event bubbles).

If you are trying to migrate your code as much as possible to the W3C DOM, use the `stopPropagation()` method instead of `cancelBubble`. For cross-browser compatibility, however, `cancelBubble` is a safe bet.

Example

See Listing 32-2 to see the `cancelBubble` property in action in an IE environment. Even though that listing has some features that apply to WinIE5.5+, the bubble cancelling demonstration works all the way back to IE4.

Related Item: `stopPropagation()` method

charCode
keyCode

Value: Integer Read-Only

Compatibility: WinIE-, MacIE-, NN6+, Moz+, Safari+, Opera-, Chrome+

eventObject.keyCode

The W3C DOM event object model clearly distinguishes between the Unicode character attached to the alphanumeric keys of the keyboard and the code attached to each of the keyboard keys (regardless of character). To inspect the character of a key, use the `onkeypress` event to create the `event` object, and then look at the `event` object's `charCode` property. This is the property that returns 97 for "a" and 65 for "A" because it's concerned with the character associated with the key action. This property's value is zero for `onkeydown` and `onkeyup` events.

In contrast, the `keyCode` property is filled with a non-zero value only from `onkeydown` and `onkeyup` events (`onkeypress` sets the property to zero) when alphanumeric keys are pressed; for most other non-character keys, all three events fill the `keyCode` property. Through this property you can look for non-character keys, such as arrows, page navigation, and function keys. For the character keys, there is no distinction between uppercase and lowercase: The "A" key on the Latin keyboard returns a value of 65, regardless of the state of the Shift key. At the same time, however, the press of the Shift key fires its own `onkeydown` and `onkeyup` events, setting the `keyCode` value to 16 (except in Safari, which does not register modifier keys in this way). Other non-character keys — arrows, page navigation, function, and similar — have their own codes as well. This gets very detailed, and involves special key codes for the numeric keyboard keys that are different from their corresponding numbers along the top row of the alphanumeric keyboard.

Cross-Reference

Be sure to see the extensive section on keyboard events in Chapter 26, "Generic HTML Element Objects," for examples of how to apply the `keyCode` property in applications. ■

Example

Listing 32-12 provides a play area to view the `charCode` and `keyCode` properties for all three keyboard events while you type into a `textarea`. You can use this later as an authoring tool to grab the precise codes for keyboard keys you may not be familiar with.

LISTING 32-12

Displaying charCode and keyCode Property Values

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>charCode and keyCode Properties</title>
      <style type="text/css">
         td {text-align:center}
      </style>
      <script type="text/javascript">
         function showCode(which, evt)
         {
            document.forms[0].elements[which + "Char"].value = evt.charCode;
            document.forms[0].elements[which + "Key"].value = evt.keyCode;
         }
         function clearEm()
         {
            for (var i = 1; i < document.forms[0].elements.length; i++)
```

```
            {
               document.forms[0].elements[i].value = "";
            }
         }

         // bind the event handlers
         function addEvent(elem, evtType, func)
         {
            if (elem.addEventListener)
            {
               elem.addEventListener(evtType, func, false);
            }
            else if (elem.attachEvent)
            {
               elem.attachEvent("on" + evtType, func);
            }
            else
            {
               elem["on" + evtType] = func;
            }
         }
         addEvent(window, "load", function()
         {
            addEvent(document.getElementById("scratchpad"), "keydown",
                     function(evt) {clearEm(); showCode("down", evt);});
            addEvent(document.getElementById("scratchpad"), "keypress",
                     function(evt) {showCode("press", evt);});
            addEvent(document.getElementById("scratchpad"), "keyup",
                     function(evt) {showCode("up", evt);});
         });
      </script>
   </head>
   <body>
      <h1>charCode and keyCode Properties</h1>
      <hr />
      <form>
         <p>
            <textarea id="scratchpad" name="scratchpad"
             cols="40" rows="5" wrap="hard"></textarea>
         </p>
         <table cellpadding="5">
            <tr>
               <th>Event</th>
               <th>event.charCode</th>
               <th>event.keyCode</th>
            </tr>
            <tr>
               <td>onKeyDown:</td>
               <td><input type="text" name="downChar" size="3" /></td>
               <td><input type="text" name="downKey" size="3" /></td>
            </tr>
            <tr>
               <td>onKeyPress:</td>
```

continued

LISTING 32-12 *(continued)*

```
            <td><input type="text" name="pressChar" size="3" /></td>
            <td><input type="text" name="pressKey" size="3" /></td>
         </tr>
         <tr>
            <td>onKeyUp:</td>
            <td><input type="text" name="upChar" size="3" /></td>
            <td><input type="text" name="upKey" size="3" /></td>
         </tr>
      </table>
   </form>
  </body>
</html>
```

Here are some specific tasks to try with the page in NN6+/Moz to examine key codes (if you are not using a browser set for English and a Latin-based keyboard, your results may vary):

1. Enter a lowercase **a**. Notice how the `onkeypress` event handler shows the `charCode` to be 97, which is the Unicode (and ASCII) value for the first of the lowercase letters of the Latin alphabet. But the other two event types record just the key's code: 65.
2. Type an uppercase **A** via the Shift key. If you watch closely, you see that the Shift key, itself, generates the key code 16 for the `onkeydown` and `onkeyup` events. But the character key then shows the value 65 for all three events (until you release the Shift key), because the ASCII value of the uppercase letter happens to match the keyboard key code for that letter.
3. Press and release the Down Arrow key (be sure the cursor still flashes in the `textarea`, because that's where the keyboard events are being monitored). As a non-character key, all three events stuff a value into the `keyCode` property, but zero into `charCode`. The `keyCode` value for this key is 40.
4. Poke around with other non-character keys. Some may produce dialog boxes or menus, but their key codes are recorded nonetheless.

Related Items: `onkeydown`, `onkeypress`, `onkeyup` event handlers

clientX
clientY
layerX
layerY
pageX
pageY
screenX
screenY

Value: Integer Read-Only

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

The W3C DOM `event` object borrows mouse coordinate properties from both the NN4 and the IE4+ event models. If you have worked with event coordinates in these other browsers, you have nothing new to learn for W3C DOM-compatible browsers.

Like the IE4+ `event` object, the W3C DOM `event` object's `clientX` and `clientY` properties are the coordinates within the viewable content region of the window. These values are relative to the window space, not the document. But unlike IE4+, you don't have to calculate the position of the coordinates within the document because another pair of NN/Moz properties, `pageX` and `pageY`, provide that information automatically. If the page has not scrolled, the values of the client and page coordinates are the same. Because it is usually more important to know an event's coordinates with respect to the document than the window, the `pageX` and `pageY` properties are used most often.

Another NN/Moz property pair, `layerX` and `layerY`, borrow terminology from the now defunct layer schemes of NN4, but the properties can still be quite valuable nonetheless. These coordinates are measured relative to the positioning context of the element that received the event. For regular, unpositioned elements in the `body` part of a document, that positioning context is the `body` element. Thus, for those elements, the values of the page and layer coordinates will be the same. But if you create a positioned element, the coordinate space is measured from the top-left corner of that space. Thus, if you are using the coordinates to assist in scripted dragging of positioned elements, you can confine your scope to just the positioned element.

One coordinate system missing from the NN6+/Moz repertoire, but present in Safari, is that of the target element itself (comparable to the `offsetX` and `offsetY` properties of IE4+). These values, however, can be calculated in NN/Moz by subtracting from the page coordinate properties the `offsetLeft` and `offsetTop` properties of both the target element and its positioning context. For example, if you want to get the coordinates of a mouse event inside an image, the event handler can calculate those values as follows:

```
var clickOffsetX = evt.pageX - evt.target.offsetLeft -
   document.body.offsetLeft;
var clickOffsetY = evt.pageY - evt.target.offsetTop -
   document.body.offsetTop;
```

The last set of coordinate properties, `screenX` and `screenY`, provide values relative to the entire video display. Of all these properties, only the client and screen coordinates are defined in the W3C DOM Level 2 standard.

Keep in mind that in most W3C DOM–compatible browsers, event targets include text nodes inside elements. Because nodes do not have all the properties of elements (for example, they have no offset properties signifying their location in the document), you may sometimes have to go to the target node's parent node to get an element object whose offset properties provide the necessary page geography. This matters, of course, only if your scripts need to concern themselves with mouse events on text.

Example

You can see the effects of the coordinate systems and associated NN6+/Moz properties with the page in Listing 32-13. You can view coordinate values for all four measuring systems, as well as some calculated value. Two clickable objects are provided so that you can see the differences between an object not in any layer and an object residing within a layer (although anything you see is clickable, including text nodes). Figure 32-1 shows the results of a click inside the positioned layer.

One of the calculated fields applies window scrolling values to the client coordinates. But, as you will see, these calculated values are the same as the more convenient page coordinates. The other calculated field shows the coordinates relative to the rectangular space of the target element. Notice in the

code that if the `nodeType` of the target indicates a text node, that node's parent node (an element) is used for the calculation.

LISTING 32-13

NN6+/Moz Event Coordinate Properties

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>X and Y Event Properties (NN6+/Moz)</title>
      <script type="text/javascript">
         function checkCoords(evt)
         {
            var form = document.forms["output"];
            var targText, targElem;
               if (evt.target.nodeType == 3)
               {
                  targText = "[textnode] inside <" +
                               evt.target.parentNode.tagName + ">";
                  targElem = evt.target.parentNode;
               }
               else
               {
                  targText = "<" + evt.target.tagName + ">";
                  targElem = evt.target;
               }
            form.srcElemTag.value = targText;
            form.clientCoords.value = evt.clientX + "," + evt.clientY;
            form.clientScrollCoords.value = (evt.clientX + window.scrollX) +
                                            "," + (evt.clientY + window.scrollY);
            form.layerCoords.value = evt.layerX + "," + evt.layerY;
            form.pageCoords.value = evt.pageX + "," + evt.pageY;
            form.inElemCoords.value =
               (evt.pageX - targElem.offsetLeft - document.body.offsetLeft) +
               "," + (evt.pageY - targElem.offsetTop - document.body.offsetTop);
            form.screenCoords.value = evt.screenX + "," + evt.screenY;
            return false;
         }

         // bind the event handler
         function addEvent(elem, evtType, func)
         {
            if (elem.addEventListener)
            {
               elem.addEventListener(evtType, func, false);
            }
            else if (elem.attachEvent)
            {
               elem.attachEvent("on" + evtType, func);
            }
```

```
            else
            {
               elem["on" + evtType] = func;
            }
         }
         addEvent(window, "load", function()
         {
            addEvent(document.body, "mousedown",
                     function(evt) {checkCoords(evt);});
         });
      </script>
   </head>
   <body>
      <h1>X and Y Event Properties (NN6+/Moz)</h1>
      <hr />
      <p>Click on the button and in the DIV/image to see the coordinate values
         for the event object.
      </p>
      <form name="output">
         <table>
            <tr>
               <td colspan="2">NN6 Mouse Event Coordinates:</td>
            </tr>
            <tr>
               <td align="right">target:</td>
               <td colspan="3"><input type="text" name="srcElemTag"
               <td size="25" /></td>
            </tr>
            <tr>
               <td align="right">clientX, clientY:</td>
               <td><input type="text" name="clientCoords" size="10" /></td>
               <td align="right">...With scrolling:</td>
               <td><input type="text" name="clientScrollCoords" size="10" /></td>
            </tr>
            <tr>
               <td align="right">layerX, layerY:</td>
               <td><input type="text" name="layerCoords" size="10" /></td>
            </tr>
            <tr>
               <td align="right">pageX, pageY:</td>
               <td><input type="text" name="pageCoords" size="10" /></td>
               <td aligh="right">Within Element:</td>
               <td><input type="text" name="inElemCoords" size="10" /></td>
            </tr>
            <tr>
               <td align="right">screenX, screenY:</td>
               <td><input type="text" name="screenCoords" size="10" /></td>
            </tr>
            <tr>
               <td align="right"><input type="button" value="Click Here" /></td>
            </tr>
         </table>
      </form>
```

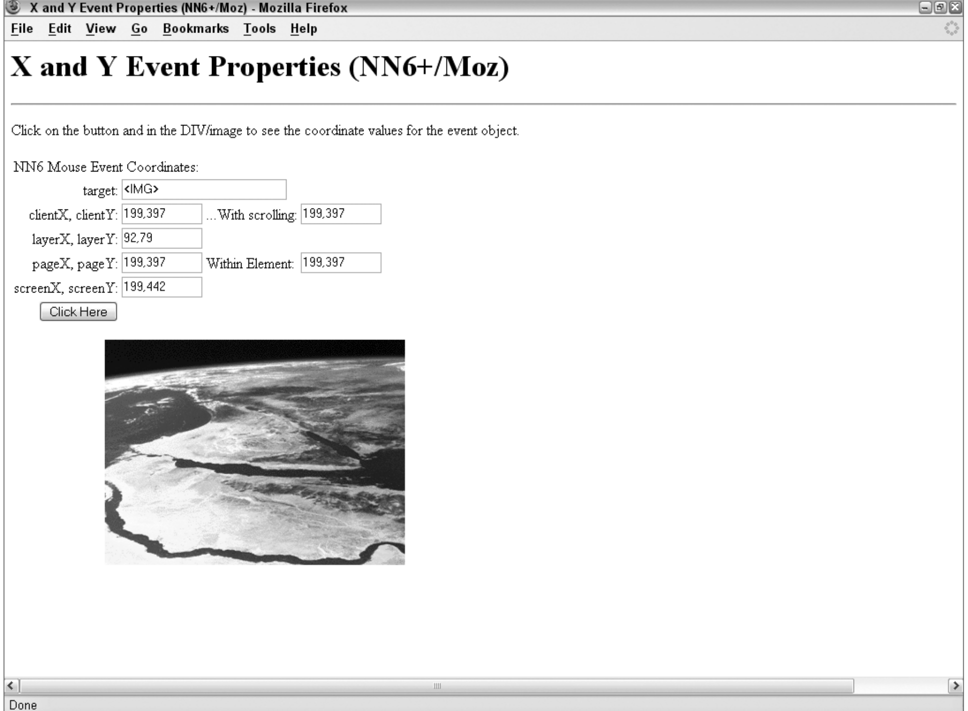
continued

LISTING 32-13 *(continued)*

```
      <div id="display" style="position:relative; left:100">
         <img alt="image" src="nile.gif" width="320" height="240" border="0" />
      </div>
   </body>
</html>
```

FIGURE 32-1

NN6+/Moz event coordinates for a click inside a positioned element.



Related Item: `target` property

`currentTarget`

Value: Element object reference — Read-Only

Compatibility: WinIE-, MacIE-, NN6+, Moz+, Safari+, Opera+, Chrome+

As an event courses its way through its propagation paths, an event listener may process that event along the way. Though the event knows what the target is, it can also be helpful for the event listener function to know which element's event listener is now processing the event. The `currentTarget` property provides a reference to the element object whose event listener is processing the event. This allows one listener function to potentially process the event from different levels, branching the code to accommodate different element levels that process the event.

A valuable companion piece of information about the event is the `eventPhase` property, which helps your event listener function determine if the event is in capture mode, bubble mode, or is at the target. This property is demonstrated in the next section.

Example

Listing 32-14 shows the power of the `currentTarget` property in revealing the element that is processing an event during event propagation. Similar to the code in Listing 32-3, this example is made simpler because it lets the event object's properties do more of the work to reveal the identity of each element that processes the event. Event listeners assigned for various propagation modes are assigned to a variety of nodes in the document. After you click the button, each listener in the propagation chain fires in sequence. The alert dialog box shows which node is processing the event. And, as in Listing 32-3, the `eventPhase` property is used to help display the propagation mode in force at the time the event is processed by each node.

LISTING 32-14

currentTarget and eventPhase Properties

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>currentTarget and eventPhase Properties</title>
      <script type="text/javascript">
         function processEvent(evt)
         {
            var currTargTag, msg;
               if (evt.currentTarget.nodeType == 1)
               {
                  currTargTag = "<" + evt.currentTarget.tagName + ">";
               }
               else
               {
                  currTargTag = evt.currentTarget.nodeName;
               }
            msg = "Event is now at the " + currTargTag + " level ";
            msg += "(" + getPhase(evt) + ").";
            alert(msg);
         }
         // reveal event phase of current event object
         function getPhase(evt)
         {
```

continued

***eventObject*.currentTarget**

LISTING 32-14 *(continued)*

```
            switch (evt.eventPhase)
            {
               case 1:
                  return "CAPTURING";
                  break;
               case 2:
                  return "AT TARGET";
                  break;
               case 3:
                  return "BUBBLING";
                  break;
               default:
                  return "";
            }
         }

         // bind the event handlers
         function addEvent(elem, evtType, func)
         {
            if (elem.addEventListener)
            {
               elem.addEventListener(evtType, func, false);
            }
            else if (elem.attachEvent)
            {
               elem.attachEvent("on" + evtType, func);
            }
            else
            {
               elem["on" + evtType] = func;
            }
         }
         addEvent(window, "load", function()
         {
            // using old syntax to assign bubble-type event handlers
            document.onclick = processEvent;
            document.body.onclick = processEvent;
            // turn on click event capture for document and form
            document.addEventListener("click", processEvent, true);
            document.forms[0].addEventListener("click", processEvent, true);
            // set bubble event listener for form
            document.forms[0].addEventListener("click", processEvent, false);
            // turn on event capture for the button
            document.getElementById("main1").addEventListener("click",
            processEvent, true);
         });
      </script>
   </head>
   <body>
```

```
      <h1>currentTarget and eventPhase Properties</h1>
      <hr />
      <form>
         <input type="button" value="A Button" id="main1" name="main1" />
      </form>
   </body>
</html>
```

You can also click other places on the page. For example, if you click to the right of the button, you will be clicking the `form` element. Event propagation and processing adjusts accordingly. Similarly, if you click the header text, the only event listeners that see the event are in the document and `body` levels.

Related Item: `eventPhase` property

detail

Value: Integer Read-Only

Compatibility: WinIE-, MacIE-, NN6+, Moz+, Safari2+, Opera8+, Chrome+

The `detail` property is included in the W3C DOM specification as an extra property whose purpose can be determined by the browser maker. Mozilla-, WebKit-, and Presto-based browsers increment a numeric value for rapid instances of `click` events on an object.

Related Items: None

eventPhase

Value: Integer Read-Only

Compatibility: WinIE-, MacIE-, NN6+, Moz+, Safari+, Opera+, Chrome+

An event fires in one of three possible event phases: at event capture, at the target, or during bubbling. Because the same event listener function may be processing an event in multiple phases, it can inspect the value of the `eventPhase` property of the event object to see in which phase the event was when the function was invoked. Values for this property are integers 1 (capture), 2 (at target), or 3 (bubbling).

Example

Refer to Listing 32-14, earlier in this chapter, for an example of how you can use a `switch` construction to branch function processing based on the event phase of the current event object.

Related Item: `currentTarget` property

isChar

Value: Boolean Read-Only

Compatibility: WinIE-, MacIE-, NN6+, Moz+, Safari-, Opera-, Chrome-

You can find out from each keyboard event whether the key being pressed is a character key by examining the `isChar` property. Most typically, however, you are already filtering for character

or non-character keys by virtue of the event handlers used to capture keyboard actions: `onkeypress` for character keys; `onkeydown` or `onkeyup` for non-character keys. Be aware that the `isChar` property returns inconsistent values (even for the same key) in the first release of NN6.

Related Items: `charCode`, `keyCode` properties

isTrusted

Value: Boolean Read-Only

Compatibility: WinIE-, MacIE-, NN8+, Moz1.7.5+, Safari-, Opera-, Chrome-

Because an event can be generated by user action or script, the `isTrusted` property enables you to determine how the event was created. Any event triggered by user action is considered to be trusted from a security point of view. Therefore, when this property returns `true`, it means that the event came to life as the result of user activity (clicking, keyboarding, and so on).

originalTarget

Value: Node object reference Read-Only

Compatibility: WinIE-, MacIE-, NN6+, Moz+, Safari-, Opera-, Chrome-

The `originalTarget` property provides a reference to the node object that serves as the genuine first target of the event. This information is typically associated with the internal construction of certain elements, which makes it less useful for scripting purposes. Additionally, in many cases the `originalTarget` property holds the same value as the `target` property.

relatedTarget

Value: Element object Read-Only

Compatibility: WinIE-, MacIE-, NN6+, Moz+, Safari+, Opera+, Chrome+

The `relatedTarget` property allows an element to uncover where the cursor rolled in from or has rolled out to. This property extends the power of the `onmouseover` and `onmouseout` event handlers by expanding their scope to outside the current element (usually to an adjacent element). This one W3C DOM property does the same duty as the `fromElement` and `toElement` properties of the IE4+ `event` object.

When the `onmouseover` event fires on an element, the cursor had to be over some other element just beforehand. The `relatedTarget` property holds a reference to that element. Conversely, when the `onmouseout` event fires, the cursor is already over some other element. The `relatedTarget` property holds a reference to that element.

Example

Listing 32-15 provides an example of how the `relatedTarget` property can reveal the life of the cursor action before and after it rolls into an element. When you roll the cursor to the center box (a table cell), its `onmouseover` event handler displays the text from the table cell from which the cursor arrived (the `nodeValue` of the text node inside the table cell). If the cursor comes in from one of the corners (not easy to do), a different message is displayed.

The two functions that report the results employ a bit of filtering to make sure that they process the event object only if the event occurs on an element and if the `relatedTarget` element is anything

other than a nested text node of the central table cell element. Because nodes respond to events in W3C DOM browsers, this extra filtering prevents processing whenever the cursor makes the transition from the central `td` element to its nested text node.

LISTING 32-15

Using the relatedTarget Property

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>relatedTarget Properties</title>
      <style type="text/css">
         .direction {background-color:#00FFFF; width:100; height:50;
                              text-align:center;}
         #main {background-color:#FF6666; text-align:center;}
      </style>
      <script type="text/javascript">
         function showArrival(evt)
         {
            if (evt.target.nodeType == 1)
            {
               if (evt.relatedTarget != evt.target.firstChild)
               {
                  var direction = (evt.relatedTarget.firstChild) ?
                     evt.relatedTarget.firstChild.nodeValue : "parts unknown";
                  window.status = "Arrived from: " + direction;
               }
            }
         }
         function showDeparture(evt)
         {
            if (evt.target.nodeType == 1)
            {
               if (evt.relatedTarget != evt.target.firstChild)
               {
                  var direction = (evt.relatedTarget.firstChild) ?
                     evt.relatedTarget.firstChild.nodeValue : "parts unknown";
                  window.status = "Departed to: " + direction;
               }
            }
         }

         // bind the event handlers
         function addEvent(elem, evtType, func)
         {
            if (elem.addEventListener)
            {
               elem.addEventListener(evtType, func, false);
            }
            else if (elem.attachEvent)
```

continued

LISTING 32-15 *(continued)*

```
          {
             elem.attachEvent("on" + evtType, func);
          }
          else
          {
             elem["on" + evtType] = func;
          }
       }
       addEvent(window, "load", function()
       {
          addEvent(document.getElementById("main"), "mouseover",
                   function(evt) {showArrival(evt);});
          addEvent(document.getElementById("main"), "mouseout",
                   function(evt) {showDeparture(evt);});
       });
    </script>
  </head>
  <body>
    <h1>relatedTarget Properties</h1>
    <hr />
    <p>Roll the mouse to the center box and look for arrival information in
       the status bar. Roll the mouse away from the center box and look for
       departure information in the status bar.
    </p>
    <table cellspacing="0" cellpadding="5">
       <tr>
          <td></td>
          <td class="direction">North</td>
          <td></td>
       </tr>
       <tr>
          <td class="direction">West</td>
          <td id="main">Roll</td>
          <td class="direction">East</td>
       </tr>
       <tr>
          <td></td>
          <td class="direction">South</td>
          <td></td>
       </tr>
    </table>
  </body>
</html>
```

Related Item: `target` property

`target`

Value: Element object reference Read-Only

Compatibility: WinIE-, MacIE-, NN6+, Moz+, Safari+, Opera+, Chrome+

The `target` property is a reference to the HTML element object that is the original target of the event. Because an event may trickle down and bubble up through the element containment hierarchy and be processed at any level along the way, having a property that points back to the element from which the event originated is comforting. As soon as you have a reference to that element, you can read or write any properties that belong to that element, or invoke any of its methods.

Example

As a simplified demonstration of the power of the `target` property, Listing 32-16 has but two event handlers defined for the `body` element, each invoking a single function. The idea is that the `onmousedown` and `onmouseup` events will bubble up from whatever their targets are, and the event handler functions will find out which element is the target and modify the color style of that element.

An extra flair is added to the script in that each function also checks the `className` property of the target element. If the `className` is `bold` — a class name shared by three `span` elements in the paragraph — the stylesheet rule for that class is modified so that all items share the same color. Your scripts can do even more in terms of filtering objects that arrive at the functions by performing special operations on certain objects or groups of objects.

Notice that the scripts don't have to know anything about the objects on the page to address each clicked one individually. That's because the `target` property provides all of the specificity needed for acting on the target element.

LISTING 32-16

Using the target Property

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>target Property</title>
      <style type="text/css">
         .bold {font-weight:bold;}
         .ital {font-style:italic;}
      </style>
      <script type="text/javascript">
         function highlight(evt)
         {
            var elem = (evt.target.nodeType == 3) ?
                       evt.target.parentNode : evt.target;
               if (elem.className == "bold")
               {
                  document.styleSheets[0].cssRules[0].style.color = "red";
               }
               else
               {
                  elem.style.color = "#FFCC00";
               }
```

continued

LISTING 32-16 *(continued)*

```
      }
      function restore(evt)
      {
         var elem = (evt.target.nodeType == 3) ?
                    evt.target.parentNode : evt.target;
            if (elem.className == "bold")
            {
               document.styleSheets[0].cssRules[0].style.color = "black";
            }
            else
            {
               elem.style.color = "black";
            }
      }

      // bind the event handlers
      function addEvent(elem, evtType, func)
      {
         if (elem.addEventListener)
         {
            elem.addEventListener(evtType, func, false);
         }
         else if (elem.attachEvent)
         {
            elem.attachEvent("on" + evtType, func);
         }
         else
         {
            elem["on" + evtType] = func;
         }
      }
      addEvent(window, "load", function()
      {
         addEvent(document.body, "mousedown",
                  function(evt) {highlight(evt);});
         addEvent(document.body, "mouseup",
                  function(evt) {restore(evt);});
      });
   </script>
</head>
<body>
   <h1>target Property</h1>
   <hr />
   <p>One event handler...</p>
   <ul>
      <li>Can</li>
      <li>Cover</li>
      <li>Many</li>
      <li>Objects</li>
   </ul>
```

```
      <p>Lorem ipsum dolor sit amet, consectetaur adipisicing elit,
         <span class="bold">sed do</span>
         eiusmod tempor incididunt
         <span class="ital">ut labore et</span>
         dolore magna aliqua. Ut enim adminim veniam,
         <span class="bold">quis nostrud exercitation</span>
         ullamco laboris nisi ut aliquip ex ea
         <span class="bold">commodo consequat</span>.
      </p>
   </body>
</html>
```

Related Item: `relatedTarget` property

timeStamp

Value: Integer — Read-Only

Compatibility: WinIE-, MacIE-, NN6+, Moz+, Safari+, Opera-, Chrome+

Each event receives a time stamp in milliseconds, based on the same date epoch as the `Date` object (1 January 1970). Just as with the `Date` object, accuracy is wholly dependent on the accuracy of the system clock of the client computer.

Although the precise time of an event may be of value in only some situations, the time between events can be useful for applications, such as timed exercises or action games. You can preserve the time of the most recent event in a global variable, and compare the time of the current time stamp against the stored value to determine the elapsed time between events.

Example

Listing 32-17 uses the `timeStamp` property to calculate the instantaneous typing speed when you type into a `textarea`. The calculations are pretty raw, and work only on intra-keystroke times without any of the averaging or smoothing that a more sophisticated typing tutor might perform. Calculated values are rounded to the nearest integer.

LISTING 32-17

Using the timeStamp Property

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>timeStamp Property</title>
      <script type="text/javascript">
         var stamp;
         function calcSpeed(evt)
         {
```

continued

eventObject.type

LISTING 32-17 *(continued)*

```
            if (stamp)
            {
               var gross = evt.timeStamp - stamp;
               var wpm = Math.round(6000/gross);
               document.getElementById("wpm").firstChild.nodeValue = wpm + " wpm.";
            }
            stamp = evt.timeStamp;
         }

         // bind the event handlers
         function addEvent(elem, evtType, func)
         {
            if (elem.addEventListener)
            {
               elem.addEventListener(evtType, func, false);
            }
            else if (elem.attachEvent)
            {
               elem.attachEvent("on" + evtType, func);
            }
            else
            {
               elem["on" + evtType] = func;
            }
         }
         addEvent(window, "load", function()
         {
            addEvent(document.getElementById("scratchpad"), "keypress",
                     function(evt) {calcSpeed(evt);});
         });
      </script>
   </head>
   <body>
      <h1>timeStamp Property</h1>
      <hr />
      <p>Start typing, and watch your instantaneous typing speed below:</p>
      <p><textarea id="scratchpad" cols="60" rows="10" wrap="hard"></textarea></p>
      <p>Typing Speed: <span id="wpm"> </span></p>
   </body>
</html>
```

Related Item: `Date` object

`type`

Value: String — Read-Only

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

You can find out what kind of event fired to create the current event object by way of the `type` property. The value is a string version of the event name — just the name of the event without the "on" prefix that is normally associated with event listener names in NN6+/Moz/W3C. This property can be helpful when you designate one event handler function to process different kinds of events. For example, both the `onmousedown` and `onclick` event listeners for an object can invoke one function. Inside the function, a branch is written for whether the type comes in as `mousedown` or `click`, with different processing for each event type. That is not to endorse such event handler function sharing, but to be aware of this power should your script constructions find the property helpful.

This property and its values are fully compatible with the NN4 and IE4+ event models.

Caution

Keyboard events in earlier versions of Safari reported their types as `khtml_keydown`, `khtml_keypress`, and `khtml_keyup`. This was probably to avoid committing to an unfinished W3C DOM Level 3 keyboard event specification. The current version of Safari reports the keyboard events without the `khtml` prefix. ■

Related Items: All event handlers (Chapter 26, "Generic HTML Element Objects")

view

Value: Window object reference Read-Only

Compatibility: WinIE-, MacIE-, NN6+, Moz+, Safari+, Opera+, Chrome+

The closest that the W3C DOM Level 2 specification comes to acknowledging the browser window is an abstract object called an *abstract view* (AbstractView class). The object's only property is a reference to the document that it contains — the root document node that you've come to know and love. User events always occur within the confines of one of these views, and this is reflected in the `event` object's `view` property. This property holds a reference to the `window` object (which can be a frame) in which the event occurs. This reference allows an event object to be passed to scripts in other frames, and those scripts can then gain access to the `document` object of the target element's window.

Related Item: `window` object

Methods

initEvent("*eventType*", *bubblesFlag*, *cancelableFlag*)

initKeyEvent("*eventType*", *bubblesFlag*, *cancelableFlag*, *view*, *ctrlKeyFlag*, *altKeyFlag*, *shiftKeyFlag*, *metaKeyFlag*, *keyCode*, *charCode*)

initMouseEvent("*eventType*", *bubblesFlag*, *cancelableFlag*, *view*, *detailVal*, *screenX*, *screenY*, *clientX*, *clientY*, *ctrlKeyFlag*, *altKeyFlag*, *shiftKeyFlag*, *metaKeyFlag*, *buttonCode*, *relatedTargetNodeRef*)

initMutationEvent("*eventType*", *bubblesFlag*, *cancelableFlag*, *relatedNodeRef*, *prevValue*, *newValue*, *attrName*, *attrChangeCode*)

initUIEvent("*eventType*", *bubblesFlag*, *cancelableFlag*, *view*, *detailVal*)

Returns: Nothing

Compatibility: WinIE-, MacIE-, NN6+, Moz+, Safari+, Opera+, Chrome+

The W3C DOM `event` object initialization methods provide a means of initializing a newly-created event with a complete set of property values associated with that particular event. The parameters to each of the initialization methods vary according to the type of event being initialized. However, all of the initialization methods share the first three parameters: `eventType`, `bubblesFlag`, and `cancelableFlag`. The `eventType` parameter is a string identifier for the event's type, such as `mousedown` or `keypress`. The `bubblesFlag` parameter is a Boolean value that specifies whether the event's default propagation behavior is to bubble (`true`) or not (`false`). The `cancelableFlag` parameter is also a Boolean value, and its job is to specify if the event's default action may be prevented with a call to the `preventDefault()` method (`true`) or not (`false`).

A few of the methods also include `view` and `detailVal` parameters, which correspond to the window or frame in which the event occurred, and the integer code of detail data associated with the event, respectively. Additional parameters are specified for some of the methods, and are unique to the event being initialized.

You don't have to use the more detailed methods if you need a simple event. For example, if you want a simple `mouseup` event, you can initialize a generic event with `initEvent()`, and dispatch the event to the desired element, without having to fill in all of the coordinate, button, and other parameters of the `initMouseEvent()` method:

```
var evt = document.createEvent("MouseEvents");
evt.initEvent("mouseup", true, true);
document.getElementById("myButton").dispatchEvent(evt);
```

For more details about W3C DOM event types and the values expected for each of the more complex initialization methods, visit `http://www.w3.org/TR/DOM-Level-2-Events/events.html#Events-eventgroupings`.

Related Item: `document.createEvent()` method

preventDefault()

Returns: Nothing

Compatibility: WinIE-, MacIE-, NN6+, Moz+, Safari+, Opera+, Chrome+

While NN6+ continues to honor the original way of preventing default action for an event handler (that is, having the last statement of the event handler evaluate to `return false`), the W3C DOM event model provides a method that lets the cancellation of default action take place entirely within a function invoked by an event handler. For example, consider a text box that is supposed to allow only numbers to be typed in it. The `onkeypress` event handler can invoke a function that inspects each typed character. If the character is not a numeric character, it does not reach the text box for display. The following validation function may be invoked from the `onkeypress` event handler of just such a text box:

```
function checkIt(evt)
{
   var charCode = evt.charCode;
      if (charCode < 48 || charCode > 57)
      {
```

```
            alert("Please make sure entries are numbers only.");
            evt.preventDefault();
        }
}
```

This way, the errant character won't appear in the text box.

Invoking the `preventDefault()` method in NN6+/Moz and Gecko-, WebKit-, and Presto-based browsers, is the equivalent of assigning `true` to `event.returnValue` in IE5+.

Related Item: `cancelable` property

stopPropagation()

Returns: Nothing

Compatibility: WinIE-, MacIE-, NN6+, Moz+, Safari+, Opera+, Chrome+

Use the `stopPropagation()` method to stop events from trickling down or bubbling up further through the element containment hierarchy. A statement in the event listener function that invokes the following is all that is needed:

```
evt.stopPropagation();
```

As an alternative, you can cancel bubbling directly in an element's event handler attribute, as in the following:

```
onclick="doButtonClick(this); event.stopPropagation()"
```

If you are writing cross-browser scripts, you also have the option of using the `cancelBubble` property, which is compatible with IE4+.

Related Items: `bubbles`, `cancelBubble` properties

Part VI

Document Objects Reference (continued)

IN THIS PART

Chapter 33
Body Text Objects

Chapter 34
The Form and Related Objects

Chapter 35
Button Objects

Chapter 36
Text-Related Form Objects

Chapter 37
Select, Option, and FileUpload Objects

Chapter 38
Style Sheet and Style Objects

Chapter 39
Ajax, E4X, and XML

Chapter 40
HTML Directive Objects

Chapter 41
Table and List Objects

Chapter 42
The Navigator and Other Environment Objects

Chapter 43
Positioned Objects

Chapter 44
Embedded Objects

Chapter 45
The Regular Expression and RegExp Objects

CHAPTER 33

Body Text Objects

A large number of HTML elements fall into a catchall category of elements whose purposes are slightly more targeted than contextual elements covered in Chapter 26, “Generic HTML Element Objects.” In this group are some very widely used elements, such as the `h1` through `h6` header elements, plus several elements that are not yet widely used because their full support may be lacking, even in some of the most modern browsers. In this chapter, you find all sorts of text-related objects, excluding those objects that act as form controls (text boxes and such, which are covered in Chapter 35, “Text-Related Form Objects”). For the most part, properties, methods, and event handlers of this chapter’s objects are the generic ones covered in Chapter 26. Only those items that are unique to each object are covered in this chapter (as will be the case in all succeeding chapters).

IN THIS CHAPTER

Objects that display running body text in documents

Using the NN/Mozilla Range and IE TextRange objects

Scripting search-and-replace actions

Beyond the HTML element objects covered in this chapter, you also meet the `TextRange` object, first introduced in IE4, and the corresponding `Range` object from the W3C DOM. This object is a very powerful one for scripters because it allows scripts to work very closely with body content — not in terms of, for example, the `innerText` or `nodeValue` properties of elements, but rather in terms of the text as it appears on the page in what users see as paragraphs, lists, and the like. The `TextRange` and `Range` objects essentially give your scripts cursor control over running body text for functions, such as cutting, copying, pasting, and applications that extend those basic operations — search and replace, for instance. Bear in mind that everything you read in this chapter requires, at minimum, the dynamic object models of IE4+ and NN6+/Moz/W3C; some items require IE5+. Unfortunately, the IE `TextRange` object is not implemented in MacIE5. In short, consider this chapter to be focused exclusively on modern browsers.

blockquote and q Element Objects

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

For HTML element properties, methods, and event handlers, see Chapter 26, "Generic HTML Element Objects."

Properties	Methods	Event Handlers
`cite`		

Syntax

Accessing `blockquote` or `q` element object properties or methods:

```
(IE4+)     [window.]document.all.elemID.property | method([parameters])
(IE5+/W3C) [window.]document.getElementById("elemID").property |
           method([parameters])
```

About these objects

The `blockquote` element is a special-purpose text container. Browsers typically start the content on a new line in the body, and indent on both the left and right margins by approximately 40 pixels. An inline quotation can be encased inside a `q` element, which does not force the quoted material to start on the next line.

From an object point of view, the only property that distinguishes these two objects from any other kind of contextual container is the `cite` property, which comes from the HTML 4.0 `cite` attribute. This attribute simply provides a URL reference for the citation and does not act as an `src` or `href` attribute to load an external document.

Property

`cite`

Value: String — Read/Write

Compatibility: WinIE6+, MacIE5+, NN6+, Moz+, Safari+, Opera+, Chrome+

The `cite` property can contain a URL (as a string) that points to the source of the quotation in the `blockquote` or `q` element. Future browsers may provide some automatic user interface link to the source document, but none of the current browsers that support the `cite` property do anything special with this information.

br Element Object

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

For HTML element properties, methods, and event handlers, see Chapter 26.

Properties	Methods	Event Handlers
`clear`		

Syntax

Accessing `br` element object properties or methods:

```
(IE4+)      [window.]document.all.elemID.property | method([parameters])
(IE5+/W3C) [window.]document.getElementById("elemID").property |
            method([parameters])
```

About this object

The `br` element forces a carriage return and line feed for rendered content on the page. This element does not provide the same kind of vertical spacing that goes between paragraphs in a series of `p` elements. Only one attribute (`clear`) distinguishes this element from generic HTML elements and objects.

Property

`clear`

Value: String Read/Write

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

The `clear` property defines how any text in an element following the `br` element wraps around a floating element (for example, an image set to float along the right margin). Although modern browsers expose this property, the attribute on which it is based is deprecated in the HTML 4.0 specification in an effort to encourage the use of the `clear` style sheet attribute for a `br` element.

Values for the `clear` property can be one of the following strings: `all`, `left`, or `right`.

Related Items: `clear` style sheet property

font Element Object

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+ Opera+, Chrome+

For HTML element properties, methods, and event handlers, see Chapter 26.

Properties	Methods	Event Handlers
`color`		
`face`		
`size`		

Syntax

Accessing `font` element object properties or methods:

```
(IE4+)      [window.]document.all.elemID.property | method([parameters])
(IE5+/W3C) [window.]document.getElementById("elemID").property |
            method([parameters])
```

About this object

In a juxtaposition of standards implementations, the `font` element is exposed as an object only in browsers that also support Cascading Style Sheets as the preferred way to control font faces, colors, and sizes. This doesn't mean that you shouldn't use `font` elements in your page with modern browsers — using this element may be necessary for a single page that needs to be backward compatible with older browsers. But it does present a quandary for scripters who want to use scripts to modify font characteristics of body text after the page has loaded. Keep in mind that the HTML element has been deprecated. A good rule of thumb to follow these days is to use the standards-compliant style sheets (and their scriptable properties). Use the `font` element (and script the `font`-HTML element object's properties) in those rare cases when the page must work in legacy browsers.

Properties

`color`

Value: String Read/Write

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

A `font` object's text color can be controlled via the `color` property. Values can be either hexadecimal triplets (for example, `#FFCCFF`) or the plain-language color names recognized by most browsers. In either case, the values are case-insensitive strings.

***fontObject*.color**

Example

Listing 33-1 contains a page that demonstrates changes to the three font element object properties: color, face, and size. Along the way, you can see an economical use of the setAttribute() method to do the work for all of the property changes. This page loads successfully in all browsers, but the select lists make changes to the text only in modern browsers.

A p element contains a nested font element that encompasses three words whose appearance is controlled by three select lists. Each list controls one of the three font object properties, and their name attributes are strategically assigned the names of the properties (as you see in a moment). value attributes for option elements contain strings that are to be assigned to the various properties. Each select element invokes the same setFontAttr() function, passing a reference to itself so that the function can inspect details of the element.

The first task of the setFontAttr() function is to make sure that only browsers capable of treating the font element as an object get to the meat of the function. The test for the existence of document.all and the myFONT element blocks all older browsers from changing the font characteristics.

For suitably equipped browsers, the function next extracts the string from the value property of the select object that was passed to the function. If a selection is made (other than the first, empty one), the single nested statement uses the setAttribute() method to assign the value to the attribute whose name matches the name of the select element.

Note

An odd bug in MacIE5 doesn't let the rendered color change when changing the color property. But the setting is valid, as proven by selecting any of the other two property choices. ■

LISTING 33-1

Dynamically Changing Font Properties

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Font Object Properties</title>
      <script type="text/javascript">
         // one function does all!
         function setFontAttr(select)
         {
            var choice = select.options[select.selectedIndex].value;
            if (choice)
```

```
            {
               document.getElementById("myFONT").setAttribute(select.name, choice);
            }
         }
      </script>
   </head>
   <body>
      <h1>Font Object Properties</h1>
      <br />
      <p>This may look like a simple sentence, but <font id="myFONT">THESE
         THREE WORDS</font> are contained by a FONT element.</p>
      <form>
         Select a text color:
         <select name="color" onchange="setFontAttr(this)">
            <option></option>
            <option value="red">Red</option>
            <option value="green">Green</option>
            <option value="blue">Blue</option>
            <option value="#FA8072">Some Hex Triplet Value</option>
         </select><br />
         Select a font face:
         <select name="face" onchange="setFontAttr(this)">
            <option></option>
            <option value="Helvetica">Helvetica</option>
            <option value="Times">Times</option>
            <option value="Comic Sans MS, sans-serif">Comic Sans MS,
            sans-serif</option>
            <option value="Courier, monospace">Courier, monospace</option>
            <option value="Zapf Dingbats, serif">Zapf Dingbats, serif</option>
         </select><br />
         Select a font size:
         <select name="size" onchange="setFontAttr(this)">
            <option></option>
            <option value="3">3 (Default)</option>
            <option value="+1">Increase Default by 1</option>
            <option value="-1">Decrease Default by 1</option>
            <option value="1">Smallest</option>
            <option value="7">Biggest</option>
         </select>
      </form>
   </body>
</html>
```

Note

The property assignment event handling technique used in this example and throughout the chapter is a deliberate simplification to make the code more readable. It is generally better to use the more modern approach of binding events using the `addEventListener()` **(NN6+/Moz/W3C) or** `attachEvent()` **(IE5+) methods. A modern cross-browser event handling technique is explained in detail in Chapter 32, "Event Objects."** ■

Related Items: `color` style sheet attribute

`face`

Value: String Read/Write

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

A `font` object's font face is controllable via the `face` property. Just as with the `face` attribute (and the corresponding `font-family` style sheet attribute), you can specify one or more font names in a comma-delimited string. Browsers start with the leftmost font face and look for a match in the client computer's system. The first matching font face that is found in the client system is applied to the text surrounded by the `font` element. You should list the most specific fonts first, and generally allow the generic font faces (`sans-serif`, `serif`, and `monospace`) to come last; that way you exert at least some control over the look of the font on systems that don't have your pretty fonts. If you know that Windows displays a certain font you like and the Macintosh has something that corresponds to that font but with a different name, you can specify both names in the same property value. The browser skips over font face names not currently installed on the client.

Example

See Listing 33-1 for an example of values that can be used to set the `face` property of a `font` element object. Although you will notice visible changes to most choices on the page, the font face selections may not change from one choice to another, since that all depends on the fonts that are installed on your PC.

Related Items: `font-family` style sheet attribute

`size`

Value: String Read/Write

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

The size of text contained by a `font` element can be controlled via the `size` property. Unlike the more highly recommended `font-size` style sheet attribute, the `size` property of the `font` element object and its corresponding `SIZE` attribute are restricted to the relative font size scale imposed by early HTML implementations: a numbering scale from 1 to 7.

Values for the `size` property are strings, even though most of the time they are single numeral values. You can also specify a size relative to the default value by including a plus or minus sign before the number. For example, if the default font size (as set by the browser's user preferences) is 3, you can bump up the size of a text segment by encasing it inside a `font` element and then setting its `size` property to `"+2"`.

For more accurate font sizing using units, such as pixels or points, use the `font-size` style sheet attribute.

Example

See Listing 33-1 for an example of values that can be used to set the `size` property of a `font` element object. Notice that incrementing or decrementing the `size` property is applied only to the size assigned to the `size` attribute of the element (or the default, if none is specified) and not the current setting adjusted by script.

Related Items: `font-size` style sheet attribute

h1 . . . h6 Element Objects

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

For HTML element properties, methods, and event handlers, see Chapter 26.

Properties	Methods	Event Handlers
`align`		

Syntax

Accessing `h1` through `h6` element object properties or methods:

```
(IE4+)     [window.]document.all.elemID.property | method([parameters])
(IE5+/W3C) [window.]document.getElementById("elemID").property |
           method([parameters])
```

About these objects

The so-called "heading" elements (denoted by `h1`, `h2`, `h3`, `h4`, `h5`, and `h6`) provide shortcuts for formatting up to six different levels of headings and subheadings. Although you can simulate the appearance of these headings with `p` elements and style sheets, the heading elements very often contain important contextual information about the structure of the document. With the power to inspect the node hierarchy of a document afforded by modern browsers, a script can generate its own table of contents or outline of a very long document by looking for elements whose `nodeName` properties are in the `hn` family. Therefore, it is a good idea to continue using these elements for contextual purposes, even if you intend to override the default appearance by way of style sheet templates.

As for the scriptable aspects of these six objects, they are essentially the same as the generic contextual objects with the addition of the `align` property. Because each `hn` element is a block-level element, you can use style sheets, rather than the corresponding attribute or property, to set their alignment. The choice is up to you.

Property

align

Value: String Read/Write

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

String values of the `align` property control whether the heading element is aligned with the left margin (`left`), center of the page (`center`), or right margin (`right`). The corresponding `align` attribute is deprecated in HTML 4.0 in favor of the `text-align` style sheet attribute.

Related Items: `text-align` style sheet attribute

hr Element Object

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

For HTML element properties, methods, and event handlers, see Chapter 26, "Generic HTML Element Objects."

Properties	Methods	Event Handlers
`align`		
`color`		
`noShade`		
`size`		
`width`		

Syntax

Accessing `hr` element object properties or methods:

```
(IE4+)     [window.]document.all.elemID.property | method([parameters])
(IE5+/W3C) [window.]document.getElementById("elemID").property |
           method([parameters])
```

About this object

The `hr` element draws a horizontal rule according to size, dimension, and alignment characteristics normally set by the attributes of this element. Style sheets can also specify many of those settings, the latter route being recommended for pages that will be loaded exclusively in pages that support CSS. In modern browsers, your scripts can modify the appearance of an `hr` element either directly through element object properties or through style sheet properties. To reference a specific `hr` element by script, you must assign an `id` attribute to the element and use the `getElementById()` function — the preferred practice used to access HTML elements in modern browsers.

Properties

align

Value: String Read/Write

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

An hr object's horizontal alignment can be controlled via the align property. String values enable you to set it to align with the left margin (left), the center of the page (center), or right margin (right). By default, the element is centered.

Example

Listing 33-2 contains a page that demonstrates the changes to the five hr element object properties: align, color, noShade, size, and width. Along the way, you can see an economical use of the setAttribute() method to do the work for all of the property changes. This page loads successfully in all browsers, including legacy browsers, but the select lists make changes to the text only in modern browsers (because they treat the element as an object).

An hr element (whose id is myHR) is displayed with the browser default settings (100% width, centered, and its "magic" color). Each list controls one of the five hr object properties, and their name attributes are strategically assigned the names of the properties (as you see in a moment). value attributes for option elements contain strings that are to be assigned to the various properties. Each select element invokes the same setHRAttr() function, passing a reference to itself so that the function can inspect details of the element.

The setHRAttr() function reads the string from the value property of the select object that is passed to the function. If a selection is made (that is, other than the first, empty one), the single nested statement uses the setAttribute() method to assign the value to the attribute whose name matches the name of the select element.

LISTING 33-2

Controlling hr Object Properties

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>hr Object Properties</title>
      <script type="text/javascript">
         // one function does all!
         function setHRAttr(select)
         {
            var choice = select.options[select.selectedIndex].value;
            if (choice)
            {
               document.getElementById("myHR").setAttribute(select.name, choice);
```

continued

LISTING 33-2 *(continued)*

```
            }
         }
      </script>
   </head>
   <body>
      <h1>hr Object Properties</h1>
      <br />
      <p>Here is the hr element you will be controlling:</p>
      <hr id="myHR" />
      <form>
         Select an alignment:
         <select name="align" onchange="setHRAttr(this)">
            <option></option>
            <option value="left">Left</option>
            <option value="center">Center</option>
            <option value="right">Right</option>
         </select><br />
         Select a rule color:
         <select name="color" onchange="setHRAttr(this)">
            <option></option>
            <option value="red">Red</option>
            <option value="green">Green</option>
            <option value="blue">Blue</option>
            <option value="#FA8072">Some Hex Triplet Value</option>
         </select><br />
         Select a rule shading (most browsers maintain color when this is set):
         <select name="noShade" onchange="setHRAttr(this)">
            <option></option>
            <option value="true">No Shading</option>
            <option value="false">Shading</option>
         </select><br />
         Select a rule height:
         <select name="size" onchange="setHRAttr(this)">
            <option></option>
            <option value="2">2 (Default)</option>
            <option value="4">4 Pixels</option>
            <option value="10">10 Pixels</option>
         </select><br />
         Select a rule width:
         <select name="width" onchange="setHRAttr(this)">
            <option></option>
            <option value="100%">100% (Default)</option>
            <option value="80%">80%</option>
            <option value="300">300 Pixels</option>
         </select>
      </form>
   </body>
</html>
```

Related Items: `text-align` style sheet attribute

`color`

Value: String Read/Write

Compatibility: WinIE4+, MacIE4+, NN-, Moz+, Safari+, Opera+, Chrome+

An `hr` object's color can be controlled via the `color` property. Values can be either hexadecimal triplets (for example, `#FFCCFF`) or the plain-language color names recognized by most browsers. In either case, the values are case-insensitive strings. If you change the color from the default, the default shading (3-D effect) of the rule disappears. We have yet to find the magic value that lets you return the color to the browser default after it has been set to another color. Also, in WebKit-based browsers, if you change the shading after you have changed the color, the color reverts to gray.

Example

See Listing 33-2, earlier in this chapter, for an example of values that can be used to set the `color` property of an `hr` element object.

Related Items: `color` style sheet attribute

`noShade`

Value: Boolean Read/Write

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

A default `hr` element is displayed with a kind of three-dimensional effect, called *shading*. You can turn shading off under script control by setting the `noShade` property to `true`. But be aware that in the browsers we tested, the `noShade` property is a one-way journey: You cannot restore shading after it is removed. Moreover, in most browsers, default shading is lost if you assign a different color to the rule. As mentioned above, in WebKit-based browsers, if you change the shading after you have changed the color, the color reverts to gray.

Example

See Listing 33-2, earlier in this chapter, for an example of values that can be used to set the `noShade` property of an `hr` element object. Because of the buggy behavior associated with setting this property, adjusting the property in the example has unexpected (and usually undesirable) consequences, even in some of the latest web browsers.

Related Items: `color` style sheet attribute

`size`

Value: Integer Read/Write

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

The size of an `hr` element is its vertical thickness, as controlled via the `size` property. Values are integers, representing the number of pixels occupied by the rule.

Example

See Listing 33-2, earlier in this chapter, for an example of values that can be used to set the `size` property of an `hr` element object.

width

Value: Integer or string Read/Write

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

The width of an `hr` element is controlled via the `width` property. By default, the element occupies the entire width of its parent container (usually the Body).

You can specify width as either an absolute number of pixels (as an integer) or as a percentage of the width of the parent container. Percentage values are strings that include a trailing percent character (`%`).

Example

See Listing 33-2, earlier in this chapter, for an example of values that can be used to set the `width` property of an `hr` element object.

Related Items: `width` style sheet attribute

label Element Object

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

For HTML element properties, methods, and event handlers, see Chapter 26.

Properties	Methods	Event Handlers
accessKey		
form		
htmlFor		

Syntax

Accessing `label` element object properties or methods:

```
(IE4+)     [window.]document.all.elemID.property | method([parameters])
(IE5+/W3C) [window.]document.getElementById("elemID").property |
           method([parameters])
```

About this object

The `label` element lets you assign a contextual relationship between a form control (text field, radio button, select list, and so on) and the otherwise freestanding text that is used to label the control on the page. This element does not control the rendering or physical relationship between the control and the label — the HTML source code order does that. Wrapping a form control label inside a `label` element is important if scripts will be navigating the element hierarchy of a page's content, and the relationship between a form control and its label effects the results of the document parsing. It is also used by the newer rendering engines for the disabled, so this has become a pretty important element to add to your web pages.

Properties

accessKey

Value: String Read/Write

Compatibility: WinIE4+, MacIE5+, NN6+, Moz+, Safari+, Opera+, Chrome+

For most other HTML element objects, the `accessKey` property description is covered in the generic element property descriptions of Chapter 26. The function of the property for the `label` object is the same as the IE implementation for all other elements. The single-character string value is the character key to be used in concert with the OS- and browser-specific modifier key (for example, Ctrl in IE for Windows) to bring focus to the form control associated with the label. This value is best set initially, via the `accesskey` attribute for the `label` element.

Related Items: `accessKey` property of generic elements

form

Value: Form object reference Read-Only

Compatibility: WinIE6+, MacIE5+, NN6+, Moz+, Safari+, Opera+, Chrome+

The `form` property of a `label` element object returns a reference to the form object that contains the form control with which the label is associated. This property can be useful in a node-parsing script that wants to retrieve the form container from the perspective of the label rather than from the form control. The form object reference returned from the `label` element object is the same form object reference returned by the `form` property of any form control object.

Example

If you want to obtain the form for a label whose ID is `myLabel`, you would use the following code:

```
document.getElementById("myLabel").form
```

Related Items: `form` property of `input` element objects

htmlFor

Value: String Read/Write

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

The htmlFor property is a string that contains the id of the form control element with which the label is associated. This value is normally set via the for attribute in the label element's tag. Modifying this property does not alter the position or rendering of the label, but it does change the relationships between label and control.

marquee Element Object

Compatibility: WinIE4+, MacIE4+, NN7+, Moz+, Safari+, Opera+, Chrome+

For HTML element properties, methods, and event handlers, see Chapter 26.

Properties	Methods	Event Handlers
behavior	start()	onbounce
bgColor	stop()	onfinish
direction		onstart
height		
hspace		
loop		
scrollAmount		
scrollDelay		
trueSpeed		
vspace		
width		

Syntax

Accessing marquee element object properties or methods:

```
(IE4+) [window.]document.all.elemID.property | method([parameters])
(IE5+) [window.]document.getElementById("elemID").property |
            method([parameters])
```

About this object

The marquee element began as a Microsoft proprietary element that displays scrolling text within a rectangle specified by the width and height attributes of the element. Text that scrolls in the element goes between the element's start and end tags. The IE4+ object model exposes the element and many properties to the object model for control by script. The element and some of its scriptability is implemented in NN7+/Moz and WebKit- and Presto-based browsers.

Properties

behavior

Value: String Read/Write

Compatibility: WinIE4+, MacIE4+, NN-, Moz+, Safari+, Opera+, Chrome+

The behavior property controls details of the way scrolled text moves within the scrolling space. Values for this property are one of the following three strings: alternate, scroll, and slide. Presto-based browsers such as Opera allow only scroll and alternate. When set to alternate, scrolling alternates between left and right (or up and down, depending on the direction property setting). A value of scroll means that the text marches completely through the space before appearing again. A value of slide causes the text to march into view until the last character is visible. When the slide value is applied as a property (instead of as an attribute value in the tag), the scrolling stops when the text reaches an edge of the rectangle. Default behavior for the marquee element is the equivalent of scroll.

Example

Listing 33-3 contains a page that demonstrates the changes to several marquee element object properties: behavior, bgColor, direction, scrollAmount, and scrollDelay. NN7+/Moz do not react to the slide behavior and the background color settings. See the description of Listing 33-1 for details on the attribute setting script.

LISTING 33-3

Controlling marquee Object Properties

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>marquee Object Properties</title>
      <script type="text/javascript">
         // one function does all!
         function setMARQUEEAttr(select)
         {
            var choice = select.options[select.selectedIndex].value;
            if (choice)
            {
```

continued

marqueeObject.behavior

LISTING 33-3 *(continued)*

```
                document.getElementById("myMARQUEE").setAttribute(select.name, choice);

             }
          }
       </script>
    </head>
    <body>
       <h1>marquee Object Properties</h1>
       <br />
       <hr />
       <marquee id="myMARQUEE" width="400" height="24">This is the marquee
       element object you will be controlling.</marquee>
       <form>
          <input type="button" value="Start Marquee"
                 onclick="document.getElementById('myMARQUEE').start()" />
          <input type="button" value="Stop Marquee"
                 onclick="document.getElementById('myMARQUEE').stop()" />
          <br />
          Select a behavior:
          <select name="behavior" onchange="setMARQUEEAttr(this)">
             <option></option>
             <option value="alternate">Alternate</option>
             <option value="scroll">Scroll</option>
             <option value="slide">Slide</option>
          </select><br />
          Select a background color:
          <select name="bgColor" onchange="setMARQUEEAttr(this)">
             <option></option>
             <option value="red">Red</option>
             <option value="green">Green</option>
             <option value="blue">Blue</option>
             <option value="#FA8072">Some Hex Triplet Value</option>
          </select><br />
          Select a scrolling direction:
          <select name="direction" onchange="setMARQUEEAttr(this)">
             <option></option>
             <option value="left">Left</option>
             <option value="right">Right</option>
             <option value="up">Up</option>
             <option value="down">Down</option>
          </select><br />
          Select a scroll amount:
          <select name="scrollAmount" onchange="setMARQUEEAttr(this)">
             <option></option>
             <option value="4">4</option>
             <option value="6">6 (Default)</option>
             <option value="10">10</option>
          </select><br />
          Select a scroll delay:
          <select name="scrollDelay" onchange="setMARQUEEAttr(this)">
```

```
        <option></option>
        <option value="50">Short</option>
        <option value="85">Normal</option>
        <option value="125">Long</option>
      </select>
    </form>
  </body>
</html>
```

Related Items: `direction` property of `marquee` object

bgColor

Value: Hexadecimal triplet or color name string Read/Write

Compatibility: WinIE4+, MacIE4+, NN-, Moz+, Safari+, Opera+, Chrome+

The `bgColor` property determines the color of the background of the `marquee` element's rectangular space. To set the color of the text, either surround the `marquee` element with a `font` element or apply the `color` style sheet attribute to the `marquee` element. Values for all color properties can be either the common HTML hexadecimal triplet value (for example, `"#00FF00"`) or any of the X11 color names (a list is available at `http://en.wikipedia.org/wiki/Web_colors`).

Example

See Listing 33-3, earlier in this chapter, for an example of how to apply values to the `bgColor` property.

direction

Value: String Read/Write

Compatibility: WinIE4+, MacIE4+, NN7+, Moz+, Safari+, Opera+, Chrome+

The `direction` property lets you get or set the horizontal or vertical direction in which the scrolling text moves. Four possible string values are `left`, `right`, `down`, `up`. NN7/Moz1 observe `left` and `right` only, whereas Moz1.5+ observes `up` and `down` as well. The default value is `left`.

Example

See Listing 33-3, earlier in this chapter, for an example of how to apply values to the `direction` property.

Related Items: `behavior` property of `marquee` object

height
width

Value: Integer Read/Write

Compatibility: WinIE4+, MacIE4+, NN7+, Moz+, Safari+, Opera+, Chrome+

The `height` and `width` properties enable you to get or set the pixel size of the rectangle occupied by the element. NN7/Moz implement `width` only. You can adjust each property independently of the other, but like most attribute-inspired properties of IE objects, if no `height` or `width` attributes are defined in the element's tag, you cannot use these properties to get the size of the element as rendered by default.

hspace
vspace

Value: Integer Read/Write

Compatibility: WinIE4+, MacIE4+, NN-, Moz+, Safari+, Opera+

The `hspace` and `vspace` properties let you get or set the amount of blank margin space surrounding the `marquee` element. Adjustments to the `hspace` property affect both the left and right (horizontal) margins of the element; `vspace` governs both top and bottom (vertical) margins. Margin thicknesses are independent of the height and width of the element.

loop

Value: Integer Read/Write

Compatibility: WinIE4+, MacIE4+, NN-, Moz+, Safari-, Opera+, Chrome-

The `loop` property allows you to discover the number of times the `marquee` element was set to repeat its scrolling, according to the `loop` attribute. Although this property is read/write, modifying it by script does not cause the text to loop only that number of times more before stopping. Treat this property as read-only.

scrollAmount
scrollDelay

Value: Integers Read/Write

Compatibility: WinIE4+, MacIE4+, NN7+, Moz+, Safari+, Opera+, Chrome+

The `scrollAmount` and `scrollDelay` properties control the perceived speed and scrolling smoothness of the `marquee` element text. The number of pixels between redrawings of the scrolling text is controlled by the `scrollAmount` property. The smaller the number, the less jerky the scrolling is (the default value is 6). At the same time, you can control the time in milliseconds between each redrawing of the text with the `scrollDelay` property. The smaller the number, the more frequently redrawing is performed (the default value is 85 or 90, depending on the operating system). Thus, a combination of low `scrollAmount` and `scrollDelay` property values presents the smoothest (albeit slowest) perceived scrolling.

Example

See Listing 33-3, earlier in this chapter, for an example of how to apply values to the `scrollAmount` and `scrollDelay` properties.

Related Items: `trueSpeed` property of `marquee` object

`trueSpeed`

Value: Boolean Read/Write

Compatibility: WinIE4+, MacIE4+, NN-, Moz+, Safari-, Opera-, Chrome-

IE has a built-in regulator that prevents `scrolldelay` attribute or `scrollDelay` property settings below 60 from causing the `marquee` element text to scroll too quickly. If you genuinely want to use a speed faster than 60 (meaning, a value lower than 60), then also set the `trueSpeed` property to `true`.

Related Items: `scrollDelay` property of `marquee` object

Methods

`start()`
`stop()`

Returns: Nothing

Compatibility: WinIE4+, MacIE4+, NN7+, Moz+, Safari+, Opera+, Chrome+

Scripts can start or stop (pause) a `marquee` element via the `start()` and `stop()` methods. Neither method takes parameters, and you are free to invoke them as often as you like after the page loads. Be aware that the `start()` method does not trigger the `onstart` event handler for the object.

Example

See Listing 33-3, earlier in this chapter, for examples of both the `start()` and `stop()` methods, which are invoked in event handlers of separate controlling buttons on the page. Notice, too, that when you have the behavior set to `slide`, stopping and restarting the `marquee` does not cause the scroll action to start from a blank region.

Event Handlers

`onbounce`

Compatibility: WinIE4+, MacIE4+, NN-, Moz+, Safari-, Opera-, Chrome-

The `onbounce` event handler fires only when the `marquee` element has its behavior set to `alternate`. In that back-and-forth mode, each time the text reaches a boundary and is about to

start its return trip, the onbounce event fires. If you truly want to annoy your users, you could have the onbounce event handlers play a sound at each bounce. (I'm kidding — please don't do this.)

Related Items: behavior property of marquee object

onfinish

Compatibility: WinIE4+, MacIE4+, NN-, Moz+, Safari-, Opera-, Chrome-

The onfinish event handler fires only when the marquee element has its loop set to a specific value of 1 or greater. After the final text loop has completed, the onfinish event fires.

Related Items: loop property of marquee object

onstart

Compatibility: WinIE4+, MacIE4+, NN-, Moz+, Safari-, Opera-, Chrome-

The onstart event handler fires as the marquee element begins its scrolling, but only as a result of the page loading. The start() method does not trigger this event handler.

Related Items: start() method of marquee object

Range Object

Compatibility: WinIE-, MacIE-, NN6+, Moz+, Safari+, Opera+, Chrome+

Properties	Methods	Event Handlers
collapsed	cloneContents()	
commonAncestorContainer	cloneRange()	
endContainer	collapse()	
endOffset	compareBoundaryPoints()	
startContainer	compareNode()	
startOffset	comparePoint()	
	createContextualFragment()	
	deleteContents()	
	detach()	
	extractContents()	
	insertNode()	
	intersectsNode()	
	isPointInRange()	

Properties	Methods	Event Handlers
	selectNode()	
	selectNodeContents()	
	setEnd()	
	setEndAfter()	
	setEndBefore()	
	setStart()	
	setStartAfter()	
	setStartBefore()	
	surroundContents()	
	toString()	

Syntax

Creating a Range object:

```
var rangeRef = document.createRange();
```

Accessing Range object properties or methods:

```
(NN6+/Moz)    rengeRef.property | method([parameters])
```

About this object

The Range object is the W3C DOM (Level 2) version of what Microsoft had implemented earlier as its TextRange object. A number of important differences (not the least of which is an almost entirely different property and method vocabulary) distinguish the behaviors and capabilities of these two similar objects. Although Microsoft participated in the W3C DOM Level 2 working groups, no participant from the company is credited on the DOM specification chapter regarding the Range object. Because the W3C version has not been implemented as of IE8, it is unknown if or when IE will eventually implement the W3C version. In the meantime, see the WinIE TextRange object section later in this chapter for comparisons between the two objects. Neither the W3C DOM Range nor Microsoft TextRange objects are implemented in MacIE5.

The purpose of the W3C DOM Range object is to provide hooks to a different "slice" of content (most typically a portion of a document's content) that is not necessarily restricted to the node hierarchy (tree) of a document. Although a Range object can be used to access and modify nodes and elements, it can also transcend node and element boundaries to encompass arbitrary segments of a document's content. The content contained by a range is sometimes referred to as a *selection*, but this does not mean that the text is highlighted on the page, such as a user selection. Instead, the term "selection" here means a segment of the document's content that can be

addressed as a unit, separate from the node tree of the document. As soon as the range is created, a variety of methods let scripts examine, modify, remove, replace, and insert content on the page.

A range object (meaning, an instance of the static `Range` object) has a start point and an end point, which together define the boundaries of the range. The points are defined in terms of an offset count of positions within a container. These counts are usually character positions within text nodes (ignoring any HTML tag or attribute characters), but when both boundaries are at the edges of the same node, the offsets may also be counts of nodes within a container that surrounds both the start and end points. An example helps clarify these concepts.

Consider the following simplified HTML document:

```
<html>
   <body>
      <p>This paragraph has an <em>emphasized</em> segment.</p>
   </body>
</html>
```

You can create a range that encompasses the text inside the `em` element from several points of view, each with its own offset counting context:

1. From the `em` *element's only child node (a text node)*. The offset of the start point is zero, which is the location of the insertion point in front of the first character (lowercase "e"); the end point offset is 10, which is the character position (zero-based) following the lowercase "d."
2. From the `em` *element.* The point of view here is that of the child text node inside the `em` element. Only one node exists here, and the offset for the start point is `0`, whereas the offset for the end point is `1`.
3. From the `p` *element's child nodes (two text nodes and an element node)*. You can set the start point of a range to the very end (counting characters) of the first child text node of the `p` element; you can then set the end point to be in front of the first character of the last child text node of the `p` element. The resulting range encompasses the text within the `em` element.
4. From the `p` *element.* From the point of view of the `p` element, the range can be set with an offset starting with `1` (the second node nested inside the `p` element) and ending with `2` (the start of the third node).

Although these different points of view provide a great deal of flexibility, they also can make it more difficult to imagine how you can use this power. The W3C vocabulary for the `Range` methods, however, helps you figure out what kind of offset measure to use.

A range object's start point could be in one element, and its end point in another. For example, consider the following HTML:

```
<p>And now to introduce our <em>very special</em> guest:</p>
```

If the text shown in boldface indicates the content of a range object, you can see that the range crosses element boundaries in a way that would make HTML element or node object properties difficult to use for replacing that range with some other text. The W3C specification provides guidelines for browser makers on how to handle the results of removing or inserting HTML content that crosses node borders.

An important aspect of the `Range` object is that the size of a range can be zero or more characters. Start and end points always position themselves between characters. When the start point and end point of a range are at the same location, the range acts like a text insertion pointer.

Working with ranges

To create a range object, use the `document.createRange()` method and assign the range object returned by this method to a variable that you can use to control the range:

```
var rng = document.createRange();
```

With an active range stored in a variable, you can use many of the object's methods to adjust the start and end points of the range. If the range is to consist of all of the contents of a node, you have two convenience methods that do so from different points of view: `selectNode()` and `selectNodeContents()`. The sole parameter passed with both methods is a reference to the node whose contents you want to turn into a range. The difference between the two methods is how the offset properties of the range are calculated as a result (see the discussion about these methods later in the chapter for details). Another series of methods (`setStartBefore()`, `setStartAfter()`, `setEndBefore()`, and `setEndAfter()`) let you adjust each end point individually to a position relative to a node boundary. For the most granular adjustment of boundaries, the `setStart()` and `setEnd()` methods let you specify a reference node (where to start counting the offset) and the offset integer value.

If you need to select an insertion point (for example, to insert some content into an existing node), you can position either end point where you want it, and then invoke the `collapse()` method. A parameter determines whether the collapse should occur at the range's start or end point.

A suite of other methods lets your scripts work with the contents of a range directly. You can copy (`cloneContents()`), delete (`deleteContents()`, `extractContents()`), insert a node (`insertNode()`), and even surround a range's contents with a new parent node (`surroundContents()`). Several properties let your scripts examine information about the range, such as the offset values, the containers that hold the offset locations, whether the range is collapsed, and a reference to the next outermost node that contains both the start and end points.

Mozilla added a proprietary method to the `Range` object (which is actually a method of an object that is built around the `Range` object) called `createContextualFragment()`. This method lets scripts create a valid node (of type `DocumentFragment`) from arbitrary strings of HTML content — a feature that the W3C DOM does not (yet) offer. This method was devised at first as a substitute for what eventually became the NN6+/Moz `innerHTML` property.

Using the Range object can be a bit tedious, because it often requires a number of script statements to execute an action. Three basic steps are generally required to work with a Range object:

1. Create the text range.
2. Set the start and end points.
3. Act on the range.

As soon as you are comfortable with this object, you will find it provides a lot of flexibility in scripting interaction with body content. For ideas about applying the Range object in your scripts, see the examples that accompany the descriptions of individual properties and methods in the following sections.

Note

The Evaluator (see Chapter 4, "JavaScript Essentials") automatically initializes a W3C DOM Range **object in browsers that support the feature. You can access the object via the** rng **global variable to work with examples in the following sections.** ■

Properties

collapsed

Value: Boolean Read-Only

Compatibility: WinIE-, MacIE-, NN6+, Moz+, Safari+, Opera+, Chrome+

The collapsed property reports whether a range has its start and end points set to the same position in a document. If the value is true, the range's start and end containers are the same, and the offsets are also the same. You can use this property to verify that a range is in the form of an insertion pointer just prior to inserting a new node:

```
if (rng.collapsed) {
   rng.insertNode(someNewNodeReference);
}
```

Example

Use The Evaluator's predefined rng object to experiment with the collapsed property. Reload the page and set the range to encompass a node:

```
rng.selectNode(document.body)
```

Enter **a.collapsed** into the top text box. The expression returns false because the end points of the range are not the same.

Related Items: endContainer, endOffset, startContainer, startOffset properties; Range.collapse() method

commonAncestorContainer

Value: Node object reference Read-Only

Compatibility: WinIE-, MacIE-, NN6+, Moz+, Safari+, Opera+, Chrome+

The `commonAncestorContainer` property returns a reference to the document tree node that both the start and end points have in common. It is not unusual for a range's start point to be in one node and the end point to be in another. Yet a more encompassing node most likely contains both of those nodes, perhaps even the `document.body` node. The W3C DOM specification also calls the shared ancestor node the *root node* for the range (a term that may make more sense to you).

Example

Use The Evaluator's predefined `rng` object to experiment with the `commonAncestorContainer` property. Reload the page. Now set the start point to the beginning of the contents of the `myEM` element, and set the end point to the end of the surrounding `myP` element:

```
rng.setStartBefore(document.getElementById("myEM").firstChild)
rng.setEndAfter(document.getElementById("myP").lastChild)
```

Verify that the text range is set to encompass content from the `myEM` node (the word "all") and end of `myP` nodes (note that while Safari 1.0 returns the wrong data here, subsequent releases do not):

```
rng.toString()
```

Verify, too, that the two end point containers are different nodes:

```
rng.startContainer.tagName
rng.endContainer.tagName
```

Finally, see what node contains both of these two end points:

```
rng.commonAncestorContainer.id
```

The result is the `myP` element, which both the `myP` and `myEM` nodes have in common.

Related Items: `endContainer`, `endOffset`, `startContainer`, `startOffset` properties; all "set" and "select" methods of the `Range` object

endContainer
startContainer

Value: Node object reference Read-Only

Compatibility: WinIE-, MacIE-, NN6+, Moz+, Safari+, Opera+, Chrome+

The `endContainer` and `startContainer` properties return a reference to the document tree node that contains the range's end point and start point, respectively. Be aware that the object model calculates the container, and that the container may not be the reference you used to set the start and end points of a range. For example, if you use the `selectNode()` method to set

the start and end points of a range to encompass a particular node, the containers of the end points are most likely the next outermost nodes. Thus, if you want to expand a range to the start of the node that contains the current range's start point, you can use the value returned by the `startContainer` property as a parameter to the `setStartBefore()` method:

```
rng.setStartBefore(rng.startContainer)
```

Example

Use The Evaluator's predefined `rng` object to experiment with the `endContainer` and `startContainer` properties. Reload the page and set the range to encompass the `myEM` element:

```
rng.selectNode(document.getElementById("myEM")
```

Inspect the containers for both the start and end points of the selection:

```
rng.startContainer.id
rng.endContainer.id
```

The range encompasses the entire `myEM` element, so the start and end points are outside of the element. Therefore, the container of both start and end points is the `myP` element that also surrounds the `myEM` element.

Related Items: `commonAncestor`, `endOffset`, `startOffset` properties; all "set" and "select" methods of the `Range` object

endOffset
startOffset

Value: Integer Read-Only

Compatibility: WinIE-, MacIE-, NN6+, Moz+, Safari+, Opera+, Chrome+

The `endOffset` and `startOffset` properties return an integer count of the number of characters or nodes for the location of the range's end point and start point, respectively. These counts are relative to the node that acts as the container node for the position of the boundary (see the `Range.endContainer` and `Range.startContainer` properties earlier in this chapter).

When a boundary is at the edge of a node (or perhaps "between" nodes is a better way to say it), the integer returned is the offset of nodes (zero-based) within the boundary's container. But when the boundary is in the middle of a text node, the integer returned is an index of the character position within the text node. The fact that each boundary has its own measuring system (nodes versus characters, relative to different containers) can get confusing if you're not careful, because conceivably the integer returned for an end point could be smaller than that for the start point. Consider the following nested elements:

```
<p>This paragraph has an <em>emphasized</em> segment.</p>
```

The next script statements set the start of the range to a character within the first text node and the end of the range to the end of the em node:

```
var rng = document.createRange();
rng.setStart(document.getElementById("myP").firstChild, 19);
rng.setEndAfter(document.getElementById("myEM"));
```

Using boldface to illustrate the body text that is now part of the range, and the pipe (|) character to designate the boundaries as far as the nodes are concerned, here is the result of the preceding script execution:

```
<p id="myP">This paragraph has |an <em id="myEM">emphasized</em>| segment.</p>
```

Because the start of the range is in a text node (the first child of the p element), the range's startOffset value is 19, which is the zero-based character position of the "a" in the word "an." The end point, however, is at the end of the em element. The system recognizes this point as a node boundary, and thus counts the endOffset value within the context of the end container: the p element. The endOffset value is 2 (the p element's text node is node index 0; the em element is node index 1; and the position of the end point is at the start of the p element's final text node, at index 2).

For the endOffset and startOffset values to be of any practical use to a script, you must also use the endContainer and startContainer properties to read the context for the offset integer values.

Example

Use The Evaluator's predefined rng object to experiment with the endOffset and startOffset properties, following similar paths you just saw in the description. Reload the page and set the range to encompass the myEM element and then move the start point outward to a character within the myP element's text node:

```
rng.selectNode(document.getElementById("myEM"))
rng.setStart(document.getElementById("myP").firstChild, 7)
```

Inspect the node types of the containers for both the start and end points of the selection:

```
rng.startContainer.nodeType
rng.endContainer.nodeType
```

The startContainer node type is 3 (text node), whereas the endContainer node type is 1 (element). Now inspect the offsets for both the start and end points of the selection:

```
rng.startOffset
rng.endOffset
```

Related Items: endContainer, startContainer properties; all "set" and "select" methods of the Range object

Methods

cloneContents()
cloneRange()

Returns: `DocumentFragment` node reference; `Range` object reference

Compatibility: WinIE-, MacIE-, NN7+, Moz+, Safari+, Opera+, Chrome+

The `cloneContents()` method (available in NN7+) takes a snapshot copy of the contents of a `Range` object and returns a reference to that copy. The copy is stored in the browser's memory, but is not part of the document tree. The `cloneRange()` method (available in NN6+) performs the same action on an entire range and stores the range copy in the browser's memory. A range's contents can consist of portions of multiple nodes and may not be surrounded by an element node; that's why its data is of the type `DocumentFragment` (one of the W3C DOM's node types). Because a `DocumentFragment` node is a valid node, it can be used with other document tree methods where nodes are required as parameters. Therefore, you can clone a text range to insert a copy elsewhere in the document.

In contrast, the `cloneRange()` method deals with range objects. Although you are always free to work with the contents of a range object, the `cloneRange()` method returns a reference to a range object, which acts as a kind of wrapper to the contents (just as it does when the range is holding content in the main document). You can use the `cloneRange()` method to obtain a copy of one range to compare the end points of another range (via the `Range.compareBoundaryPoints()` method).

Example

Use The Evaluator's predefined `rng` object in NN7+/Moz/ and WebKit- and Presto-based browsers to see the `cloneContents()` method in action. Begin by reloading the page and setting the range to the `myP` paragraph element:

```
rng.selectNode(document.getElementById("myP"))
```

Next, clone the original range and preserve the copy in variable `b`:

```
b = rng.cloneContents()
```

Move the original range so that it is an insertion point at the end of the body by first expanding it to encompass the entire body and then collapsing it to the end:

```
rng.selectNode(document.body)
rng.collapse(false)
```

Now, insert the copy at the very end of the body:

```
rng.insertNode(b)
```

If you scroll to the bottom of the page, you see a copy of the text. While Safari 1.0 appears to miscalculate the range's boundary points after `collapse()`, causing a DOM hierarchy error when invoking `insertNode()`, subsequent versions do not. But you can also use `appendChild()` or `insertBefore()` on any element node to put the cloned range into the document tree.

See the description of the `compareBoundaryPoints()` method later in this chapter to see an example of the `cloneRange()` method.

Related Items: `compareBoundaryPoints()`, `extractContents()` methods

collapse([*startBoolean*])

Returns: Nothing

Compatibility: WinIE-, MacIE-, NN6+, Moz+, Safari+, Opera+, Chrome+

Use the `collapse()` method to shrink a range from its current size down to a single insertion point between characters. Collapsing a range becomes more important than you may think at first, especially in a function that is traversing the body of the page or a large chunk of text. For example, in a typical looping word-counting script, you create a text range that encompasses the body fully. To begin counting words, you can first collapse the range to the insertion point at the very beginning of the range. Next, use the `expand()` method to set the range to the first word of text (and increment the counter if the `expand()` method returns `true`). At that point, the text range extends around the first word. You want the range to collapse at the end of the current range so that the search for the next word starts after the current one. Use `collapse()` once more, but this time include parameters.

The optional parameter of the `collapse()` method is a Boolean value that directs the range to collapse itself either at the start or end of the current range. The default behavior is the equivalent of a value of `true`, which means that unless otherwise directed, a `collapse()` method shifts the text range to the point in front of the current range. This method works great at the start of a word-counting script, because you want the text range to collapse to the start of the text. But for subsequent movements through the range, you want to collapse the range so that it is after the current range. Thus, you include a `false` parameter to the `collapse()` method.

Example

Refer to Listings 33-11 and 26-14 to see the `collapse()` method at work (albeit with the IE `TextRange` object).

Related Items: `Range.setEnd()`, `Range.setStart()` methods

compareBoundaryPoints(*typeInteger, sourceRangeRef*)

Returns: Integer (`-1`, `0`, or `1`)

Compatibility: WinIE-, MacIE-, NN6+, Moz+, Safari+, Opera+, Chrome+

Generating multiple range objects and assigning them to different variables is not a problem. You can then use the `compareBoundaryPoints()` method to compare the relative positions of start and end points of both ranges. One range is the object you use to invoke the

`compareBoundaryPoints()` method, and the other range is the second parameter of the method. The order in which you reference the two ranges influences the results, based on the value assigned to the first parameter.

Values for the first parameter can be one of four constant values that are properties of the static `Range` object: `Range.START_TO_START`, `Range.START_TO_END`, `Range.END_TO_START`, and `Range.END_TO_END`. What these values specify is which point of the current range is compared with which point of the range passed as the second parameter. For example, consider the following body text that has two text ranges defined within it:

```
It was the best of times.
```

The first text range (assigned in our discussion here to variable `rng1`) is shown in boldface, whereas the second text range (`rng2`) is shown in bold-italic. In other words, `rng2` is nested inside `rng1`. We can compare the position of the start of `rng1` against the position of the start of `rng2` by using the `Range.START_TO_START` value as the first parameter of the `compareBoundaryPoints()` method:

```
var result = rng1.compareBoundaryPoints(Range.START_TO_START, rng2);
```

The value returned from the `compareBoundaryPoints()` method is an integer of one of three values. If the positions of both points under test are the same, then the value returned is `0`. If the start point of the (so-called) source range is before the range on which you invoke the method, the value returned is `-1`; in the opposite positions in the code, the return value is `1`. Therefore, from the previous example, because the start of `rng1` is before the start of `rng2`, the method returns `-1`. If you change the statement to invoke the method on `rng2`, as in

```
var result = rng2.compareBoundaryPoints(Range.START_TO_START, rng1);
```

the result is `1`.

In practice, this method is helpful in knowing if two ranges are the same, if one of the boundary points of both ranges are the same, or if one range starts where the other ends.

Example

The page rendered by Listing 33-4 lets you experiment with text range comparisons in NN6+/Moz/W3C. The bottom paragraph contains a `span` element that has a `Range` object assigned to its nested text node after the page loads (in the `init()` function). That fixed range becomes a solid reference point for you to use while you select text in the paragraph.

After you make a selection, all four versions of the `compareBoundaryPoints()` method run to compare the start and end points of the fixed range against your selection. One column of the results table shows the raw value returned by the `compareBoundaryPoints()` method, whereas the third column puts the results into plain language.

To see how this page works, begin by selecting the first word of the fixed text range (carefully drag the selection from the first red character). You can see that the starting positions of both ranges are the same, because the returned value is `0`. Because all of the invocations of the

compareBoundaryPoints() method are on the fixed text range, all comparisons are from the point of view of that range. Thus, the first row of the table for the START_TO_END parameter indicates that the start point of the fixed range comes before the end point of the selection, yielding a return value of -1.

Other selections to make include:

- Text that starts before the fixed range and ends inside the range
- Text that starts inside the fixed range and ends beyond the range
- Text that starts and ends precisely at the fixed range boundaries
- Text that starts and ends before the fixed range
- Text that starts after the fixed range

Study the returned values and the plain language results and see how they align with the selection you made.

LISTING 33-4

Lab for NN6+/Moz/W3C compareBoundaryPoints() Method

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>TextRange.compareBoundaryPoints() Method</title>
      <style type="text/css">
         td
         {
            text-align:center;
         }
         .propName
         {
            font-family:Courier, monospace;
         }
         #fixedRangeElem {
            color:red; font-weight:bold;
         }
      </style>
      <script type="text/javascript">
         var fixedRange;

         function replaceHTML(elem, text)
         {
            while(elem.firstChild)
            {
               elem.removeChild(elem.firstChild);
            }
            elem.appendChild(document.createTextNode(text));
         }
```

continued

LISTING 33-4 *(continued)*

```
function setAndShowRangeData()
{
   try
   {
      var selectedRange = window.getSelection();
      selectedRange = selectedRange.getRangeAt(0);
      var result1 = fixedRange.compareBoundaryPoints(Range.START_TO_END,
         selectedRange);
      var result2 = fixedRange.compareBoundaryPoints(Range.START_TO_START,
         selectedRange);
      var result3 = fixedRange.compareBoundaryPoints(Range.END_TO_START,
         selectedRange);
      var result4 = fixedRange.compareBoundaryPoints(Range.END_TO_END,
         selectedRange);

      replaceHTML(document.getElementById("B1"), result1);
      replaceHTML(document.getElementById("compare1"),
         getDescription(result1));
      replaceHTML(document.getElementById("B2"), result2);
      replaceHTML(document.getElementById("compare2"),
         getDescription(result2));
      replaceHTML(document.getElementById("B3"), result3);
      replaceHTML(document.getElementById("compare3"),
         getDescription(result3));
      replaceHTML(document.getElementById("B4"), result4);
      replaceHTML(document.getElementById("compare4"),
        getDescription(result4));
   }
   catch(err)
   {
      alert("Vital Range or Selection object services are not yet
      implemented in this browser.");
   }
}

function getDescription(comparisonValue)
{
   switch (comparisonValue)
   {
   case -1 :
      return "comes before";
      break;
   case 0 :
      return "is the same as";
      break;
   case 1 :
      return "comes after";
      break;
   default :
```

```
            return "vs.";
         }
      }

      function init()
      {
         fixedRange = document.createRange();
         fixedRange.selectNodeContents(document.getElementById
                  ("fixedRangeElem").firstChild);
         fixedRange.setEnd(fixedRange.endContainer,
                   fixedRange.endContainer.nodeValue.length);
      }
   </script>
</head>
<body onload="init()">
   <h1>TextRange.compareBoundaryPoints() Method</h1>
   <hr />
   <p>Select text in the paragraph in various places relative to the fixed
      text range (shown in red). See the relations between the fixed and
      selected ranges with respect to their start and end points.
   </p>
   <table id="results" border="1" cellspacing="2" cellpadding="2">
      <tr>
         <th>Property</th>
         <th>Returned Value</th>
         <th>Fixed Range vs. Selection</th>
      </tr>
      <tr>
         <td class="propName">StartToEnd</td>
         <td class="count" id="B1"> </td>
         <td class="count" id="C1">Start of Fixed <span
         id="compare1">vs.</span> End of Selection</td>
      </tr>
      <tr>
         <td class="propName">StartToStart</td>
         <td class="count" id="B2"> </td>
         <td class="count" id="C2">Start of Fixed <span
         id="compare2">vs.</span> Start of Selection</td>
      </tr>
      <tr>
         <td class="propName">EndToStart</td>
         <td class="count" id="B3"> </td>
         <td class="count" id="C3">End of Fixed <span
         id="compare3">vs.</span> Start of Selection</td>
      </tr>
      <tr>
         <td class="propName">EndToEnd</td>
         <td class="count" id="B4"> </td>
         <td class="count" id="C4">End of Fixed <span
         id="compare4">vs.</span> End of Selection</td>
      </tr>
   </table>
```

continued

LISTING 33-4 *(continued)*

```
      <hr />
      <p onmouseup="setAndShowRangeData()">Lorem ipsum dolor sit,
         <span id="fixedRangeElem">consectetaur adipisicing elit</span>, sed
         do eiusmod tempor incididunt ut labore et dolore aliqua. Ut enim
         adminim veniam, quis nostrud exercitation ullamco laboris nisi ut
         aliquip ex ea commodo consequat.
      </p>
   </body>
</html>
```

compareNode(*nodeReference*)

Returns: Integer (0, 1, 2, or 3)

Compatibility: WinIE-, MacIE-, NN6+, Moz+, Safari+, Opera-, Chrome+

The `compareNode()` method returns an integer code that indicates the relative position of the specified node with respect to the range. The node is passed as the only parameter to the method, and the value returned indicates the relative location of the node. The following four constants may be returned from the `compareNode()` method, and correspond to integer values in the range 0–3: `Range.NODE_BEFORE`, `Range.NODE_AFTER`, `Range.NODE_BEFORE_AND_AFTER`, `Range.NODE_INSIDE`. The first two values explain themselves, but the third value (`Range.NODE_BEFORE_AND_AFTER`) indicates that the node begins before the range and ends after the range. The final value (`Range.NODE_INSIDE`), on the other hand, indicates that the node is contained in its entirety by the range.

Related Items: `comparePoint()` method

comparePoint(*nodeReference, offset*)

Returns: Integer (-1, 0, or 1)

Compatibility: WinIE-, MacIE-, NN6+, Moz+, Safari+, Opera+, Chrome+

The `comparePoint()` method returns an integer code that indicates the relative position of the specified node at a certain offset with respect to the range. The node (as an object reference) and its offset (an integer count of an element's nodes or a text node's characters) are passed as parameters to the method, and the value returned indicates the relative location of the node. This location is specified with respect to the point (node and offset), not the range. The location of the node is indicated by the integer values -1, 0, and 1, where -1 indicates that the point comes before the start of the range, 0 indicates that the point is located within the range, and 1 reveals that the point comes after the end of the range.

Related Items: `compareNode()` method

createContextualFragment("*text*")

Returns: W3C DOM document fragment node

Compatibility: WinIE-, MacIE-, NN6+, Moz+, Safari+, Opera+, Chrome+

The `createContextualFragment()` method provides a way, within the context of the W3C DOM Level 2 node hierarchy, to create a string of HTML text (with or without HTML tags, as needed) for insertion or appendage to existing node trees. During the development of the NN6 browser, this method closed a gap that was eventually closed by Netscape's adoption of the Microsoft proprietary `innerHTML` property. The method obviates the need for tediously assembling a complex HTML element via a long series of `document.createElement()` and `document.createTextNode()` methods for each segment, plus the assembly of the node tree, prior to inserting it into the actual visible document.

Note

Although the existence of the `innerHTML` property of all `element` objects reduced the need for the `createContextualFragment()` method, the W3C approach of removing and appending child nodes in order to alter HTML content (see the `replaceHTML()` function in Listing 33-4) is now the preferred technique for altering the "inner HTML" of an element. ■

The parameter to the `createContextualFragment()` method is any text, including HTML tags. To invoke the method, however, you need to have an existing range object available. Therefore, the sequence used to generate a document fragment node is

```
var rng = document.createRange();
rng.selectNode(document.body); // any node will do
var fragment = rng.createContextualFragment("<h1>Howdy</h1>");
```

As a document fragment, the node is not part of the document node tree until you use the fragment as a parameter to one of the tree modification methods, such as `Node.insertBefore()` or `Node.appendChild()`.

Example

Use The Evaluator's predefined `rng` object to replace an existing document tree node with the fragment. Begin by creating the fragment from The Evaluator's built-in range:

```
b = rng.createContextualFragment("<span style='font-size:22pt'>a bunch
    of </span>")
```

This fragment consists of a `span` element node with a text node nested inside. At this point, you can inspect the properties of the document fragment by entering `b` into the bottom text box.

To replace the `myEM` element on the page with this new fragment, use the `replaceChild()` method on the enclosing `myP` element:

```
document.getElementById("myP").replaceChild(b, document.getElementById("myEM"))
```

The fragment now becomes a legitimate child node of the `myP` element and can be referenced like any node in the document tree. For example, if you enter the following statement into the

top text box of The Evaluator, you can retrieve a copy of the text node inside the new `span` element:

```
document.getElementById("myP").childNodes[1].firstChild.nodeValue
```

Related Items: `Node` object (Chapter 26, "Generic HTML Element Objects")

deleteContents()

Returns: Nothing

Compatibility: WinIE-, MacIE-, NN6+, Moz+, Safari+, Opera+, Chrome+

The `deleteContents()` method removes all contents of the current range from the document tree. After deletion, the range collapses to an insertion point where surrounding content (if any) cinches up to its neighbors.

Alignment of a range's boundaries forces the browser to make decisions about how element boundaries inside the range are treated after the deletion. An easy deletion is one for which the range boundaries are symmetrical. For example, consider the following HTML with a range highlighted in bold:

```
<p>One paragraph with an <em>emphasis</em> inside.</p>
```

After you delete the contents of this range, the text node inside the `em` element disappears, but the `em` element remains in the document tree (with no child nodes). Similarly, if the range is defined as being the entire second child node of the `p` element, as follows

```
<p>One paragraph with an <em>emphasis</em> inside.</p>
```

then deleting the range contents removes both the text node and the `em` element node, leaving the `p` element with a single, unbroken text node as a child (although, as in the previous case, an extra space would be between the words "an" and "inside" because the `em` element does not encompass a space on either side).

When range boundaries are not symmetrical, the browser does its best to maintain document tree integrity after the deletion. Consider the following HTML and range:

```
<p>One paragraph with an <em>emphasis</em> inside.</p>
```

After deleting this range's contents, the document tree for this segment looks like the following:

```
<p>One paragraph <em>phasis</em> inside.</p>
```

The range collapses to an insertion point just before the `<em>` tag. But notice that the `em` element persists to take care of the text still under its control. Many other combinations of range boundaries and nodes are possible, so be sure that you check out the results of a contents deletion for asymmetrical boundaries before applying the deletion.

Example

Use The Evaluator's predefined `rng` object to experiment with deleting contents of both a text node and a complete element node. Begin by adjusting the text range to the text node inside the `myEM` element (enter the third statement, which wraps below, as one continuous expression):

```
rng.setStart(document.getElementById("myEM").firstChild, 0)
rng.setEnd(document.getElementById("myEM").lastChild, ↩
    document.getElementById("myEM").lastChild.length)
```

Verify the makeup of the range by entering `a` into the bottom text box and inspecting its properties. Both containers are text nodes (they happen to be the same text node), and offsets are measured by character positions.

Now, delete the contents of the range:

```
rng.deleteContents()
```

The italicized word "all" is gone from the tree, but the `myEM` element is still there. To prove it, put some new text inside the element:

```
document.getElementById("myEM").innerHTML = "a band of "
```

The italic style of the `em` element applies to the text, as it should.

Next, adjust the range boundaries to include the `myEM` element tags, as well:

```
rng.selectNode(document.getElementById("myEM"))
```

Inspect the `Range` object's properties again by entering `rng` into the bottom text box. The container nodes are the `p` element that surrounds the `em` element; the offset values are measured in nodes. Delete the range's contents:

```
rng.deleteContents()
```

Not only is the italicized text gone, but the `myEM` element is gone, too. The `myP` element now has two child nodes: the two text nodes that used to flank the `em` element. The following entries into the top text box of The Evaluator verify this fact:

```
document.getElementById("myP").childNodes.length
document.getElementById("myP").childNodes[0].nodeValue
```

To combine the two sibling text nodes into one, invoke the `normalize()` method of the container:

```
document.getElementById("myP").normalize()
```

Check the number of child nodes again to verify the results.

Related Items: `Range.extractContents()` method

detach()

Returns: Nothing

Compatibility: WinIE-, MacIE-, NN6+, Moz+, Safari+, Opera+, Chrome+

The `detach()` method instructs the browser to release the current range object from the object model. In the process, the range object is nulled out to the extent that an attempt to access the object results in a script error. You can still assign a new range to the same variable if you like. You are not required to detach a range when you're finished with it, and the browser resources employed by a range are not that large. But it is good practice to "clean up after yourself," especially when a script repetitively creates and manages a series of new ranges.

Related Items: `document.createRange()`

extractContents()

Returns: `DocumentFragment` node reference

Compatibility: WinIE-, MacIE-, NN7+, Moz+, Safari+, Opera+, Chrome+

The `extractContents()` method deletes the contents of the range and returns a reference to the document fragment node that is held in the browser memory, but which is no longer part of the document tree. A range's contents can consist of portions of multiple nodes and may not be surrounded by an element node; that's why its data is of the type `DocumentFragment` (one of the W3C DOM's node types). Because a `DocumentFragment` node is a valid node, it can be used with other document tree methods where nodes are required as parameters. Therefore, you can extract a text range from one part of a document to insert elsewhere in the document.

Example

Use The Evaluator's predefined `rng` object in NN7+/Moz/Safari to see how the `extractContents()` method works. Begin by setting the built-in range object to contain the text of the `myP` paragraph element.

```
rng.selectNode(document.getElementById("myP"))
```

Next, extract the original range's content and preserve the copy in variable `b`:

```
b = a.extractContents()
```

Now, insert the extracted fragment at the very end of the body:

```
document.body.appendChild(b)
```

If you scroll to the bottom of the page, you see a copy of the text.

Related Items: `cloneContents()`, `deleteContents()` methods

insertNode(*nodeReference*)

Returns: Nothing

Compatibility: WinIE-, MacIE-, NN7+, Moz+, Safari+, Opera+, Chrome+

The `insertNode()` method inserts a node at the start point of the current range. The node being inserted may be an element or text fragment, and its source can be any valid node creation mechanism, such as the `document.createTextNode()` method, or any node extraction method.

Example

Listing 33-5 demonstrates the `insertNode()` method plus some additional items from the `selection` object. The example even includes a rudimentary undo buffer for scripted changes to a text range. In the page generated by this listing, users can select any text in a paragraph and have the script automatically convert the text to all uppercase characters. The task of replacing a selection with other text requires several steps, starting with the selection, which is retrieved via the `window.getSelection()` method. After making sure the selection contains some text (that is, the selection isn't collapsed), the selection is preserved as a range object so that the starting text can be stored in a global variable (as a property of the `undoBuffer` global variable object). After that, the selection is deleted from the document tree, leaving the selection as a collapsed insertion point. A copy of that selection in the form of a range object is preserved in the `undoBuffer` object so that the undo script knows where to reinsert the original text. A new text node is created with an uppercase version of the original text, and, finally, the `insertNode()` method is invoked to stick the converted text into the collapsed range.

Undoing this operation works in reverse. Original locations and strings are copied from the `undoBuffer` object. After creating the range with the old start and end points (which represent a collapsed insertion point), the resurrected text (converted to a text node) is inserted into the collapsed range. For good housekeeping, the `undoBuffer` object is restored to its unused form.

LISTING 33-5

Inserting a Node into a Range

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>NN Selection Object Replacement</title>
      <script type="text/javascript">
         var undoBuffer = {rng:{}, txt:""};

         function convertSelection()
         {
            var sel, grossRng, netRng, newText;
            try
            {
```

continued

LISTING 33-5 *(continued)*

```
            sel = window.getSelection();
            if (!sel.isCollapsed)
            {
               grossRng = sel.getRangeAt(0);
               undoBuffer.txt = grossRng.toString();
               undoBuffer.rng.startContainer = grossRng.startContainer;
               undoBuffer.rng.startOffset = grossRng.startOffset;
               undoBuffer.rng.endContainer = grossRng.endContainer;
               undoBuffer.rng.endOffset = grossRng.endOffset;
               sel.deleteFromDocument();
               netRng = sel.getRangeAt(0);
               newText = document.createTextNode(undoBuffer.txt.toUpperCase());
               netRng.insertNode(newText);
               netRng.commonAncestorContainer.parentNode.normalize();
            }
         }
         catch(err)
         {
            alert("Vital Range or Selection object services are not yet ↩
            implemented in this browser.");
         }
      }

      function undoConversion()
      {
         var rng, oldText;
         if (undoBuffer.rng)
         {
            rng = document.createRange();
            rng.setStart(undoBuffer.rng.startContainer,
                 undoBuffer.rng.startOffset);
            rng.setEnd(undoBuffer.rng.endContainer, undoBuffer.rng.endOffset);
            rng.extractContents();
            oldText = document.createTextNode(undoBuffer.txt);
            rng.insertNode(oldText);
            undoBuffer.rng = {};
            undoBuffer.txt = "";
         }
      }
   </script>
</head>
<body>
   <h1 id="H1_1">NN6+/Moz/W3C Selection Object Replacement</h1>
   <hr />
   <p onmouseup="convertSelection()">This paragraph contains text
      that you can select. Selections are deleted and replaced by all
      uppercase versions of the selected text.
   </p>
   <button onclick="undoConversion()">Undo Last</button>
```

```
      <button onclick="location.reload(true)">Start Over</button>
   </body>
</html>
```

intersectsNode(*nodeReference*)

Returns: Boolean

Compatibility: WinIE-, MacIE-, NN6+, Moz+, Safari+, Opera+, Chrome+

The `intersectsNode()` method returns a Boolean value that indicates whether (`true`) or not (`false`) any part of the range overlaps the node whose reference is passed as the method's parameter.

Related Items: `compareNode()` method

isPointInRange(*nodeReference, offset*)

Returns: Boolean

Compatibility: WinIE-, MacIE-, NN6+, Moz+, Safari+, Opera-, Chrome+

The `isPointInRange()` method returns a Boolean value that indicates whether (`true`) or not (`false`) the specified node (a node object reference) and offset (integer count of an element's nodes or a text node's characters) are located entirely within the range.

selectNode(*nodeReference*)
selectNodeContents(*nodeReference*)

Returns: Nothing

Compatibility: WinIE-, MacIE-, NN6+, Moz+, Safari+, Opera+, Chrome+

The `selectNode()` and `selectNodeContents()` methods are convenience methods for setting both end points of a range to surround a node or a node's contents. The kind of node you supply as the parameter to either method (text node or element node) has a bearing on the range's container node types and units of measure for each (see the container- and offset-related properties of the `Range` object earlier in this chapter).

Starting with the `selectNode()` method, if you specify an element node as the one to select, the start and end container node of the new range is the next outermost element node; offset values count nodes within that parent element. If you specify a text node to be selected, the container node for both ends is the parent element of that text node; offset values count the nodes within that parent.

With the `selectNodeContents()` method, the start and end container nodes are the very same element specified as the parameter; offset values count the nodes within that element. If you specify a text node's contents to be selected, the text node is the start and end parent, but the range is collapsed at the beginning of the text.

By and large, you specify element nodes as the parameter to either method, allowing you to set the range to either encompass the element (via `selectNode()`) or just the contents of the element (via `selectNodeContents()`).

Example

Use The Evaluator's predefined `rng` object to see the behavior of both the `selectNode()` and `selectNodeContents()` methods work. Begin by setting the range object's boundaries to include the `myP` element node:

```
rng.selectNode(document.getElementById("myP"))
```

Enter `a` into the bottom text box to view the properties of the range. Notice that because the range has selected the entire paragraph node, the container of the range's start and end points is the `body` element of the page (the parent element of the `myP` element).

Now change the range so that it encompasses only the contents of the `myP` element:

```
rng.selectNodeContents(document.getElementById("myP"))
```

Click the List Properties button to view the current properties of the range. The container of the range's boundary points is the `p` element that holds the element's contents.

Related Items: `setEnd()`, `setEndAfter()`, `setEndBefore()`, `setStart()`, `setStartAfter()`, `setStartBefore()` methods

setEnd(*nodeReference*, *offset*)
setStart(*nodeReference*, *offset*)

Returns: Nothing

Compatibility: WinIE-, MacIE-, NN6+, Moz+, Safari+, Opera+, Chrome+

You can adjust the start and end points of a text range independently via the `setStart()` and `setEnd()` methods. Although not as convenient as the `selectNode()` or `selectNodeContents()` methods, these two methods give you the ultimate in granularity over precise positioning of a range boundary.

The first parameter to both methods is a reference to a node. This reference can be an element or text node, but your choice here also influences the kind of measure applied to the integer offset value supplied as the second parameter. When the first parameter is an element node, the offset counts are in increments of child nodes inside the specified element node. But if the first parameter is a text node, the offset counts are in increments of characters within the text node.

When you adjust the start and end points of a range with these methods, you have no restrictions to the symmetry of your boundaries. One boundary can be defined relative to a text node, and the other relative to an element node — or vice versa.

To set the end point of a range to the last node or character within a text node (depending on the unit of measure for the `offset` parameter), you can use the `length` property of the units

being measured. For example, to set the end point to the end of the last node within an element (perhaps there are multiple nested elements and text nodes within that outer element), you can use the first parameter reference to help you get there:

```
rng.setEnd(document.getElementById("myP"), ↩
   document.getElementById("myP").childNodes.length);
```

These kinds of expressions get lengthy, so you may want to make a shortcut to the reference, to simplify the values of the parameters, as shown in this version that sets the end point to after the last character of the last text node of a `p` element:

```
var nodeRef = document.getElementById("myP").lastChild;
rng.setEnd(nodeRef, nodeRef.nodeValue.length);
```

In both previous examples with the `length` properties, the values of those properties are always pointing to the node or character position after the final object because the index values for those objects' counts are zero-based. Also bear in mind that if you want to set a range end point at the edge of a node, you have four other methods to choose from (`setEndAfter()`, `setEndBefore()`, `setStartAfter()`, and `setStartBefore()`). The `setEnd()` and `setStart()` methods are best used when an end point needs to be set at a location other than at a node boundary.

Example

Use The Evaluator's predefined `rng` object to experiment with both the `setStart()` and `setEnd()` methods. For the first range, set the start and end points to encompass the second node (the `myEM` element) inside the `myP` element:

```
rng.setStart(document.getElementById("myP"), 1)
rng.setEnd(document.getElementById("myP"), 2)
```

The text encompassed by the range consists of the word "all" plus the trailing space that is contained by the `myEM` element. Prove this by entering the following statement into the top text box (while Safari 1.0 returns an incorrect value, later versions do not):

```
rng.toString()
```

If you then click the Results box to the right of the word "all," you see that the results contain the trailing space. Yet, if you examine the properties of the range (enter `a` into the bottom text box), you see that the range is defined as actually starting before the `myEM` element, and ending after it.

Next, adjust the start point of the range to a character position inside the first text node of the `myP` element:

```
rng.setStart(document.getElementById("myP").firstChild, 11)
```

Click the List Properties button to see that the `startContainer` property of the range is the text node, and that the `startOffset` measures the character position. All end boundary

properties, however, have not changed. Enter `rng.toString()` in the top box again to see that the range now encompasses text from two of the nodes inside the `myP` element.

You can continue to experiment by setting the start and end points to other element and text nodes on the page. After each adjustment, verify the properties of the `a` range object and the text it encompasses (via `rng.toString()`).

Related Items: `selectNode()`, `selectNodeContents()`, `setEndAfter()`, `setEndBefore()`, `setStartAfter()`, `setStartBefore()` methods

setEndAfter(*nodeReference*)
setEndBefore(*nodeReference*)
setStartAfter(*nodeReference*)
setStartBefore(*nodeReference*)

Returns: Nothing

Compatibility: WinIE-, MacIE-, NN6+, Moz+, Safari+, Opera+, Chrome+

You can adjust the start and end points of a text range relative to existing node boundaries via your choice of these four methods. The "before" and "after" designations are used to specify which side of the existing node boundary the range should have for its boundary. For example, using `setStartBefore()` and `setEndAfter()` with the same element node as a parameter is the equivalent of the `selectNode()` method on that element. You may also specify a text node as the parameter to any of these methods. But because these methods work with node boundaries, the offset values are always defined in terms of node counts, rather than character counts. At the same time, however, the boundaries do not need to be symmetrical, so that one boundary can be inside one node and the other boundary inside another node.

Example

Use The Evaluator's predefined `rng` object to experiment with all four methods. For the first range, set the start and end points to encompass the `myEM` element inside the `myP` element:

```
rng.setStartBefore(document.getElementById("myEM"))
rng.setEndAfter(document.getElementById("myEM"))
```

The text encompassed by the range consists of the word "all" plus the trailing space that is contained by the `myEM` element. Prove this by entering the following statement into the top text box (while Safari 1.0 returns an incorrect value later versions do not):

```
rng.toString()
```

Next, adjust the start point of the range to the beginning of the first text node of the `myP` element:

```
rng.setStartBefore(document.getElementById("myP").firstChild)
```

Enter `rng` into the bottom text box to see that the `startContainer` property of the range is the `p` element node, whereas the `endContainer` property points to the `em` element.

You can continue to experiment by setting the start and end points to before and after other element and text nodes on the page. After each adjustment, verify the properties of the a range object and the text it encompasses (via `rng.toString()`).

Related Items: `selectNode()`, `selectNodeContents()`, `setEnd()`, `setStart()` methods

surroundContents(*nodeReference*)

Returns: Nothing

Compatibility: WinIE-, MacIE-, NN7+, Moz+, Safari+, Opera+, Chrome+

The `surroundContents()` method surrounds the current range with a new parent element. Pass the new parent element as a parameter to the method. No document tree nodes or elements are removed or replaced in the process, but the current range becomes a child node of the new node; if the range coincides with an existing node, then the relationship between that node and its original parent becomes that of grandchild and grandparent. An application of this method may be to surround user-selected text with a `span` element whose class renders the content with a special font style or other display characteristic based on a style sheet selector for that class name.

When the element node being applied as the new parent has child nodes itself, those nodes are discarded before the element is applied to its new location. Therefore, for the most predictable results, using content-free element nodes as the parameter to the `surroundContents()` method is best.

Example

Listing 33-6 demonstrates how the `surroundContents()` method wraps a range inside a new element. As the page loads, a global variable (`newSpan`) stores a `span` element as a prototype for elements to be used as new surrounding parent nodes. When you select text in either of the two paragraphs, the selection is converted to a range. The `surroundContents()` method then wraps the range with the `newSpan` element. Because that `span` element has a class name of `hilite`, the element and its contents pick up the style sheet properties as defined for that class selector.

Note

Full support for this method didn't take place until Mozilla 1.6, which translates into Netscape 7.2 and Firefox 1.0 in terms of browser implementations. Prior to Mozilla 1.6, this method threw an exception when used with a range derived from a `selection` object, or when the range is not aligned with element boundaries. ■

LISTING 33-6

Using the Range.surroundContents() Method

```
<!DOCTYPE html>
<html>
   <head>
```

continued

LISTING 33-6 *(continued)*

```
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Range.surroundContents() Method</title>
      <style type="text/css">
         .hilite
         {
            background-color:yellow; color:red; font-weight:bold;
         }
      </style>
      <script type="text/javascript">
         var newSpan = document.createElement("span");
         newSpan.className = "hilite";

         function highlightSelection()
         {
            var sel, rng;
            try
            {
               sel = window.getSelection();
               if (!sel.isCollapsed)
               {
                  rng = sel.getRangeAt(0);
                  rng.surroundContents(newSpan.cloneNode(false));
               }
            }
            catch(err)
            {
               alert("Vital Range or Selection object services are not yet ↩
               implemented in this browser.");
            }
         }
      </script>
   </head>
   <body>
      <h1>Range.surroundContents() Method</h1>
      <hr />
      <p onmouseup="highlightSelection()">These paragraphs contain text that
         you can select. Selections are surrounded by span elements that share
         a style sheet class selector for special font and display
         characteristics.
      </p>
      <p onmouseup="highlightSelection()">Lorem ipsum dolor sit amet,
         consectetaur adipisicing elit, sed do eiusmod tempor incididunt ut
         labore et dolore magna aliqua. Ut enim adminim veniam, quis nostrud
         exercitation ullamco laboris nisi ut aliquip ex ea commodo
         consequat.
      </p>
   </body>
</html>
```

Related Items: `Range.insertNode()` method

toString()

Returns: String

Compatibility: WinIE-, MacIE-, NN6+, Moz+, Safari+, Opera+, Chrome+

Use the `toString()` method to retrieve a copy of the body text that is contained by the current text range. The text returned from this method is ignorant of any HTML tags or node boundaries that exist in the document tree. You also use this method to get the text of a user selection, after it has been converted to a text range.

Related Items: Selection.getRangeAt(), Range.extractContents() methods

selection Object

Compatibility: WinIE4+, MacIE-, NN6+, Moz+, Safari+, Opera+, Chrome+

Properties	Methods	Event Handlers
anchorNode	addRange()	
anchorOffset	clear()	
focusNode	collapse()	
focusOffset	collapseToEnd()	
isCollapsed	collapseToStart()	
rangeCount	containsNode()	
type	createRange()	
typeDetail	deleteFromDocument()	
	empty()	
	extend()	
	getRangeAt()	
	removeAllRanges()	
	removeRange()	
	selectAllChildren()	
	toString()	

Syntax

Accessing `selection` object properties or methods:

```
(IE4+)     [window.]document.selection.property | method()
(NN6+/Moz) window.selection.property | method()
```

About this object

The `selection` object provides scripted access to any body text or text in a form text control that is selected either by the user or by script. A `selection` object of one character or more is always highlighted on the page, and only one `selection` object can be active at any given instant.

Take advantage of the `selection` object when your page invites a user to select text for some operation that utilizes the selected text. The best event to use for working with a selection is the `onmouseup` event handler. This event fires on every release of the mouse, and your script can investigate the `document.selection` object to see if any text has been selected (using the selection's `type` property).

If you intend to perform some action on a selection, you may not be able to trigger that action by way of a button or link. In some browser versions and operating systems, clicking one of these elements automatically deselects the body selection.

One important difference between the IE and NN/Moz and WebKit- and Presto-based browsers selections is that the NN6+/Moz/WebKit/Presto `selection` object works only on body text, and not on selections inside text-oriented form controls. An NN6+/Moz/WebKit/Presto `selection` object has relationships with the document's node tree in that the object defines itself by the nodes (and offsets within those nodes) that encase the start and end points of a selection. When a user drags a selection, the node in which the selection starts is called the *anchor* node; the node holding the text at the point of the selection release is called the *focus* node; for double- or triple-clicked selections, the direction between anchor and focus nodes is in the direction of the language script (for example, left-to-right in Latin-based script families). In many ways, an NN6+/Moz `selection` object behaves just as the W3C DOM `Range` object, complete with methods to collapse and extend the selection. Unlike a range, however, the text encompassed by a `selection` object is highlighted on the page. If your scripts need to work with the nodes inside a selection, the `getRangeAt()` method of the `selection` object returns a range object whose boundary points coincide with the selection's boundary points.

Properties

`anchorNode`
`focusNode`

Value: Node reference — Read-Only

Compatibility: WinIE-, MacIE-, NN6+, Moz+, Safari+, Opera+, Chrome+

These two properties return a reference to the node where the user started (`anchorNode`) and ended (`focusNode`) the selection. If the selection is modified via the `addRange()` method, these properties point to the node boundaries of the most recently added range.

Related Items: `anchorOffset` and `focusOffset` properties

anchorOffset
focusOffset

Value: Integer Read-Only

Compatibility: WinIE-, MacIE-, NN6+, Moz+, Safari+, Opera+, Chrome+

These two properties return an integer count of the number of characters or nodes from the beginning of the selection to the anchor node (`anchorOffset`) and focus node (`focusOffset`). The count represents characters for text nodes, and nodes for element nodes. If the selection is modified via the `addRange()` method, these properties point to the node offsets of the most recently added range.

Related Items: `anchorNode` and `focusNode` properties

isCollapsed

Value: Boolean Read-Only

Compatibility: WinIE-, MacIE-, NN6+, Moz+, Safari+, Opera+, Chrome+

The `isCollapsed` property returns a Boolean value indicating whether or not the anchor and focus boundaries of the selection are the same. If they are the same (`true`), it means the selection has zero length between two characters (or before the first, or after the last character of the document).

Related Items: `anchorNode` and `focusNode` properties

rangeCount

Value: Integer Read-Only

Compatibility: WinIE-, MacIE-, NN6+, Moz+, Safari+, Opera+, Chrome+

The `rangeCount` property returns an integer count of the range objects contained within the selection. A manual user selection always results in a single range being selected, but the `addRange()` method can result in multiple ranges being contained by the selection.

Related Items: `getRangeAt()` method

type

Value: String Read-Only

Compatibility: WinIE4+, MacIE-, NN-, Moz-, Safari-, Opera+, Chrome-

The WinIE-originated `type` property returns `Text` whenever a selection exists on the page. Otherwise the property returns `None`. A script can use this information to determine if a selection is made on the page:

```
if (document.selection.type == "Text")
{
   // process selection
   // ...
}
```

selectionObject.**type**

Microsoft indicates that this property can sometimes return `Control`, but that terminology is associated with an edit mode outside the scope of this book.

Example

Listing 33-7 contains a page that demonstrates several features of the IE `selection` object. When you make a selection with the Deselect radio button selected, you see the value of the `selection.type` property (in the status bar) before and after the selection is deselected. After the selection goes away, the `type` property returns `None`. While Opera supports the `type` property, it does not support all of the methods that are used in this script.

LISTING 33-7

Using the document.selection Object

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>selection Object</title>
      <script type="text/javascript">
         function processSelection()
         {
            if (!document.selection)
            {
               alert("document.selection object is not supported by your browser");
            }
            else
            {
               if (document.choices.process[0].checked)
               {
                  status = "Selection is type: " + document.selection.type;
                  setTimeout("emptySelection()", 2000);
               }
               else if (document.choices.process[1].checked)
               {
                  var rng = document.selection.createRange();
                  document.selection.clear();
               }
            }
         }
         function emptySelection()
         {
            document.selection.empty();
            status = "Selection is type: " + document.selection.type;
         }
      </script>
   </head>
   <body>
      <h1>IE selection Object</h1>
      <h4>
```

```
        <dl>
          <dt>De-select Button</dt>
          <dd>Look at the status bar. Then, with the Deselect
            radio button selected, select some of the Latin text.
            Watch the status bar change after two seconds.
          </dd>
          <dt>Deleted Button</dt>
          <dd>The Latin text you select will be deleted.</dd>
      </h4>
      <hr />
      <form name="choices">
        <input type="radio" name="process" checked="checked" />
          De-select after two seconds<br />
        <input type="radio" name="process" />
          Delete selected text
      </form>
      <p onmouseup="processSelection()">Lorem ipsum dolor sit amet,
        consectetaur adipisicing elit, sed do eiusmod tempor incididunt ut
        labore et dolore magna aliqua. Ut enim adminim veniam, quis nostrud
        exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.
        Duis aute irure dolor in reprehenderit involuptate velit esse cillum
        dolore eu fugiat nulla pariatur.
     </p>
   </body>
</html>
```

Related Items: `TextRange.select()` method

typeDetail

Value: See text — Read-Only

Compatibility: WinIE5.5+, MacIE-, NN-, Moz-, Safari-, Opera-, Chrome-

The `typeDetail` property serves as a placeholder for applications that use the IE browser component, in which case the property can serve as a means of providing additional selection type information.

Related Items: `type` property

Methods

addRange(*rangeRef*)

Returns: Nothing

Compatibility: WinIE-, MacIE-, NN6+, Moz+, Safari+, Opera+, Chrome+

The `addRange()` method is used to highlight a selection on a page based upon a `Range` object. You can add multiple ranges to a selection by making repeated calls to the `addRange()` method.

Related Items: `removeRange()` method

clear()

Returns: Nothing

Compatibility: WinIE4+, MacIE-, NN-, Moz-, Safari-, Opera-, Chrome-

Use the `clear()` method to delete the current selection from the document. To the user, the `clear()` method has the same effect as setting the `TextRange.text` property to an empty string. The difference is that you can use the `clear()` method without having to generate a text range for the selection. After you delete a selection, the `selection.type` property returns `None`.

Example

See Listing 33-7, earlier in this chapter, to see the `selection.clear()` method at work.

Related Items: `selection.empty()` method

collapse(*nodeRef, offset*)

Returns: Nothing

Compatibility: WinIE-, MacIE-, NN6+, Moz+, Safari+, Opera+, Chrome+

The `collapse()` method collapses the current selection to a location specified by the two parameters, which consist of a node reference and an offset. The `nodeRef` parameter is a text or element node in the document tree to which the collapsed selection is to be moved, whereas the `offset` parameter is an integer count of characters or nodes within the target node where the collapsed selection should be moved.

Related Items: `collapseToEnd()` and `collapseToStart()` methods

collapseToEnd()
collapseToStart()

Returns: Nothing

Compatibility: WinIE-, MacIE-, NN6+, Moz+, Safari+, Opera+, Chrome+

These methods collapse the current selection to a location at the start (`collapseToStart()`) or end (`collapseToEnd()`) of the selection. After the collapse, any previously highlighted selection returns to normal display, and the selection contains only one range.

Related Items: `collapse()` method

containsNode(*nodeRef, entirelyFlag*)

Returns: Boolean

Compatibility: WinIE-, MacIE-, NN6+, Moz+, Safari+, Opera+, Chrome+

The `containsNode()` method returns a Boolean value indicating whether or not the specified node is contained in the selection. The `nodeRef` parameter is the node you are checking for selection containment, whereas the `entirelyFlag` parameter specifies whether or not the node must be contained in its entirety (passing `null` for the `entirelyFlag` parameter is usually sufficient).

`createRange()`

Returns: `TextRange` object

Compatibility: WinIE4+, MacIE-, NN-, Moz-, Safari-, Opera+, Chrome-

To generate a text range for a user selection in WinIE, invoke the `createRange()` method of the `selection` object. We're not sure why the method for the `selection` object is called `createRange()`, whereas text ranges for other valid objects are created with a `createTextRange()` method. The result of both methods is a full-fledged `TextRange` object.

Example

Refer to Listing 26-36 to see the `selection.createRange()` method turn user selections into text ranges.

Related Items: `TextRange` object

`deleteFromDocument()`

Returns: Nothing

Compatibility: WinIE-, MacIE-, NN6+, Moz+, Safari+, Opera+, Chrome+

The `deleteFromDocument()` method deletes the current selection from the document tree.

Related Items: `removeRange()` method

`empty()`

Returns: Nothing

Compatibility: WinIE4+, MacIE-, NN-, Moz-, Safari-, Opera-, Chrome-

The `empty()` method deselects the current WinIE selection. After deselection, the `selection.type` property returns `None`. The action of the `empty()` method is the same as the `UnSelect` command invoked via the `execCommand()` method for a document. If the selection was made from a `TextRange` object (via the `TextRange.select()` method), the `empty()` method affects only the visible selection and not the text range.

Example

See Listing 33-7, earlier in this chapter, to view the `selection.empty()` method at work.

Related Items: `selection.clear()` method

`extend(`*`nodeRef, offset`*`)`

Returns: Nothing

Compatibility: WinIE-, MacIE-, NN6+, Moz+, Safari+, Opera+, Chrome+

The `extend()` method extends the boundary of the selection to the specified node (`nodeRef`) and offset (`offset`) within that node. The start of the boundary (anchor node) remains unaffected by this method; only the end of the boundary (focus node) is altered.

getRangeAt(*rangeIndex*)

Returns: Range object

Compatibility: WinIE-, MacIE-, NN6+, Moz+, Safari+, Opera+, Chrome+

The getRangeAt() method obtains the range at the specified zero-based index (rangeIndex) within the selection. You can use the rangeCount property to determine how many ranges are contained within the selection.

Related Items: rangeCount property

removeAllRanges()

Returns: Nothing

Compatibility: WinIE-, MacIE-, NN6+, Moz+, Safari+, Opera+, Chrome+

The removeAllRanges() method empties the selection by removing all of the ranges from it. Upon calling this method, the selection collapses and the rangeCount property goes to zero; the document tree remains unaffected.

Related Items: removeRange() method

removeRange(*rangeRef*)

Returns: Nothing

Compatibility: WinIE-, MacIE-, NN6+, Moz+, Safari+, Opera+, Chrome+

The removeRange() method removes the specified range (rangeRef) from the selection, but not from the document tree.

Related Items: removeAllRanges() method

selectAllChildren(*elementNodeRef*)

Returns: Nothing

Compatibility: WinIE-, MacIE-, NN6+, Moz+, Safari+, Opera+, Chrome

The selectAllChildren() method forces the selection to encompass the specified node (elementNodeRef) and all of its children. Calling this method on an element node results in the anchor and focus nodes being set to that node.

toString()

Returns: String

Compatibility: WinIE-, MacIE-, NN6+, Moz+, Safari+, Opera+, Chrome+

The toString() method returns a string representation of the selection, which is the body content from the selection, minus tags and attributes.

Text and TextNode Objects

Compatibility: WinIE5+, MacIE5+, NN6+, Moz+, Safari+, Opera+, Chrome+

Properties	Methods	Event Handlers
attributes†	appendChild()†	
childNodes†	appendData()	
data	cloneNode()†	
firstChild†	deleteData()	
lastChild†	hasChildNodes()†	
length†	insertBefore()†	
localName†	insertData()	
namespaceURI†	normalize()†	
nextSibling†	removeChild()†	
nodeName†	replaceChild()†	
nodeType†	replaceData()	
nodeValue†	splitText()	
ownerDocument†	substringData()	
parentNode†		
prefix†		
previousSibling†		

†See Chapter 26, ''Generic HTML Element Objects.''

Syntax

Accessing `Text` and `TextNode` object properties or methods:

```
(IE5+/W3C) [window.]document.getElementById("id").textNodeRef.property |
           method()
```

About this object

Discussing the `Text` object of the W3C DOM (as implemented in NN6+/Moz and WebKit- and Presto-based browsers) in the same breath as the IE5+ `TextNode` object is a little tricky. Conceptually, they are the same kind of object in that they are the document tree objects — text nodes — that contain an HTML element's text (see Chapter 25, "Document Object Model Essentials," for details on the role of the text node in the `document` object hierarchy). Generating a new text node by script is achieved the same way in both object

models: `document.createTextNode()`. What makes the discussion of the two objects tricky is that although the W3C DOM version comes from a strictly object-oriented specification (in which a text node is an instance of a `CharacterData` object, which, in turn is an instance of the generic `Node` object), the IE object model is not quite as complete. For example, whereas the W3C DOM `Text` object inherits all of the properties and methods of the `CharacterData` and `Node` definitions, the IE `TextNode` object exposes only those properties and methods that Microsoft deems appropriate.

No discrepancy in terminology gets in the way when calling these objects because their object names never become part of the script. Instead, script statements always refer to text nodes by other means, such as through a child node–related property of an element object, or as a variable that receives the result of the `document.createTextNode()` method.

Although both objects share a number of properties and one method, the W3C DOM `Text` object contains a few methods that have "data" in their names. These properties and methods are inherited from the `CharacterData` object in the DOM specification. They are discussed as a group in the section about object methods in this chapter. In all cases, check the browser version support for each property and method described here.

Properties

`data`

Value: String — Read/Write

Compatibility: WinIE5+, MacIE5+, NN6+, Moz+, Safari+, Opera+, Chrome+

The `data` property contains the string comprising the text node. Its value is identical to the `nodeValue` property of a text node. See the description of the `nodeValue` property in Chapter 26.

Example

In the example for the `nodeValue` property used in a text replacement script (Listing 25-2), you can substitute the `data` property for `nodeValue` to accomplish the same result.

Related Items: `nodeValue` property of all element objects (Chapter 26)

Methods

`appendData("`*`text`*`")`
`deleteData(`*`offset, count`*`)`
`insertData(`*`offset, "text"`*`)`
`replaceData(`*`offset, count, "text"`*`)`
`substringData(`*`offset, count`*`)`

Returns: See text

Compatibility: WinIE6+, MacIE5+, NN6+, Moz+, Safari+, Opera+, Chrome+

These five methods of the W3C DOM `Text` object provide scripted manipulation of the text inside a text node. Methods that modify the node's data automatically change the values of both the `data` and `nodeValue` properties.

The purposes of these methods are obvious, for the most part. Any method that requires an `offset` parameter uses this integer value to indicate where in the existing text node the deletion, insertion, or replacement starts. Offsets are zero-based, meaning that, in order to indicate that the action should take place starting with the first character, specify a zero for the parameter. A `count` parameter is another integer, but one that indicates how many characters are to be included. For example, consider a text node that contains the following data:

```
abcdefgh
```

This node could be a node of an element on the page, or a node that has been created and assigned to a variable but not yet inserted into the page. To delete the first three characters of that text node, the statement is

```
textNodeReference.deleteData(0,3)
```

This leaves the text node content as

```
defgh
```

As for the `replaceData()` method, the length of the text being put in place of the original chunk of text need not match the `count` parameter. The `count` parameter, in concert with the `offset` parameter, defines what text is to be removed and replaced by the new text.

The `substringData()` method is similar to the JavaScript core language `String.substr()` method, in that both require parameters indicating the offset within the string to start reading and the number of characters. You get the same result with the `substringData()` method of a text node as you do from a `nodeValue.substr()` method when both are invoked from a valid text node object.

Of all five methods discussed here, only `substringData()` returns a value: a string.

Example

The page created by Listing 33-8 is a working laboratory that you can use to experiment with the five data-related methods in IE6+/NN6+/Moz and WebKi- and Presto-based browsers. The text node that invokes the methods is a simple sentence in a `p` element. Each method has its own clickable button, followed by two or three text boxes into which you enter values for method parameters. Don't be put off by the length of the listing. Each method's operation is confined to its own function, and is fairly simple.

Each of the data-related methods throws exceptions of different kinds. To help handle these errors gracefully, the method calls are wrapped inside a `try/catch` construction. All caught exceptions are routed to the `handleError()` function, where details of the error are inspected,

and friendly alert messages are displayed to the user. See Chapter 21, "Control Structures and Exception Handling," for details on the `try/catch` approach to error handling in W3C DOM-capable browsers.

LISTING 33-8

Text object Data Method Laboratory

HTML: jsb33-08.html

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Data Methods of a W3C Text Object</title>
      <script type="text/javascript" src="jsb33-08.js"></script>
   </head>
   <body>
      <h1>Data Methods of a W3C Text Object</h1>
      <hr />
      <p id="myP" style="font-weight:bold; text-align:center">
         So I called myself Pip, and became to be called Pip.</p>
      <form name="choices">
         <p><input type="button" onclick="doAppend(this.form)"
                   value="appendData()" />
            String:<input type="text" name="appendStr" size="30" />
         </p>
         <p><input type="button" onclick="doDelete(this.form)"
                   value="deleteData()" />
            Offset:<input type="text"
                          name="deleteOffset" size="3" />
            Count:<input type="text"
                         name="deleteCount" size="3" />
         </p>
         <p><input type="button" onclick="doInsert(this.form)"
                   value="insertData()" />
            Offset:<input type="text"
                          name="insertOffset" size="3" />
            String:<input type="text"
                          name="insertStr" size="30" />
         </p>
         <p><input type="button" onclick="doReplace(this.form)"
                   value="replaceData()" />
            Offset:<input type="text"
                          name="replaceOffset" size="3" />
            Count:<input type="text"
                         name="replaceCount" size="3" />
            String:<input type="text"
                          name="replaceStr" size="30" />
         </p>
         <p><input type="button" onclick="showSubstring(this.form)"
                   value="substringData()" />
            Offset:<input type="text"
```

```
                          name="substrOffset" size="3" />
            Count:<input type="text"
                         name="substrCount" size="3" />
         </p>
      </form>
   </body>
</html>
```

JavaScript: jsb33-08.js

```
function doAppend(form)
{
   var node = document.getElementById("myP").firstChild;
   var newString = form.appendStr.value;
   try
   {
      node.appendData(newString);
   }
   catch(err)
   {
      handleError(err);
   }
}

function doDelete(form)
{
   var node = document.getElementById("myP").firstChild;
   var offset = form.deleteOffset.value;
   var count = form.deleteCount.value;
   try
   {
      node.deleteData(offset, count);
   }
   catch(err)
   {
      handleError(err);
   }
}

function doInsert(form)
{
   var node = document.getElementById("myP").firstChild;
   var offset = form.insertOffset.value;
   var newString = form.insertStr.value;
   try
   {
      node.insertData(offset, newString);
   }
   catch(err)
   {
      handleError(err);
   }
```

continued

LISTING 33-8 *(continued)*

```
}

function doReplace(form)
{
   var node = document.getElementById("myP").firstChild;
   var offset = form.replaceOffset.value;
   var count = form.replaceCount.value;
   var newString = form.replaceStr.value;
   try
   {
      node.replaceData(offset, count, newString);
   }
   catch(err)
   {
      handleError(err);
   }
}

function showSubstring(form)
{
   var node = document.getElementById("myP").firstChild;
   var offset = form.substrOffset.value;
   var count = form.substrCount.value;
   try
   {
      alert(node.substringData(offset, count));
   }
   catch(err)
   {
      handleError(err);
   }
}

// error handler for these methods
function handleError(err)
{
   switch (err.name)
   {
   case "NS_ERROR_DOM_INDEX_SIZE_ERR":
      alert("The offset number is outside the allowable range.");
      break;
   case "NS_ERROR_DOM_NOT_NUMBER_ERR":
      alert("Make sure each numeric entry is a valid number.");
      break;
   default:
      alert("Double-check your text box entries.");
   }
}
```

Related Items: `appendChild()`, `removeChild()`, `replaceChild()` methods of element objects (Chapter 26, "Generic HTML Element Objects")

splitText(*offset*)

Returns: `Text` or `TextNode` object

Compatibility: WinIE6+, MacIE5+, NN6+, Moz+, Safari+, Opera+, Chrome+

The `splitText()` method performs multiple actions with one blow. The `offset` parameter is an integer indicating the zero-based index position within the text node at which the node is to divide into two nodes. After you invoke the method on the current text node, the node consists of the text from the beginning of the node, up to the offset position. The method returns a reference to the text node whose data starts with the character after the dividing point, and extends to the end of the original node. Users won't notice any change in the rendered text: This method influences only the text node structure of the document. Using this method means, for example, that an HTML element that starts with only one text node will have two after the `splitText()` method is invoked. The opposite action (combining contiguous text node objects into a single node) is performed by the IE7+/NN6+/Moz/Safari1.2+ `normalize()` method (see Chapter 26).

Note

Although IE6 and MacIE5 both support the `normalize()` method, IE6 in particular has a buggy implementation that we wouldn't rely on. IE7 appears to have remedied this problem. ■

Example

Use The Evaluator (Chapter 4, "JavaScript Essentials") to see the `splitText()` method in action. Begin by verifying that the `myEM` element has but one child node, and that its `nodeValue` is the string "all":

```
document.getElementById("myEM").childNodes.length
document.getElementById("myEM").firstChild.nodeValue
```

Next, split the text node into two pieces after the first character:

```
document.getElementById("myEM").firstChild.splitText(1)
```

Two text nodes are now inside the element:

```
document.getElementById("myEM").childNodes.length
```

Each text node contains its respective portion of the original text:

```
document.getElementById("myEM").firstChild.nodeValue
document.getElementById("myEM").lastChild.nodeValue
```

If you are using IE7+/NN6+/Moz and WebKit- and Presto-based browsers, now bring the text nodes back together:

```
document.getElementById("myEM").normalize()
document.getElementById("myEM").childNodes.length
```

At no time during these statement executions does the rendered text change.

Related Items: `normalize()` method (Chapter 26, "Generic HTML Element Objects")

TextRange Object

Compatibility: WinIE4+, MacIE-, NN-, Moz-, Safari-, Opera+, Chrome-

Properties	Methods	Event Handlers
boundingHeight	collapse()	
boundingLeft	compareEndPoints()	
boundingTop	duplicate()	
boundingWidth	execCommand()	
htmlText	expand()	
offsetLeft†	findText()	
offsetTop†	getBookmark()	
text	getBoundingClientRect()†	
	getClientRects()†	
	inRange()	
	isEqual()	
	move()	
	moveEnd()	
	moveStart()	
	moveToBookmark()	
	moveToElementText()	
	moveToPoint()	
	parentElement()	
	pasteHTML()	
	queryCommandEnabled()	
	queryCommandIndeterm()	
	queryCommandState()	

Properties	Methods	Event Handlers
	`queryCommandSupported()`	
	`queryCommandText()`	
	`queryCommandValue()`	
	`scrollIntoView()`†	
	`select()`	
	`setEndPoint()`	

†See Chapter 26, ''Generic HTML Element Objects.''

Syntax

Creating a `TextRange` object:

```
var rangeRef = document.body.createTextRange();
var rangeRef = buttonControlRef.createTextRange();
var rangeRef = textControlRef.createTextRange();
var rangeRef = document.selection.createRange();
```

Accessing `TextRange` object properties or methods:

```
(IE4+)        rangeRef.property | method([parameters])
```

About this object

Unlike most of the objects covered in Part III of this book, the IE4+ `TextRange` object is not tied to a specific HTML element. The `TextRange` object is, instead, an abstract object that represents text content anywhere on the page (including content of a text-oriented form control) between a start point and an end point (collectively, the *boundaries* of the range). The user may not necessarily know that a `TextRange` object exists, because no requirement exists to force a `TextRange` object to physically select text on the page (although the `TextRange` object can be used to assist scripts in automating the selection of text, or a script may turn a user selection into a `TextRange` object for further processing).

The purpose of the `TextRange` object is to give scripts the power to examine, modify, remove, replace, and insert content on the page. Start and end points of an IE `TextRange` object are defined exclusively in terms of character positions within the element that is used to create the range (usually the `body` element, but also button- and text-related form control elements). Character positions of body text do not take into account source code characters that may define HTML elements. This factor is what distinguishes a `TextRange`'s behavior from, for instance, the various properties and methods of HTML elements that let you modify or copy elements and their text (for example, `innerText` and `outerText` properties). A `TextRange` object's start point can be in one element, and its end point in another. For example, consider the following HTML:

```
<p>And now to introduce our <em>very special</em> guest:</p>
```

If the text shown in boldface indicates the content of a `TextRange` object, you can see that the range crosses element boundaries in a way that makes HTML element object properties difficult to use for replacing that range with some other text. Challenges still remain in this example, however. Simply replacing the text of the range with some other text forces your script (or the browser) to reconcile the issue of what to do about the nested `em` element, because the `TextRange` object handles only its text. (Your word processing program must address the same kind of issue when you select a phrase that starts in italic but ends in normal font, and then you paste text into that selection.)

An important aspect of the `TextRange` object is that the size of the range can be zero or more characters. Start and end points always position themselves between characters. When the start point and end point of a range are at the same location, the range acts as a text insertion pointer. In fact, when the `TextRange` object represents text inside a text-related form control, the `select()` method of the `TextRange` object can be used to display the text insertion pointer where your script desires. Therefore, through the `TextRange` object you can script your forms to always display the text insertion pointer at the end of existing text in a text box or textarea when the control receives focus.

Working with text ranges

To create a `TextRange` object, use the `createTextRange()` method with the `document.body` object or any button- or text-related form control object. If you want to convert a block of selected text to a text range, use the special `createRange()` method of the `document.selection` object. Regardless of how you create it, the range encompasses the entire text of the object used to generate the range. In other words, the start point is at the very beginning of the text, and the end point is at the very end. Note that when you create a `TextRange` object from the `body` element, text that is inside text-related form controls is not part of the text of the `TextRange` (just as text field content isn't selected if you manually select the entire text of the page).

After you create a `TextRange` object (assigned to a variable), the typical next steps involve some of the many methods associated with the object that help narrow the size of the range. Some methods (`move()`, `moveEnd()`, `moveStart()`, and `sentEndPoint()`) offer manual control over the intra-character position for the start and end points. Parameters of some of these methods understand concepts, such as words and sentences, so not every action entails tedious character counts. Another method, `moveToElementText()`, automatically adjusts the range to encompass a named element. The oft-used `collapse()` method brings the start and end points together at the beginning or end of the current range — helpful when a script must iterate through a range for tasks, such as word counting or search and replace. The `expand()` method can extend a collapsed range to encompass the whole word, whole sentence, or entire range surrounding the insertion point. Perhaps the most powerful method is `findText()`, which allows scripts to perform single or global search-and-replace operations on body text.

After the range encompasses the desired text, several other methods let scripts act on the selection. The types of operations include scrolling the page to make the text represented by the range visible to the user (`scrollIntoView()`), and selecting the text (`select()`) to provide

visual feedback to the user that something is happening (or to set the insertion pointer at a location in a text form control). An entire library of additional commands are accessed through the `execCommand()` method for operations, such as copying text to the clipboard, and a host of formatting commands that can be used in place of style sheet property changes. To swap text from the range with new text accumulated by your script, you can modify the `text` property of the range.

Using the `TextRange` object can be a bit tedious, because it often requires a number of script statements to execute an action. Three basic steps are generally required to work with a `TextRange` object:

1. Create the text range.
2. Set the start and end points.
3. Act on the range.

As soon as you are comfortable with this object, you will find it provides a lot of flexibility in scripting interaction with body content. For ideas about applying the `TextRange` object in your scripts, see the examples that accompany the following descriptions of individual properties and methods.

About browser compatibility

The `TextRange` object is available only for 32-bit versions of IE4 and later. There is no support for the `TextRange` object in MacIE.

The W3C DOM and NN6+ implement a slightly different concept of text ranges in what they call the `Range` object. In many respects, the fundamental way of working with a `Range` object is the same as for a `TextRange` object: create and adjust start and end points, and act on the range. But the W3C version (like the W3C DOM itself) is more conscious of the node hierarchy of a document. Properties and methods of the W3C `Range` object reflect this node-centric point of view, so that most of the terminology for the `Range` object differs from that of the IE `TextRange` object. As of this writing, IE8 doesn't support the W3C `Range` object, and it is unknown if or when Microsoft will add support for it.

At the same time, the W3C `Range` object lacks a couple of methods that are quite useful with the IE `TextRange` object — notably `findText()` and `select()`. On the other hand, the `Range` object, as implemented in NN6+/Moz and WebKit- and Presto-based browsers, works on all OS platforms.

The bottom-line question, then, is whether you can make range-related scripts work in both browsers. Whereas the basic sequence of operations is the same for both objects, the scripting vocabulary is quite different. Table 33-1 presents a summary of the property and method behaviors that the two objects have in common, and their respective vocabulary terms. (Sometimes the value of a property in one object is accessed via a method in the other object.) Notice that the ways of moving individual end points are not listed in the table because the corresponding

methods for each object (for example, `moveStart()` in `TextRange` versus `setStart()` in `Range`) use very different spatial paradigms.

TABLE 33-1

TextRange versus Range Common Denominators

TextRange Object	Range Object
`text`	`toString()`
`collapse()`	`collapse()`
`compareEndPoints()`	`compareEndPoints()`
`duplicate()`	`clone()`
`moveToElementText()`	`selectContents()`
`parentElement()`	`commonParent`

To blend text range actions for both object models into a single scripted page, you have to include script execution branches for each category of object model, or create your own API to invoke library functions that perform the branching. On the IE side of things, too, you have to script around actions that can cause script errors when run on MacOS and other non-Windows versions of the browser.

Properties

boundingHeight
boundingLeft
boundingTop
boundingWidth

Value: Integer Read-Only

Compatibility: WinIE4+, MacIE-, NN-, Moz-, Safari-, Opera-, Chrome-

Every text range has physical dimension and location on the page, even if you cannot see the range reflected graphically with highlighting. Even a text insertion pointer (meaning a collapsed text range) has a rectangle whose height equals the line height of the body text in which the insertion point resides; its width, however, is zero.

The pixel dimensions of the rectangle of a text range can be retrieved via the `boundingHeight` and `boundingWidth` properties of the `TextRange` object. When a text range extends across multiple lines, the dimensions of the rectangle are equal to the smallest single rectangle that can contain the text (a concept identical to the bounding rectangle of inline body text, as described in the `TextRectangle` object later in this chapter). Therefore, even a range consisting of one character at the end of one line and another character at the beginning of the next, forces the bounding rectangle to be as wide as the paragraph element.

A text range rectangle has a physical location on the page. The top-left position of the rectangle (with respect to the browser window edge) is reported by the `boundingTop` and `boundingLeft` properties.

Example

Listing 33-9 provides a simple playground to explore the four bounding properties (and two offset properties) of a `TextRange` object. As you select text in the big paragraph, the values of all six properties are displayed in the table. Values are also updated if you resize the window via an `onresize` event handler.

Notice, for example, that if you simply click in the paragraph without dragging a selection, the `boundingWidth` property shows up as zero. This action is the equivalent of a `TextRange` acting as an insertion point.

LISTING 33-9

Exploring the Bounding TextRange Properties

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>TextRange Object Dimension Properties</title>
      <style type="text/css">
         td
         {
            text-align:center;
         }
         .propName
         {
            font-family: Courier, monospace;
         }
      </style>
      <script type="text/javascript">
         function replaceHTML(elem, text)
         {
            while(elem.firstChild)
            {
               elem.removeChild(elem.firstChild);
            }
            elem.appendChild(document.createTextNode(text));
         }

         function setAndShowRangeData()
         {
            var range = document.selection.createRange();
            replaceHTML(document.getElementById("B1"), range.boundingHeight);
            replaceHTML(document.getElementById("B2"), range.boundingWidth);
```

continued

LISTING 33-9 *(continued)*

```
            replaceHTML(document.getElementById("B3"), range.boundingTop);
            replaceHTML(document.getElementById("B4"), range.boundingLeft);
            replaceHTML(document.getElementById("B5"), range.offsetTop);
            replaceHTML(document.getElementById("B6"), range.offsetLeft);
         }
      </script>
   </head>
   <body onresize="setAndShowRangeData()">
      <h1>TextRange Object Dimension Properties</h1>
      <hr />
      <p>Select text in the paragraph below and observe the "bounding" property
         values for the TextRange object created for that selection.</p>
      <table id="results" border="1" cellspacing="2" cellpadding="2">
         <tr>
            <th>Property</th>
            <th>Pixel Value</th>
         </tr>
         <tr>
            <td class="propName">boundingHeight</td>
            <td class="count" id="B1"> </td>
         </tr>
         <tr>
            <td class="propName">boundingWidth</td>
            <td class="count" id="B2"> </td>
         </tr>
         <tr>
            <td class="propName">boundingTop</td>
            <td class="count" id="B3"> </td>
         </tr>
         <tr>
            <td class="propName">boundingLeft</td>
            <td class="count" id="B4"> </td>
         </tr>
         <tr>
            <td class="propName">offsetTop</td>
            <td class="count" id="B5"> </td>
         </tr>
         <tr>
            <td class="propName">offsetLeft</td>
            <td class="count" id="B6"> </td>
         </tr>
      </table>
      <hr />
      <p onmouseup="setAndShowRangeData()">Lorem ipsum dolor sit amet,
         consectetaur adipisicing elit, sed do eiusmod tempor incididunt
         ut labore et dolore magna aliqua. Ut enim adminim veniam, quis
         nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo
         consequat. Duis aute irure dolor in reprehenderit involuptate velit
```

```
        esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat
        cupidatat non proident, sunt in culpa qui officia deseruntmollit
        anim id est laborum Et harumd und lookum like Greek to me, dereud
        facilis est er expedit.</p>
    </body>
</html>
```

Related Items: `offsetLeft`, `offsetTop` properties of element objects (Chapter 26, "Generic HTML Element Objects")

htmlText

Value: String Read-Only

Compatibility: WinIE4+, MacIE-, NN-, Moz-, Safari-, Opera-, Chrome-

The `htmlText` property returns the HTML of the text contained by a text range. If a range's start and end points are at the very edges of an element's text, the HTML tag for that element becomes part of the `htmlText` property value. Also, if the range starts in one element and ends partway in another, the tags that influence the text of the end portion become part of the `htmlText`. This property is read-only, so you cannot use it to insert or replace HTML in the text range (see the `pasteHTML()` method, and various insert commands associated with the `execCommand()` method, in the following section).

Example

Use The Evaluator (Chapter 4, "JavaScript Essentials") to investigate values returned by the `htmlText` property. Use the top text box to enter the following statements, and see the values in the Results box.

Begin by creating a `TextRange` object for the entire body and store the range in local variable `a`:

```
a = document.body.createTextRange()
```

Next, use the `findText()` method to set the start and end points of the text range around the word "all," which is an `em` element inside the `myP` paragraph:

```
a.findText("all")
```

The method returns `true` (see the `findText()` method) if the text is found, and the text range adjusts to surround it. To prove that the text of the text range is what you think it is, examine the `text` property of the range:

```
a.text
```

Because the text range encompasses all of the text of the element, the `htmlText` property contains the tags for the element as well:

```
a.htmlText
```

If you want to experiment by finding other chunks of text and looking at both the `text` and `htmlText` properties, first restore the text range to encompass the entire body with the following statement:

```
a.expand("textEdit")
```

You can read about the `expand()` method later in this chapter. In other tests, use `findText()` to set the range to "for all" and just "for al." Then, see how the `htmlText` property exposes the `em` element's tags.

Related Items: `text` property

text

Value: String Read/Write

Compatibility: WinIE4+, MacIE-, NN-, Moz-, Safari-, Opera+, Chrome-

Use the `text` property to view or change the string of visible characters that comprise a text range. The browser makes some decisions for you if the range you are about to change has nested elements inside. By and large, the nested element (and any formatting that may be associated with it) is deleted, and the new text becomes part of the text of the container that houses the start point of the text range. By the same token, if the range starts in the middle of one element and ends in the parent element's text, the tag that governs the start point now wraps all of the new text.

Example

See Listing 33-11, later in this chapter, for the `findText()` method where the `text` property is used to perform the replace action of a search-and-replace function.

Related Items: `htmlText` property

Methods

collapse([*startBoolean*])

Returns: Nothing

Compatibility: WinIE4+, MacIE-, NN-, Moz-, Safari-, Opera+, Chrome-

Use the `collapse()` method to shrink a text range from its current size down to a single insertion point between characters. This method becomes more important than you may think at first, especially in a function that is traversing the body of the page or a large chunk of text. For example, in a typical looping word-counting script, you create a text range that encompasses the full body of the page (or text in a `textarea`). When the range is created, its start point is at the beginning of the text, and its end point is at the very end. To begin counting words, you can first collapse the range to the insertion point at the beginning of the range. Next, use the `expand()` method to set the range to the first word of text (and increment the counter if the `expand()` method returns `true`). At that point, the text range extends around the first word.

What you want is for the range to collapse at the end of the current range so that the search for the next word starts after the current one. Use `collapse()` once more, but this time include parameters.

The optional parameter of the `collapse()` method is a Boolean value that directs the range to collapse itself at either the start or end of the current range. The default behavior is the equivalent of a value of `true`, which means that unless otherwise directed, a `collapse()` method shifts the text range to the point in front of the current range. That works great as an early step in the word-counting example, because you want the text range to collapse to the start of the text before doing any counting. But for subsequent movements through the range, you want to collapse the range so that it is after the current range. Thus, you include a `false` parameter to the `collapse()` method.

Example

Refer to Listings 33-11 and 26-14 to see the `collapse()` method at work.

Related Items: `Range.collapse()`, `TextRange.expand()` methods

compareEndPoints("*type*", *rangeRef*)

Returns: Integer (`-1`, `0`, or `1`)

Compatibility: WinIE4+, MacIE-, NN-, Moz-, Safari-, Opera-, Chrome-

Generating multiple `TextRange` objects and assigning them to different variables is no problem. You can then use the `compareEndPoints()` method to compare the relative positions of start and end points of two ranges. One range is the object that you use to invoke the `compareEndPoints()` method, and the other range is the second parameter of the method. The order doesn't matter, because the first parameter of the method determines which points in each range you will be comparing.

Values for the first parameter can be one of four explicit strings: `StartToEnd`, `StartToStart`, `EndToStart`, and `EndToEnd`. What these values specify is which point of the current range is compared with which point of the range passed as the second parameter. For example, consider the following body text that has two text ranges defined within it:

It was the **best of *times*.**

The first text range (assigned in our discussion here to variable `rng1`) is shown in boldface, whereas the second text range (`rng2`) is shown in bold-italic. In other words, `rng2` is nested inside `rng1`. We can compare the position of the start of `rng1` against the position of the start of `rng2` by using the `StartToStart` parameter of the `compareEndPoints()` method:

```
var result = rng1.compareEndPoints("StartToStart", rng2);
```

The value returned from the `compareEndPoints()` method is an integer of one of three values. If the positions of both points under test are the same, the value returned is `0`. If the first point is before the second, the value returned is `-1`; if the first point is after the second, the value is `1`. Therefore, from the previous example, because the start of `rng1` is before the start

of rng2, the method returns -1. If you changed the statement to invoke the method on rng2, as in

```
var result = rng2.compareEndPoints("StartToStart", rng1);
```

the result would be 1.

In practice, this method is helpful in knowing if two ranges are the same, if one of the boundary points of both ranges is the same, or if one range starts where the other ends.

Example

The page rendered by Listing 33-10 lets you experiment with text range comparisons. The bottom paragraph contains a span element that has a TextRange object assigned to its text after the page loads (in the init() function). That fixed range becomes a solid reference point for you to use while you select text in the paragraph. After you make a selection, all four versions of the compareEndPoints() method run to compare the start and end points of the fixed range against your selection. One column of the results table shows the raw value returned by the compareEndPoints() method, whereas the third column puts the results into plain language.

To see how this page works, begin by selecting the first word of the fixed text range (double-click the word). You can see that the starting positions of both ranges are the same, because the returned value is 0. Because all of the invocations of the compareEndPoints() method are on the fixed text range, all comparisons are from the point of view of that range. Thus, the first row of the table for the StartToEnd parameter indicates that the start point of the fixed range comes before the end point of the selection, yielding a return value of -1.

Other selections to make include:

- Text that starts before the fixed range and ends inside the range
- Text that starts inside the fixed range and ends beyond the range
- Text that starts and ends precisely at the fixed range boundaries
- Text that starts and ends before the fixed range
- Text that starts after the fixed range

Study the returned values and the plain language results and see how they align with the selection you make.

LISTING 33-10

Lab for compareEndPoints() Method

HTML: jsb33-10.html

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
```

```
      <title>TextRange.compareEndPoints() Method</title>
      <style type="text/css">
         td
         {
            text-align:center;
         }
         .propName
         {
            font-family:Courier, monospace;
         }
         #fixedRangeElem
         {
            color:red; font-weight:bold;
         }
      </style>
      <script type="text/javascript" src="jsb33-10.js"></script>
   </head>
   <body onload="init()">
      <h1>TextRange.compareEndPoints() Method</h1>
      <hr />
      <p>Select text in the paragraph in various places relative to the fixed
         text range (shown in red). See the relations between the fixed and
         selected ranges with respect to their start and end points.</p>
      <table id="results" border="1" cellspacing="2" cellpadding="2">
         <tr>
            <th>Property</th>
            <th>Returned Value</th>
            <th>Fixed Range vs. Selection</th>
         </tr>
         <tr>
            <td class="propName">StartToEnd</td>
            <td class="count" id="B1"> </td>
            <td class="count" id="C1">Start of
               Fixed <span id="compare1">vs.</span> End
               of Selection
            </td>
         </tr>
         <tr>
            <td class="propName">StartToStart</td>
            <td class="count" id="B2"> </td>
            <td class="count" id="C2">Start of
               Fixed <span id="compare2">vs.</span> Start of
               Selection
            </td>
         </tr>
         <tr>
            <td class="propName">EndToStart</td>
            <td class="count" id="B3"> </td>
            <td class="count" id="C3">End of
               Fixed <span id="compare3">vs.</span> Start of
               Selection
            </td>
         </tr>
```

continued

LISTING 33-10 *(continued)*

```
        <tr>
          <td class="propName">EndToEnd</td>
          <td class="count" id="B4"> </td>
          <td class="count" id="C4">End of
            Fixed <span id="compare4">vs.</span> End of
            Selection
          </td>
        </tr>
      </table>
      <hr />
      <p onmouseup="setAndShowRangeData()">Lorem ipsum dolor
        sit, <span id="fixedRangeElem">consectetaur
        adipisicing elit</span>, sed do eiusmod tempor incididunt ut labore et
        dolore aliqua. Ut enim adminim veniam, quis nostrud exercitation ullamco
        laboris nisi ut aliquip ex ea commodo consequat.</p>
   </body>
</html>
```

JavaScript: jsb33-10.js

```
var fixedRange;

function replaceHTML(elem, text)
{
   while(elem.firstChild)
   {
      elem.removeChild(elem.firstChild);
   }
   elem.appendChild(document.createTextNode(text));
}

function setAndShowRangeData()
{
   var selectedRange = document.selection.createRange();
   var result1 = fixedRange.compareEndPoints("StartToEnd", selectedRange);
   var result2 = fixedRange.compareEndPoints("StartToStart", selectedRange);
   var result3 = fixedRange.compareEndPoints("EndToStart", selectedRange);
   var result4 = fixedRange.compareEndPoints("EndToEnd", selectedRange);

   replaceHTML(document.getElementById("B1"), result1);
   replaceHTML(document.getElementById("compare1"), getDescription(result1));
   replaceHTML(document.getElementById("B2"), result2);
   replaceHTML(document.getElementById("compare2"), getDescription(result2));
   replaceHTML(document.getElementById("B3"), result3);
   replaceHTML(document.getElementById("compare3"), getDescription(result3));
   replaceHTML(document.getElementById("B4"), result4);
   replaceHTML(document.getElementById("compare4"), getDescription(result4));
}
```

```
function getDescription(comparisonValue)
{
   switch (comparisonValue)
   {
   case -1 :
      return "comes before";
      break;
   case 0 :
      return "is the same as";
      break;
   case 1 :
      return "comes after";
      break;
   default :
      return "vs.";
   }
}

function init()
{
   fixedRange = document.body.createTextRange();
   fixedRange.moveToElementText(fixedRangeElem);
}
```

Related Items: `Range.compareEndPoints()` method

duplicate()

Returns: `TextRange` object

Compatibility: WinIE4+, MacIE-, NN-, Moz-, Safari-, Opera+, Chrome-

The `duplicate()` method returns a `TextRange` object that is a snapshot copy of the current `TextRange` object. In a way, a non-intuitive relationship exists between the two objects. If you alter the `text` property of the copy without moving the start or end points of the original, the original takes on the new text. But if you move the start or end points of the original, the `text` and `htmlText` of the original obviously change, whereas the copy retains its properties from the time of the duplication. Therefore, this method can be used to clone text from one part of the document to other parts.

Example

Use The Evaluator (Chapter 4, "JavaScript Essentials") in WinIE to see how the `duplicate()` method works. Begin by creating a new `TextRange` object that contains the text of the `myP` paragraph element:

```
a = document.body.createTextRange()
a.moveToElementText(myP)
```

Next, clone the original range and preserve the copy in variable `b`:

```
b = a.duplicate()
```

The method returns no value, so don't be alarmed by the "undefined" that appears in the Results box. Move the original range so that it is an insertion point at the end of the body by first expanding it to encompass the entire body, and then collapsing it to the end:

```
a.expand("textedit")
a.collapse(false)
```

Now, insert the copy at the very end of the body:

```
a.text = b.text
```

If you scroll to the bottom of the page, you'll see a copy of the text.

Related Items: `Range.clone()`, `TextRange.isEqual()` methods

execCommand("*commandName*"[, *UIFlag*[, *value*]])

Returns: Boolean

Compatibility: WinIE4+, MacIE-, NN-, Moz-, Safari-, Opera-, Chrome-

IE4+ for Windows operating systems lets scripts access a very large number of commands that act on insertion points, abstract text ranges, and selections that are made with the help of the `TextRange` object. Access to these commands is through the `execCommand()` method, which works with `TextRange` objects and the `document` object (see the `document.execCommand()` method discussion in Chapter 29, "The Document and Body Objects," and list of document- and selection-related commands in Table 29-2).

The first, required, parameter is the name of the command that you want to execute. Only a handful of these commands offer unique capabilities that aren't better accomplished within the IE object model using style sheets. Of particular importance is the command that lets you copy a text range into the Clipboard. Most of the rest of the commands modify styles or insert HTML tags at the position of a collapsed text range. These actions are better handled by other means, but they are included in Table 33-2 for the sake of completeness only (see Table 29-2 for additional commands).

An optional second parameter is a Boolean flag to instruct the command to display any user interface artifacts that may be associated with the command. The default is `false`. For the third parameter, some commands require an attribute value for the command to work. For example, to insert a new paragraph at an insertion point, you pass the identifier to be assigned to the `id` attribute of the `p` element. The syntax is

```
myRange.execCommand("InsertParagraph", true, "myNewP");
```

The `execCommand()` method returns Boolean `true` if the command is successful, `false` if not successful. Some commands can return values (for example, finding out the font name of a selection), but these values are accessed through the `queryCommandValue()` method.

TABLE 33-2

TextRange.execCommand() Commands

Command	Parameter	Description
`Bold`	None	Encloses the text range in a `<b>` tag pair
`Copy`	None	Copies the text range into the Clipboard
`Cut`	None	Copies the text range into the Clipboard and deletes it from the document or text control
`Delete`	None	Deletes the text range
`InsertButton`	ID String	Inserts a `<button>` tag at the insertion point, assigning the parameter value to the `id` attribute
`InsertFieldset`	ID String	Inserts a `<fieldset>` tag at the insertion point, assigning the parameter value to the `id` attribute
`InsertHoritontalRule`	ID String	Inserts an `<hr>` tag at the insertion point, assigning the parameter value to the `id` attribute
`InsertIFrame`	ID String	Inserts an `<iframe>` tag at the insertion point, assigning the parameter value to the `id` attribute
`InsertInputButton`	ID String	Inserts an `<input type="button">` tag at the insertion point, assigning the parameter value to the `id` attribute
`InsertInputCheckbox`	ID String	Inserts an `<input type="checkbox">` tag at the insertion point, assigning the parameter value to the `id` attribute
`InsertInputFileUpload`	ID String	Inserts an `<input type="FileUpload">` tag at the insertion point, assigning the parameter value to the `id` attribute
`InsertInputHidden`	ID String	Inserts an `<input type="hidden">` tag at the insertion point, assigning the parameter value to the `id` attribute
`InsertInputImage`	ID String	Inserts an `<input type="image">` tag at the insertion point, assigning the parameter value to the `id` attribute
`InsertInputPassword`	ID String	Inserts an `<input type="password">` tag at the insertion point, assigning the parameter value to the `id` attribute
`InsertInputRadio`	ID String	Inserts an `<input type="radio">` tag at the insertion point, assigning the parameter value to the `id` attribute
`InsertInputReset`	ID String	Inserts an `<input type="reset">` tag at the insertion point, assigning the parameter value to the `id` attribute
`InsertInputSubmit`	ID String	Inserts an `<input type="submit">` tag at the insertion point, assigning the parameter value to the `id` attribute
`InsertInputText`	ID String	Inserts an `<input type="text">` tag at the insertion point, assigning the parameter value to the `id` attribute

Command	Parameter	Description
InsertMarquee	ID String	Inserts a <marquee> tag at the insertion point, assigning the parameter value to the id attribute
InsertOrderedList	ID String	Inserts an tag at the insertion point, assigning the parameter value to the id attribute
InsertParagraph	ID String	Inserts a <p> tag at the insertion point, assigning the parameter value to the id attribute
InsertSelectDropdown	ID String	Inserts a <select type="select-one"> tag at the insertion point, assigning the parameter value to the id attribute
InsertSelectListbox	ID String	Inserts a <select type="select-multiple"> tag at the insertion point, assigning the parameter value to the id attribute
InsertTextArea	ID String	Inserts an empty <textarea> tag at the insertion point, assigning the parameter value to the id attribute
InsertUnorderedList	ID String	Inserts a tag at the insertion point, assigning the parameter value to the id attribute
Italic	None	Encloses the text range in an <i> tag pair
OverWrite	Boolean	Sets the text input control mode to overwrite (true) or insert (false)
Paste	None	Pastes the current Clipboard contents into the insertion point or selection
PlayImage	None	Begins playing dynamic images if they are assigned to the dynsrc attribute of the img element
Refresh	None	Reloads the current page
StopImage	None	Stops playing dynamic images if they are assigned to the dynsrc attribute of the img element
Underline	None	Encloses the text range in a <u> tag pair

Although the commands in Table 33-2 work on text ranges, even the commands that work on selections (Table 29-2) can frequently benefit from some preprocessing with a text range. Consider, for example, a function whose job it is to find every instance of a particular word in a document and set its background color to a yellow highlight. Such a function utilizes the powers of the findText() method of a text range to locate each instance. Then the select() method selects the text in preparation for applying the BackColor command. Here is a sample:

```
function hiliteIt(txt)
{
   var rng = document.body.createTextRange();
   for (var i = 0; rng.findText(txt); i++)
{
```

```
            rng.select();
            rng.execCommand("BackColor", "false", "yellow");
            rng.execCommand("Unselect");
            // prepare for next search
            rng.collapse(false);
        }
    }
```

This example is a rare case that makes the `execCommand()` method way of modifying HTML content more efficient than trying to wrap some existing text inside a new tag. The downside is that you don't have control over the methodology used to assign a background color to a span of text (in this case, IE wraps the text in a `<font>` tag with a `style` attribute set to `background-color:yellow` — probably not the way you'd do it on your own).

Example

Use The Evaluator (Chapter 4, "JavaScript Essentials") to see how to copy a text range's text into the client computer's Clipboard. Begin by setting the text range to the `myP` element:

```
a = document.body.createTextRange()
a.moveToElementText(myP)
```

Now use `execCommand()` to copy the range into the Clipboard:

```
a.execCommand("Copy")
```

To prove that the text is in the Clipboard, click the bottom text field and choose Paste from the Edit menu (or type Ctrl+V).

Related Items: Several query command methods of the `TextRange` object

expand("*unit*")

Returns: Boolean

Compatibility: WinIE4+, MacIE-, NN-, Moz-, Safari-, Opera-, Chrome-

The single `expand()` method can open any range — collapsed or not — to the next largest character, word, or sentence, or to the entire original range (for example, encompassing the text of the `body` element if the range was generated by `document.body.createTextRange()`). The parameter is a string designating which unit to expand outward to: `character`, `word`, `sentence`, or `textedit`. If the operation is successful, the method returns `true`; otherwise it returns `false`.

When operating from an insertion point, the `expand()` method looks for the word or sentence that encloses the point. The routine is not very smart about sentences, however. If you have some text prior to a sentence that you want to expand to, but that text does not end in a period, the `expand()` routine expands outward until it can find either a period or the beginning of the range. Listing 26-14 demonstrates a workaround for this phenomenon. When expanding from an insertion point to a character, the method expands forward to the next character in character set

order. If the insertion point is at the end of the range, it cannot expand to the next characters, and the `expand()` method returns `false`.

It is not uncommon, in an extensive script that needs to move the start and end points all over the initial range, to perform several `collapse()` and `expand()` method operations from time to time. Expanding to the full range is a way to start some range manipulation with a clean slate, as if you just created the range.

Example

You can find examples of the `expand()` method in Listing 26-14.

Related Items: `TextRange.collapse()` method

findText("*searchString*"[, *searchScope*, *flags*])

Returns: Boolean

Compatibility: WinIE4+, MacIE-, NN-, Moz-, Safari-, Opera-, Chrome-

One of the most useful methods of the `TextRange` object is `findText()`, whose default behavior is to look through a text range, starting at the range's start point, up to the end of the range, in search of a case-insensitive match for a search string. If an instance is found in the range, the start and end points of the range are cinched up to the found text and the method returns `true`; otherwise it returns `false`, and the start and end points do not move. Only the rendered text is searched and not any of the tags or attributes.

Optional parameters let you exert some additional control over the search process. You can restrict the distance from a collapsed range to be used for searching. The `searchScope` parameter is an integer value indicating the number of characters from the start point. The larger the number, the more text of the range is included in the search. Negative values force the search to operate backward from the current start point. If you want to search backward to the beginning of the range, but you don't know how far away the start of the range is, you can specify an arbitrary number that's big enough to encompass the text.

The optional `flags` parameter lets you set whether the search is to be case-sensitive and/or to match whole words only. The parameter is a single integer value that uses bit-wise math to calculate the single value that accommodates one or both settings. The value for matching whole words is `2`; the value for matching case is `4`. If you want only one of those behaviors, then supply just the desired number. But for both behaviors, use the bit-wise XOR operator (the `^` operator) on the values to reach a value of `6`.

The most common applications of the `findText()` method include a search-and-replace action, and format changes to every instance of a string within the range. This iterative process requires some extra management. Because searching always starts with the range's current start point, advancing the start point to the end of the text found in the range is necessary. This advancing allows a successive application of `findText()` to look through the rest of the range for another

match. And because findText() ignores the arbitrary end point of the current range and continues to the end of the initial range, you can use the collapse(false) method to force the starting point to the end of the range that contains the first match.

A repetitive search can be accomplished by a while or for repeat loop. The returned Boolean value of the findText() method can act as the condition for continuing the loop. If the number of times the search succeeds is not essential to your script, a while loop works nicely:

```
while (rng.findText(searchString))
{
   // ...
   rng.collapse(false);
}
```

Or you can use a for loop counter to maintain a count of successes, such as a counter of how many times a string appears in the body:

```
for (var i = 0; rng.findText(searchString); i++)
{
   // ...
   rng.collapse(false);
}
```

Some of the operations you want to perform on a range (including many of the commands invoked by the execCommand() method) require that a selection exists for the command to work. Be prepared to use the select() method on the range after the findText() method locates a matching range of text.

Example

Listing 33-11 implements two varieties of text search-and-replace operations, while showing you how to include extra parameters for case-sensitive and whole-word searches. Both approaches begin by creating a TextRange for the entire body, but they immediately shift the starting point to the beginning of the div element that contains the text to search.

One search-and-replace function prompts the user to accept or decline replacement for each instance of a found string. The select() and scrollIntoView() methods are invoked to help the user see what is about to be replaced. Notice that even when the user declines to accept the replacement, the text range is collapsed to the end of the found range so that the next search can begin after the previously found text. Without the collapse() method, the search can get caught in an infinite loop as it keeps finding the same text over and over (with no replacement made). Because no counting is required, this search-and-replace operation is implemented inside a while repeat loop.

The other search-and-replace function goes ahead and replaces every match, and then displays the number of replacements made. After the loop exits (because there are no more matches), the loop counter is used to display the number of replacements.

LISTING 33-11

Two Search-and-Replace Approaches (with Undo)

HTML: jsb33-11.html

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>TextRange.findText() Method</title>
      <script type="text/javascript" src="jsb33-11.js"></script>
   </head>
   <body>
      <h1>TextRange.findText() Method</h1>
      <hr />
      <form>
         <p>Enter a string to search for in the following text:
            <input type="text" name="searchString"
                   size="20" value="Law" />  
            <input type="checkbox"
                   name="caseSensitive" />Case-sensitive  
            <input type="checkbox"
                   name="wholeWord" />Whole words only
         </p>
         <p>Enter a string with which to replace found text:
            <input type="text" name="replaceString"
                   size="20" value="legislation" />
         </p>
         <p><input type="button" value="Search and Replace (with prompt)"
                   onclick="sAndR(this.form)" />
         </p>
         <p><input type="button" onclick="sAndRCount(this.form)"
                   value="Search, Replace, and Count (no prompt)" />
            <span id="counter">0</span> items found and replaced.
         </p>
         <p><input type="button" value="Undo Search and Replace"
                   onclick="undoReplace()"/></p>
      </form>
      <div id="rights">
         <h2><a id="article1" name="article1">ARTICLE I</a></h2>
         <p>Congress shall make no law respecting an establishment of religion,
            or prohibiting the free exercise thereof; or abridging the freedom
            of speech, or of the press; or the right of the people peaceably to
            assemble, and to petition the government for a redress of
            grievances.</p>
         <h2><a name="article2">ARTICLE II</a></h2>
         <p>A well regulated militia, being necessary to the security of a
            free state, the right of the people to keep and bear arms, shall
            not be infringed.</p>
```

```
    <h2><a name="article3">ARTICLE III</a></h2>
    <p>No soldier shall, in time of peace, be quartered in any house,
       without the consent of the owner, nor in time of war, but in
       a manner to be prescribed by law.</p>
    <h2><a name="article4">ARTICLE IV</a></h2>
    <p>The right of the people to be secure in their persons, houses,
       papers, and effects, against unreasonable searches and seizures
       shall not be violated, and no warrants shall issue, but upon
       probable cause, supported by oath or affirmation, and
       particularly describing the place to be searched, and the
       persons or things to be seized.</p>
    <h2><a name="article5">ARTICLE V</a></h2>
    <p>No person shall be held to answer for a capital, or otherwise
       infamous crime, unless on a presentment or indictment of a
       grand jury, except in cases arising in the land or naval
       forces, or in the militia, when in actual service in time
       of war or public danger; nor shall any person be subject for
       the same offense to be twice put in jeopardy of life or limb;
       nor shall be compelled in any criminal case to be a witness
       against himself, nor be deprived of life, liberty, or
       property, without due process of law; nor shall private
       property be taken for public use, without just compensation.</p>
    <h2><a name="article6">ARTICLE VI</a></h2>
    <p>In all criminal prosecutions the accused shall enjoy the right
       to a speedy and public trial, by an impartial jury of the state
       and district wherein the crime shall have been committed, which
       district shall have been previously ascertained by law, and to
       be informed of the nature and cause of the accusation; to be
       confronted with the witnesses against him; to have compulsory
       process for obtaining witnesses in his favor, and to have the
       assistance of counsel for his defense.</p>
    <h2><a name="article7">ARTICLE VII</a></h2>
    <p>In suits at common law, where the value in controversy shall
       exceed twenty dollars, the right of trial by jury shall be
       preserved, and no fact tried by a jury shall be otherwise
       re-examined in any court of the United States, than according
       to the rules of the common law.</p>
    <h2><a name="article8">ARTICLE VIII</a></h2>
    <p>Excessive bail shall not be required, nor excessive fines
       imposed, nor cruel and unusual punishments inflicted.</p>
    <h2><a name="article9">ARTICLE IX</a></h2>
    <p>The enumeration in the Constitution, of certain rights, shall
       not be construed to deny or disparage others retained by the
       people.</p>
    <h2><a name="article10">ARTICLE X</a></h2>
    <p>The powers not delegated to the United States by the Constitution,
       nor prohibited by it to the states, are reserved to the states
       respectively, or to the people.</p>
</div>
```

continued

LISTING 33-11 *(continued)*

```
   </body>
</html>
```

JavaScript: jsb33-11.js

```
// global range var for use with Undo
var rng;

function replaceHTML(elem, text)
{
   while(elem.firstChild)
   {
      elem.removeChild(elem.firstChild);
   }
   elem.appendChild(document.createTextNode(text));
}

// return findText() third parameter arguments
function getArgs(form)
{
   var isCaseSensitive = (form.caseSensitive.checked) ? 4 : 0;
   var isWholeWord = (form.wholeWord.checked) ? 2 : 0;
   return isCaseSensitive ^ isWholeWord;
}

// prompted search and replace
function sAndR(form)
{
   var replString = form.replaceString.value;
   var srchString = form.searchString.value;
      if (srchString)
      {
         var args = getArgs(form);
         rng = document.body.createTextRange();
         rng.moveToElementText(rights);
         clearUndoBuffer();
         while (rng.findText(srchString, 10000, args))
         {
            rng.select();
            rng.scrollIntoView();
            if (confirm("Replace?"))
            {
               rng.text = replString;
               pushUndoNew(rng, srchString, replString);
            }
            rng.collapse(false);
         }
      }
}
```

```
// unprompted search and replace with counter
function sAndRCount(form)
{
   var i;
   var replString = form.replaceString.value;
   var srchString = form.searchString.value;
      if (srchString)
      {
         var args = getArgs(form);
         rng = document.body.createTextRange();
         rng.moveToElementText(rights);
         for (i = 0; rng.findText(srchString, 10000, args); i++)
         {
            rng.text = replString;
            pushUndoNew(rng, srchString, replString);
            rng.collapse(false);
         }
         if (i > 1)
         {
            clearUndoBuffer();
         }
      }
   replaceHTML(document.getElementById("counter"), i);
}

// BEGIN UNDO BUFFER CODE
// buffer global variables
var newRanges = new Array();
var origSearchString;

// store original search string and bookmarks of each replaced range
function pushUndoNew(rng, srchString, replString)
{
   origSearchString = srchString;
   rng.moveStart("character", -replString.length);
   newRanges[newRanges.length] = rng.getBookmark();
}

// empty array and search string global
function clearUndoBuffer()
{
   replaceHTML(document.getElementById("counter"), "0");
   origSearchString = "";
   newRanges.length = 0;
}

// perform the undo
function undoReplace()
```

continued

LISTING 33-11 *(continued)*

```
{
   if (newRanges.length && origSearchString)
   {
      for (var i = 0; i < newRanges.length; i++)
      {
         rng.moveToBookmark(newRanges[i]);
         rng.text = origSearchString;
      }
      replaceHTML(document.getElementById("counter"), i);
      clearUndoBuffer();
   }
}
```

Having a search-and-replace function available in a document is only one half of the battle. The other half is offering the facilities to undo the changes. To that end, Listing 33-11 includes an undo buffer that accurately undoes only the changes made in the initial replacement actions.

The undo buffer stores its data in two global variables. The first, `origSearchString`, is simply the string used to perform the original search. This variable is the string that has to be put back in the places where it had been replaced. The second global variable is an array that stores `TextRange` bookmarks (see `getBookmark()` later in this chapter). These references are string values that don't mean much to humans, but the browser can use them to re-create a range with its desired start and end point. Values for both the global search string and bookmark specifications are stored in calls to the `pushUndoNew()` method each time text is replaced.

A perhaps unexpected action of setting the `text` property of a text range is that the start and end points collapse to the end of the new text. Because the stored bookmark must include the replaced text as part of its specification, the start point of the current range must be adjusted back to the beginning of the replacement text before the bookmark can be saved. Thus, the `pushUndoNew()` function receives the replacement text string so that the `moveStart()` method can be adjusted by the number of characters matching the length of the replacement string.

After all of the bookmarks are stored in the array, the undo action can do its job in a rather simple `for` loop inside the `undoReplace()` function. After verifying that the undo buffer has data stored in it, the function loops through the array of bookmarks and replaces the bookmarked text with the old string. The benefit of using the bookmarks this time, rather than the replacement function, is that only those ranges originally affected by the search-and-replace operation are touched in the undo operation. For example, in this document, if you replace a case-sensitive "states" with "States," two replacements are performed. At that point, however, the document has four instances of "States," two of which existed before. Redoing the replacement function by inverting the search-and-replace strings would convert all four back to the lowercase version — not the desired effect.

Related Items: `TextRange.select()` method

getBookmark()

Returns: Bookmark string

Compatibility: WinIE4+, MacIE-, NN-, Moz-, Safari-, Opera-, Chrome-

In the context of a `TextRange` object, a bookmark is not to be confused with the kinds of bookmarks you add to a browser list of favorite web sites. Instead, a bookmark is a string that represents a definition of a text range, including its location in a document, its text, and so on. Viewing the string is futile, because it contains string versions of binary data, so the string means nothing in plain language. But, a bookmark allows your scripts to save the current state of a text range so that it may be restored at a later time. The `getBookmark()` method returns the string representation of a snapshot of the current text range. Some other script statement can adjust the `TextRange` object to the exact specifications of the snapshot with the `moveToBookmark()` method (described later in this chapter).

Example

Listing 33-11, earlier in this chapter, shows how the `getBookmark()` method is used to preserve specifications for text ranges so that they can be called upon again to be used to undo changes made to the text range. The `getBookmark()` method is used to save the snapshots, whereas the `moveToBookmark()` method is used during the undo process.

Related Items: `TextRange.moveToBookmark()` method

inRange(*otherRangeRef*)

Returns: Boolean

Compatibility: WinIE4+, MacIE-, NN-, Moz-, Safari-, Opera-, Chrome-

You can compare the physical stretches of text contained by two different text ranges via the `inRange()` method. Typically, you invoke the method on the larger of the two ranges and pass a reference to the smaller range as the sole parameter to the method. If the range passed as a parameter is either contained by or equal to the text range that invokes the method, the method returns `true`; otherwise the method returns `false`.

Example

Use The Evaluator (Chapter 4, "JavaScript Essentials") to see the `inRange()` method in action. The following statements generate two distinct text ranges, one for the `myP` paragraph element and the other for the `myEM` element nested within:

```
a = document.body.createTextRange()
a.moveToElementText(myP)
b = document.body.createTextRange()
b.moveToElementText(myEM)
```

Because the `myP` text range is larger than the other, we invoke the `inRange()` method on it, fully expecting the return value of `true`:

```
a.inRange(b)
```

But if you switch the references, you see that the larger text range is not “in” the smaller one:

```
b.inRange(a)
```

Related Items: `TextRange.isEqual()` method

isEqual(*otherRangeRef*)

Returns: Boolean

Compatibility: WinIE4+, MacIE-, NN-, Moz-, Safari-, Opera-, Chrome-

If your script has references to two independently adjusted `TextRange` objects, you can use the `isEqual()` method to test whether the two objects are identical. This method tests for a very literal equality, requiring that the text of the two ranges be character-for-character and position-for-position equal in the context of the original ranges (for example, body or text control content). To see if one range is contained by another, use the `inRange()` method instead.

Example

Use The Evaluator (Chapter 4, “JavaScript Essentials”) to try the `isEqual()` method. Begin by creating two separate `TextRange` objects, one for the `myP` element and one for `myEM`:

```
a = document.body.createTextRange()
a.moveToElement(myP)
b = document.body.createTextRange()
b.moveToElement(myEM)
```

Because these two ranges encompass different sets of text, they are not equal, as the results show from the following statement:

```
a.isEqual(b)
```

But if you now adjust the first range boundaries to surround the `myEM` element, both ranges are the same values:

```
a.moveToElement(myEM)
a.isEqual(b)
```

Related Items: `TextRange.inRange()` method

move("*unit*"[, *count*])

Returns: Integer

Compatibility: WinIE4+, MacIE-, NN-, Moz-, Safari-, Opera+, Chrome-

The `move()` method performs two operations. First, the method collapses the current text range to become an insertion point at the location of the previous end point. Next, it moves that insertion point to a position forward or backward any number of character, word, or sentence units.

The first parameter is a string specifying the desired unit (`character`, `word`, `sentence`, or `textedit`). A value of `textedit` moves the pointer to the beginning or end of the entire initial text range. If you omit the second parameter, the default value is 1. Otherwise, you can specify an integer indicating the number of units the collapsed range should be moved ahead (positive integer) or backward (negative). The method returns an integer revealing the exact number of units the pointer is able to move — if you specify more units than are available, the returned value lets you know how far it can go.

Bear in mind that the range is still collapsed after the `move()` method executes. Expanding the range around the desired text is the job of other methods.

You can also use the `move()` method in concert with the `select()` method to position the flashing text insertion pointer within a text box or textarea. Thus, you can script a text field that, upon receiving focus or when the page loads, has the text pointer waiting for the user at the end of existing text. A generic function for such an action is shown in the following code:

```
function setCursorToEnd(elem)
{
   if (elem)
   {
      if (elem.type && (elem.type == "text" || elem.type == "textarea"))
      {
         var rng = elem.createTextRange();
         rng.move("textedit");
         rng.select();
      }
   }
}
```

You can then invoke this method from a text field's `onfocus` event handler:

```
<input type="text" ... onfocus="setCursorToEnd(this)" />
```

The function previously shown includes a couple of layers of error checking, such as making sure that the function is invoked with a valid object as a parameter, and that the object has a `type` property whose value is capable of having a text range made for its content.

Example

Use The Evaluator (Chapter 4, "JavaScript Essentials") to experiment with the `move()` method in IE. To see how the method returns just the number of units it moves the pointer, begin by creating a text range and set it to enclose the `myP` element:

```
a = document.body.createTextRange()
a.moveToElementText(myP)
```

Now enter the following statement to collapse and move the range backward by 20 words:

```
a.move("word", -20)
```

Continue to click the Evaluate button and watch the returned value in the Results box. The value shows `20` while it can still move backward by 20 words. But eventually the last movement will be some other value closer to zero. And after the range is at the beginning of the `body` element, the range can move no more in that direction, so the result is zero.

Related Items: `TextRange.moveEnd()`, `TextRange.moveStart()` methods

moveEnd("*unit*"[, *count*])
moveStart("*unit*"[, *count*])

Returns: Integer

Compatibility: WinIE4+, MacIE-, NN-, Moz-, Safari-, Opera+, Chrome-

The `moveEnd()` and `moveStart()` methods are similar to the `move()` method, but they each act only on the end and starting points of the current range, respectively. In other words, the range does not collapse before the point is moved. These methods allow you to expand or shrink a range by a specific number of units by moving only one of the range's boundaries.

The first parameter is a string specifying the desired unit (`character`, `word`, `sentence`, or `textedit`). A value of `textedit` moves the pointer to the beginning or end of the entire initial text range. Therefore, if you want the end point of the current range to zip to the end of the body (or text form control), use `moveEnd("textedit")`. If you omit the second parameter, the default value is `1`. Otherwise you can specify an integer indicating the number of units the collapsed range is to move ahead (positive integer) or backward (negative). Moving either point beyond the location of the other, forces the range to collapse and move to the location specified by the method. The method returns an integer revealing the exact number of units the pointer is able to move — if you specify more units than are available, the returned value lets you know how far it can go.

Example

Use The Evaluator (Chapter 4, "JavaScript Essentials") to experiment with the `moveEnd()` and `moveStart()` methods. Begin by creating a text range and set it to enclose the `myEM` element:

```
a = document.body.createTextRange()
a.moveToElementText(myEM)
```

To help you see how movements of the pointers affect the text enclosed by the range, type `a` into the bottom text box and view all the properties of the text range. Note especially the `htmlText` and `text` properties.

Now enter the following statement to move the end of the range forward by one word:

```
a.moveEnd("word")
```

Click the List Properties button to see that the text of the range now includes the word following the `em` element. Try each of the following statements in the top text box and examine both the

integer results and (by clicking the List Properties button) the properties of the range after each statement:

```
a.moveStart("word", -1)
a.moveEnd("sentence")
```

Notice that for a sentence, a default unit of 1 expands to the end of the current sentence. And if you move the start point backward by one sentence, you'll see that the lack of a period-ending sentence prior to the `myP` element causes strange results.

Finally, force the start point backward in increments of 20 words and watch the results as the starting point nears and reaches the start of the `body`:

```
a.moveStart("word", -20)
```

Eventually the last movement will be some other value closer to zero. And as soon as the range is at the beginning of the `body` element, the range can move no more in that direction, so the result is zero.

Related Items: `TextRange.move()` method

moveToBookmark("*bookmarkString*")

Returns: Boolean

Compatibility: WinIE4+, MacIE-, NN-, Moz-, Safari-, Opera-, Chrome-

If a snapshot of a text range specification has been preserved in a variable (with the help of the `getBookmark()` method), the `moveToBookmark()` method uses that bookmark string as its parameter to set the text range to exactly the way it appeared when the bookmark was originally obtained. If the method is successful, it returns a value of `true`, and the text range is set to the same string of text as originally preserved via `getBookmark()`. It is possible that the state of the content of the text range has been altered to such an extent that resurrecting the original text range is not feasible. In that case, the method returns `false`.

Example

Listing 33-11, earlier in this chapter, shows how to use the `moveToBookmark()` method to restore a text range so that changes that created the state saved by the bookmark can be undone. The `getBookmark()` method is used to save the snapshots, whereas the `moveToBookmark()` method is used during the undo process.

Related Items: `TextRange.getBookmark()` method

moveToElementText(*elemObjRef*)

Returns: Nothing

Compatibility: WinIE4+, MacIE-, NN-, Moz-, Safari-, Opera+, Chrome-

The fastest way to cinch up a text range to the boundaries of an HTML element on the page is to use the `moveToElementText()` method. Any valid reference to the HTML element object is

accepted as the sole parameter — just don't try to use a string version of the object ID unless it is wrapped in the W3C `document.getElementById()` method (IE5+). When the boundaries are moved to the element, the range's `htmlText` property contains the tags for the element.

Example

A majority of examples for other `TextRange` object methods in this chapter use the `moveToElementText()` method. Listings 33-10 and 33-11, earlier in this chapter, show the method within an application context.

Related Items: `TextRange.parentElement()` method

moveToPoint(*x*, *y*)

Returns: Nothing

Compatibility: WinIE4+, MacIE-, NN-, Moz-, Safari-, Opera-, Chrome-

The `moveToPoint()` method shrinks the current text range object to an insertion point and then moves it to a position in the current browser window or frame. You control the precise position via the `x` (horizontal) and `y` (vertical) pixel coordinates specified as parameters. The position is relative to the visible window, and not the document, which may have been scrolled to a different position. Invoking the `moveToPoint()` method is the scripted equivalent of the user clicking that spot in the window. Use the `expand()` method to flesh out the collapsed text range to encompass the surrounding character, word, or sentence.

Note

Using the `moveToPoint()` method on a text range defined for a text form control may cause a browser crash in versions of IE prior to 7. The method appears safe with the `document.body` text ranges, even if the x,y position falls within the rectangle of a text control. Such a position, however, does not drop the text range into the form control or its content. ■

Example

Use The Evaluator (Chapter 4, "JavaScript Essentials") to see the `moveToPoint()` method in action. Begin by creating a text range for the entire `body` element:

```
a = document.body.createTextRange()
```

Now, invoke the `moveToPoint()` method to a location `100,100`, which turns out to be in the rectangle space of the Results textarea:

```
a.moveToPoint(100,100)
```

If you type `a` into the bottom text box and view the properties, both the `htmlText` and `text` properties are empty because the insertion point represents no visible text content. But if you gradually move, for example, the start point backward one character at a time, you will see the `htmlText` and `text` properties begin to fill in with the body text that comes before the `textarea` element, namely the "Results:" label, the `<br>` tag between it, and the `textarea`

element. Enter the following statement into the top text box and click the Evaluate button several times:

```
a.moveStart("character", -1)
```

Enter `a` into the bottom text box after each evaluation to list the properties of the range.

Related Items: `TextRange.move()`, `TextRange.moveStart()`, `TextRange.moveEnd()` methods

parentElement()

Returns: Element object reference

Compatibility: WinIE4+, MacIE-, NN-, Moz-, Safari-, Opera+, Chrome-

The `parentElement()` method returns a reference to the next outermost HTML element container that holds the text range boundaries. If the text range boundaries are at the boundaries of a single element, the `parentElement()` method returns that element's reference. But if the boundaries straddle elements, the object returned by the method is the element that contains the text of the least-nested text portion. In contrast to the `expand()` and various move-related methods, which understand text constructs, such as words and sentences, the `parentElement()` method is concerned solely with element objects. Therefore, if a text range is collapsed to an insertion point in body text, you can expand it to encompass the HTML element by using the `parentElement()` method as a parameter to `moveToElementText()`:

```
rng.moveToElementText(rng.parentElement());
```

Example

Use The Evaluator (Chapter 4, "JavaScript Essentials") to experiment with the `parentElement()` method. Begin by setting the text range to the `myEM` element:

```
a = document.body.createTextRange()
a.moveToElementText(myEM)
```

To inspect the object returned by the `parentElement()` method, enter the following statement in the lower text box:

```
a.parentElement()
```

If you scroll down to the `outerHTML` property, you see that the parent of the text range is the `myEM` element, tag and all.

Next, extend the end point of the text range by one word:

```
a.moveEnd("word")
```

Because part of the text range now contains text of the `myP` object, the `outerHTML` property of `a.parentElement()` shows the entire `myP` element and tags.

Related Items: `TextRange.expand()`, `TextRange.move()`, `TextRange.moveEnd()`, `TextRange.moveStart()` methods

pasteHTML("*HTMLText*")

Returns: Nothing

Compatibility: WinIE4+, MacIE-, NN-, Moz-, Safari-, Opera-, Chrome-

Although the `execCommand()` method offers several commands that insert HTML elements into a text range, it is probably more convenient to simply paste fully-formed HTML into the current text range (assuming you need to be working with a text range instead of even more simply setting new values to an element object's `outerHTML` property). Provide the HTML to be inserted as a string parameter to the `pasteHTML()` method.

Use the `pasteHTML()` method with some forethought. Some HTML that you may attempt to paste into a text range may force the method to wrap additional tags around the content you provide, to ensure the validity of the resulting HTML. For example, if you were to replace a text range consisting of a portion of the text of a `p` element with, for instance, an `li` element, the `pasteHTML()` method has no choice but to divide the `p` element into two pieces, because a `p` element is not a valid container for a solo `li` element. This division can greatly disrupt your document object hierarchy, because the divided `p` element assumes the same ID for both pieces. Existing references to that `p` element will break, because the reference now returns an array of two like-named objects.

Example

Use The Evaluator (Chapter 4, "JavaScript Essentials") to experiment with the `pasteHTML()` method. The goal of the following sequence is to change the `<em>` tag to a `<span>` tag whose `style` attribute sets the color of the original text that was in the `em` element.

Begin by creating the text range and setting the boundaries to the `myEM` element:

```
a = document.body.createTextRange()
a.moveToElementText(myEM)
```

Although you can pass the HTML string directly as a parameter to `pasteHTML()`, it may be more convenient (and more easy to test) to store the HTML string in its own temporary variable, like this:

```
b = "<span style='color:red'>" + a.text + "</span>"
```

Notice that we concatenate the text of the current text range, because it has not yet been modified. Now we can paste the new HTML string into the current text range:

```
a.pasteHTML(b)
```

At this point the `em` element is gone from the object model, and the `span` element is in its place. Prove it to yourself by looking at the HTML for the `myP` element:

```
myP.innerHTML
```

As noted earlier, the `pasteHTML()` method is not the only way to insert or replace HTML in a document. This method makes excellent sense when the user selects some text in the document to be replaced, because you can use the `document.selection.createRange()` method to get the text range for the selection. But if you're not using text ranges for other related operations, consider the other generic object properties and methods available to you.

Related Items: `outerHTML` property; `insertAdjacenHTML()` method

queryCommandEnabled("*commandName*")
queryCommandIndeterm("*commandName*")
queryCommandState("*commandName*")
queryCommandSupported("*commandName*")
queryCommandText("*commandName*")
queryCommandValue("*commandName*")

Returns: See `document.queryCommandEnabled()` in Chapter 29, "The Document and Body Objects."

Compatibility: WinIE4+, MacIE-, NN-, Moz-, Safari-, Opera-, Chrome-

See descriptions under `document.queryCommandEnabled()` in Chapter 29.

select()

Returns: Nothing

Compatibility: WinIE4+, MacIE-, NN-, Moz-, Safari-, Opera+, Chrome-

The `select()` method selects the text inside the boundaries of the current text range. For some operations, such as prompted search and replace, it is helpful to show the user the text of the current range to highlight what text is about to be replaced. In some other operations, especially several commands invoked by `execCommand()`, the operation works only on a text selection in the document. Thus, you can use the `TextRange` object facilities to set the boundaries, followed by the `select()` method to prepare the text for whatever command you like. Text selected by the `select()` method becomes a `selection` object (covered earlier in this chapter).

Example

See Listing 33-11, earlier in this chapter, for an example of the `select()` method in use.

Related Items: `selection` object

setEndPoint("*type*", *otherRangeRef*)

Returns: Nothing.

Compatibility: WinIE4+, MacIE-, NN-, Moz-, Safari-, Opera-, Chrome-

In contrast to the `moveEnd()` method, which adjusts the end point of the current range with respect to characters, words, sentences, and the complete range, the `setEndPoint()` method sets a boundary of the current range (not necessarily the ending boundary) relative to a boundary

of another text range whose reference is passed as the second parameter. The first parameter is one of four types that control which boundary of the current range is to be adjusted and which boundary of the other range is the reference point. Table 33-3 shows the four possible values and their meanings.

TABLE 33-3

setEndPoint() Method Types

Type	Description
`StartToEnd`	Moves the start point of the current range to the end of the other range
`StartToStart`	Moves the start point of the current range to the start of the other range
`EndToStart`	Moves the end point of the current range to the start of the other range
`EndToEnd`	Moves the end point of the current range to the end of the other range

Note that the method moves only one boundary of the current range at a time. If you want to make two ranges equal to each other, you have to invoke the method twice, once with `StartToStart` and once with `EndToEnd`. At that instant, the `isEqual()` method applied to those two ranges returns `true`.

Setting a boundary point with the `setEndPoint()` method can have unexpected results when the revised text range straddles multiple elements. Don't be surprised to find that the new HTML text for the revised range does not include tags from the outer element container.

Example

Use The Evaluator (Chapter 4, "JavaScript Essentials") to experiment with the `setEndPoint()` method. Begin by creating two independent text ranges, one for the `myP` element and one for `myEM`:

```
a = document.body.createTextRange()
a.moveToElementText(myP)
b = document.body.createTextRange()
b.moveToElementText(myEM)
```

Before moving any end points, compare the HTML for each of those ranges:

```
a.htmlText
b.htmlText
```

Now, move the start point of the `a` text range to the end point of the `b` text range:

```
a.setEndPoint("StartToEnd", b)
```

If you now view the HTML for the `a` range,

```
a.htmlText
```

you see that the <p> tag of the original `a` text range is nowhere to be found. This demonstration is a good lesson in using the `setEndPoint()` method primarily if you are concerned only with visible body text being inside ranges, rather than an element with its tags.

Related Items: `TextRange.moveEnd()`, `TextRange.moveStart()`, `TextRange.moveToElementText()` methods

TextRectangle Object

Compatibility: WinIE5+, MacIE-, NN-, Moz5+, Safari4+, Opera9.5+, Chrome3+

Properties	Methods	Event Handlers
`bottom`		
`left`		
`right`		
`top`		

Syntax

Accessing `TextRectangle` object properties:

```
(IE5+) [window.]document.getElementById("elemID").
          getBoundingClientRect().property
(IE5+) [window.]document.getElementById("elemID").
          getClientRects()[i].property
```

About this object

The `TextRectangle` object exposes to scripts a concept that is described in the HTML 4.0 specification, whereby an element's rendered text occupies a rectangular space on the page just large enough to contain the text. For a single word, the rectangle is as tall as the line height for the font used to render the word, and no wider than the space occupied by the text. But for a sequence of words that wraps to multiple lines, the rectangle is as tall as the line height times the number of lines, and as wide as the distance between the leftmost and rightmost character edges, even if it means that the rectangle encloses some other text that is not part of the element.

If you extract the `TextRectangle` object for an element by way of, for example, the `getBoundingClientRect()` method, be aware that the object is but a snapshot of the rectangle when the method was invoked. Resizing the page may very well alter dimensions of the actual rectangle enclosing the element's text, but the `TextRectangle` object copy that you made previously does not change its values to reflect the element's physical changes. After a window resize or modification of body text, any dependent `TextRectangle` objects should be recopied from the element.

Properties

bottom
left
right
top

Values: Integers Read-Only

Compatibility: WinIE5+, MacIE-, NN-, Moz-, Safari4+, Opera9.5+, Chrome3+

The screen pixel coordinates of its four edges define every `TextRectangle` object. These coordinates are relative to the window or frame displaying the page. Therefore, if you intend to align a positioned element with an inline element's `TextRectangle`, your position assignments must take into account the scrolling of the body.

To our eyes, the left edge of a `TextRectangle` does not always fully encompass the left-most pixels of the rendered text. You may have to fudge a few pixels in the measure when trying to align a real element with the `TextRectangle` of another element.

Example

Listing 33-12 lets you click one of four nested elements to see how the `TextRectangle` is treated. When you click one of the elements, that element's `TextRectangle` dimension properties are used to set the size of a positioned element that highlights the space of the rectangle. Be careful not to confuse the visible rectangle object that you see on the page with the abstract `TextRectangle` object that is associated with each of the clicked elements.

An important part of the listing is the way the action of sizing and showing the positioned element is broken out as a separate function (`setHiliter()`) from the one that is the `onclick` event handler function (`handleClick()`). This is done so that the `onresize` event handler can trigger a script that gets the current rectangle for the last element clicked, and the positioned element can be sized and moved to maintain the highlight of the same text. As an experiment, try removing the `onresize` event handler from the `<body>` tag and watch what happens to the highlighted rectangle after you resize the browser window: the rectangle that represents the `TextRectangle` remains unchanged, and loses track of the abstract `TextRectangle` associated with the actual element object.

LISTING 33-12

Using the TextRectangle Object Properties

HTML: jsb33-12.html

```
<html>
   <head>
      <title>TextRectangle Object</title>
```

```
      <script type="text/javascript" src="../jsb-global.js"></script>
      <script type="text/javascript" src="jsb33-12.js"></script>
   </head>
   <body>
   <h1>TextRectangle Object</h1>
      <hr />
      <p>Click on any of the four colored elements in the paragraph below and
         watch the highlight rectangle adjust itself to the element's
         TextRectangle object.</p>
      <p id="sample" class="sample">Lorem ipsum dolor sit amet, <span class="sample"
         style="color:red">consectetaur adipisicing elit</span>, sed do eiusmod
         tempor <span class="sample" style="color:green">incididunt ut labore
         et dolore <span class="sample" style="color:blue">magna aliqua</span>.
         Ut enim adminim veniam, quis nostrud exercitation ullamco</span>
         laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor
         in reprehenderit involuptate velit esse cillum dolore eu fugiat nulla
         pariatur.</p>
      <div id="hiliter"
          style="position:absolute; background-color:salmon; z-index:-1;↵
          visibility:hidden">
      </div>
   </body>
</html>
```

JavaScript: jsb33-12.js

```
// initialize when the page has loaded
addEvent(window, "load", initialize);

function initialize()
{
   addEvent(document.getElementById("sample"), "click", handleClick);
   addEvent(document.getElementById("sample"), "resize", setHiliter);
}

// preserve reference to last clicked elem so resize can re-use it
var lastElem;
// TextRectangle left tends to be out of registration by a couple of pixels
var rectLeftCorrection = 2;

// process mouse click
function handleClick()
{
   var elem = (event.target)
              ? event.target : event.srcElement;
   if (elem.className && elem.className == "sample")
   {
      // set hiliter element only on a subset of elements
      lastElem = elem;
      setHiliter();
   }
   else
```

continued

LISTING 33-12 *(continued)*

```
   {
      // otherwise, hide the hiliter
      hideHiliter();
   }
}

function setHiliter()
{
   if (lastElem)
   {
      var textRect = lastElem.getBoundingClientRect();
      hiliter.style.pixelTop = textRect.top + document.body.scrollTop;
      hiliter.style.pixelLeft = textRect.left + document.body.scrollLeft -
         rectLeftCorrection;
      hiliter.style.pixelHeight = textRect.bottom - textRect.top;
      hiliter.style.pixelWidth = textRect.right - textRect.left;
      hiliter.style.visibility = "visible";
   }
}

function hideHiliter()
{
   hiliter.style.visibility = "hidden";
   lastElem = null;
}
```

Related Items: `getBoundingClientRect()`, `getClientRects()` methods of element objects (Chapter 26, "Generic HTML Element Objects")

CHAPTER

34

The Form and Related Objects

IN THIS CHAPTER

The form object as a container of form controls

Processing form validations

label, fieldset, and legend element objects

Putting Web Forms 2.0 to work

Prior to the advent of dynamic object models and automatic page reflow, the majority of scripting in an HTML document took place in and around forms. Even with all the modern DHTML powers, forms remain the primary user interface elements of HTML documents because they enable users to input information and make choices in a very familiar way, using buttons, option lists, and so on.

Expanded object models of W3C-compatible browsers include scriptable access to form-related elements that are part of the HTML 4.0 specification. One pair of elements, `fieldset` and `legend`, provides both contextual and visual containment of form controls in a document. Another element, `label`, provides context for text labels that usually appear adjacent to form controls. Although there is generally little reason to script these objects, the browsers give you access to them just as they do for virtually every HTML element supported by the browser.

An interesting new twist in the form equation is Web Forms 2.0, which is a dramatically improved form technology based upon the original form features in HTML 4.0. Web Forms 2.0 establishes a powerful and consistent set of form controls that includes built-in validation and much-needed standard controls, such as an interactive date picker, among other things. Only Opera 9+ supports the new features, but other browsers will likely adopt them in the future.

The Form in the Object Hierarchy

Take another look at the JavaScript object hierarchy in the lowest common denominator object model (refer to Figure 25-1). The `form` element object can contain a wide variety of form element objects (sometimes called *form controls*), which we cover in Chapters 34 through 36. In this chapter, however, we focus primarily on the container.

The good news on the compatibility front is that much of the client-side scripting works on all scriptable browsers. We encourage you to use the newer `getElementById()` approach of addressing forms and their nested elements when your audience uses newer browsers exclusively. However, it can serve you well to be comfortable with the old-fashioned reference syntax. In fact, one of the older approaches to form referencing is still endorsed by the W3C, so you can use it without worrying about your code being antiquated for the sake of compatibility. Knowing this, almost all example code in this and the next three chapters uses syntax that is compatible across all browsers, including the earliest scriptable browsers.

form Object

For HTML element properties, methods, and event handlers, see Chapter 26.

Properties	Methods	Event Handlers
acceptCharset	handleEvent()	onreset
action	reset()	onsubmit
autocomplete	submit()	
elements[]		
encoding		
enctype		
length		
method		
name		
target		

Syntax

Accessing `form` object properties or methods:

```
(All)       [window.]document.formName. property | method([parameters])
(All)       [window.]document.forms[index]. property | method([parameters])
(IE4+)      [window.]document.all.elemID.property | method([parameters])
(All/W3C)   [window.]document.forms["formName"]. property |
            method([parameters])
(All/W3C)   [window.]document.forms["formName"].elements["property"] |
            method([parameters])
(IE5+/W3C)  [window.]document.getElementById("elemID").property |
            method([parameters])
```

Compatibility: WinIE3+, MacIE3+, NN2+, Moz+, Safari+, Opera+, Chrome+

About this object

Forms and their elements are the most common two-way gateways between users and JavaScript scripts. A form control element provides the only way that users can enter textual information across all browsers. Form controls also provide somewhat standardized and recognizable user interface elements for the user to make a selection from a predetermined set of choices. Those choices appear in the form of an on/off check box, in a set of mutually exclusive radio buttons, or as a selection from a list.

As you have seen in many web sites, the form is the avenue for the user to enter information that is sent to the server housing the web files. Just what the server does with this information depends on the programs running on the server. If your web site runs on a server directly under your control (that is, it is *in-house* or *hosted* by a service), you have the freedom to set up all kinds of data-gathering or database search programs to interact with the user. But with some of the more consumer-oriented Internet service providers (ISPs), you may have no server-side application support available — or, at best, a limited set of popular but inflexible CGI (Common Gateway Interface) programs, which are available to all customers of the service. Custom databases or transactional services are rarely provided for this kind of Internet service.

Regardless of your Internet server status, you can find plenty of uses for JavaScript scripts in forms. For instance, rather than using data exchanges (and Internet bandwidth) to gather raw user input and report any input errors, a JavaScript-enhanced document can preprocess the information to make sure that it employs the format that your back-end database or other programs most easily process. All corrective interaction takes place in the browser, without one extra bit flowing across the Net. We devote all of Chapter 46 (on the CD-ROM) to these kinds of form data-validation techniques. Additionally, Web Forms 2.0, which you meet later in this chapter, includes built-in support for many of the common validation tasks that are typically solved via JavaScript.

How you define a `form` element (independent of the user interface elements described in subsequent chapters) depends a great deal on how you plan to use the information from the form's controls. If you intend to use the form exclusively for JavaScript purposes (that is, no queries or postings going to the server), you do not need to use the `action`, `target`, and `method` attributes. But if your web page will be feeding information or queries back to a server, you need to specify at least the `action` and `method` attributes. You need to also specify the `target` attribute if the resulting data from the server is to be displayed in a window other than the calling window, and the `enctype` attribute if your form's scripts fashion the server-bound data in a MIME type other than in a plain ASCII stream.

References to form control elements

For most client-side scripting, user interaction comes from the elements within a form; the `form` element object is merely a container for the various control elements. If your scripts perform any data validation checks on user entries prior to submission or other calculations, many statements have the `form` object as part of the reference to the element.

A complex HTML document can have multiple `form` objects. Each `<form>...</form>` tag pair defines one form. You don't receive any penalties (except for potential confusion on the part of

someone reading your script) if you reuse a name for an element in each of a document's forms. For example, if each of three forms has a grouping of radio buttons with the name "choice," the object reference to each button ensures that JavaScript doesn't confuse them. The reference to the first button of each of those button groups is as follows:

```
document.forms[0].choice[0]
document.forms[1].choice[0]
document.forms[2].choice[0]
```

If you assign identifiers to `id` attributes, however, you should not reuse an identifier on the same page.

Passing forms and elements to functions

When a form or form element contains an event handler that calls a function defined elsewhere in the document, you can use a couple of shortcuts to simplify the task of addressing the objects while the function does its work. Failure to grasp this concept not only causes you to write more code than you have to, but also to get hopelessly lost you when you try to trace somebody else's code in his or her JavaScripted document. The watchword in event handler parameters is

```
this
```

which represents a reference to the current object that contains the event handler attribute. For example, consider the function and form definition in Listing 34-1. The entire user interface for this listing consists of form elements, as shown in Figure 34-1.

LISTING 34-1

Passing the form Object as a Parameter

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Beatle Picker</title>
      <script type="text/javascript">
      function processData(form)
      {
         for (var i = 0; i < form.Beatles.length; i++)
         {
            if (form.Beatles[i].checked)
            {
               break;
            }
         }
         var chosenBeatle = form.Beatles[i].value;
         var chosenSong = form.song.value;
         alert("Looking to see if " + chosenSong + " was written by " +
```

```
            chosenBeatle + "...");
      }

      function checkSong(songTitle)
      {
         var enteredSong = songTitle.value;
         alert("Making sure that " + enteredSong +
            " was recorded by the Beatles.");
      }
      </script>
   </head>
   <body>
      <form name="Abbey Road">
         Choose your favorite Beatle:
         <input type="radio" name="Beatles" id="Beatles1"
                value="John Lennon" checked="true" />John
         <input type="radio" name="Beatles" id="Beatles2"
                value="Paul McCartney" />Paul
         <input type="radio" name="Beatles" id="Beatles3"
                value="George Harrison" />George
         <input type="radio" name="Beatles" id="Beatles4"
                value="Ringo Starr" />Ringo
         <p>Enter the name of your favorite Beatles song:<br />
            <input type="text" name="song" id="song" value="Eleanor Rigby"
                   onchange="checkSong(this)" />
         </p>
         <p><input type="button" name="process" id="process"
                   value="Process Request..." onclick="processData(this.form)" />
         </p>
      </form>
   </body>
</html>
```

Note

The property assignment event handling technique in the previous example is a deliberate simplification to make the code more readable. It is generally better to use the more modern approach of binding events using the `addEventListener()` (NN6+/Moz/W3C) or `attachEvent()` (IE5+) methods. A modern cross-browser event handling technique is explained in detail in Chapter 32. Several other chapters use the modern technique extensively. ■

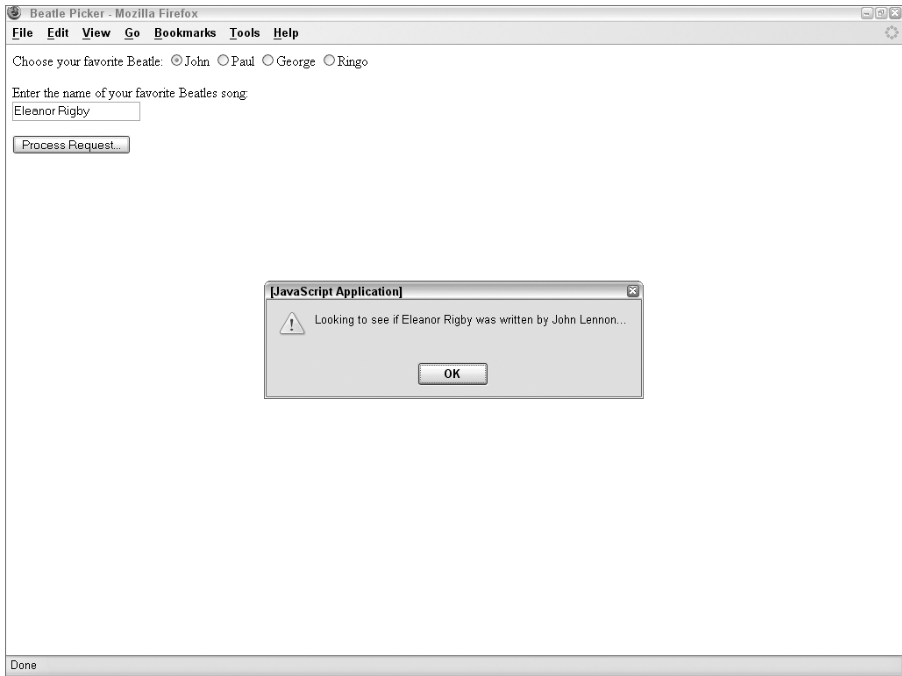
The `processData()` function, which needs to read and write properties of multiple form control elements, can reference the controls in two ways. One way is to have the `onclick` event handler (in the button element at the bottom of the document) call the `processData()` function, and not pass any parameters. Inside the function, all references to objects (such as the radio buttons or the song field) must be complete references, as in

```
document.forms[0].song.value
```

to retrieve the value entered into the `song` field.

FIGURE 34-1

Controls pass different object references to functions in Listing 34-1.



A more efficient way is to send a reference to the `form` object as a parameter with the call to the function (as shown in Listing 34-1). By specifying `this.form` as the parameter, you tell JavaScript to send along everything it knows about the form from which the function is called. This works because `form` is a property of every form control element; the property is a reference to the form that contains the control. Therefore, `this.form` passes the value of the `form` property of the control.

At the function, the reference to the `form` object is assigned to a variable name (arbitrarily set to `form` here) that appears in parentheses after the function name. We use the parameter variable name `form` here because it represents an entire form. But you can use any valid variable name you like.

The reference to the form contains everything the browser needs to know to find that form within the document. Any statements in the function can therefore use the parameter value in place of the longer, more cumbersome reference to the form. Thus, here we can use `form` to take the place of `document.forms[0]` or `document.forms["Beatles"]` in any address. To get the value of the `song` field, the reference is:

```
form.song.value
```

Had we assigned the `form` object to a parameter variable called `sylvester`, the reference would have been:

```
sylvester.song.value
```

When a function parameter is a reference to an object, statements in the function can retrieve or set properties of that object, as well as invoke the object's methods.

Another version of the `this` parameter-passing style simply uses the word `this` as the parameter. Unlike `this.form`, which passes a reference to the entire form connected to a particular element, `this` passes a reference only to that one element. In Listing 34-1, you can add an event handler to the `song` field to do some validation of the entry (to make sure that the entry appears in a database array of Beatles' songs created elsewhere in the document). Therefore, you want to send only the `field` object to the function for analysis:

```
<input type="text" name="song" id="song" onchange="checkSong(this)" />
```

You then have to create a function to catch this call:

```
function checkSong(songTitle)
{
   var enteredSong = songTitle.value;
   alert("Making sure that " + enteredSong + " was recorded by the Beatles.");
}
```

Within this function, you can go straight to the heart — the `value` property of the field element — without a long reference.

One further extension of this methodology passes only a single property of a form control element as a parameter. In the last example, the `checkSong()` function needs only the `value` property of the field, so the event handler can pass `this.value` as a parameter. Because `this` refers to the very object in which the event handler appears, the `this.propertyName` syntax enables you to extract and pass along a single property:

```
<input type="text" name="song" id="song" onchange="checkSong(this.value)" />
```

A benefit of this way of passing form element data is that the function doesn't have to do as much work:

```
function checkSong(songTitle)
{
   alert("Making sure that " + songTitle + " was recorded by the Beatles.");
}
```

Unlike passing object references (like the form and text field objects above), passing property value (for example, `this.value`) passes the value with no reference to the object from which it came. This suffices when the function just needs the value to do its job. However, if part of that job is to modify the object's property (for example, converting all text from a field to uppercase and redisplaying the converted text), the value passed to the function does not maintain a "live" connection with its object. To modify a property of the object that invokes an event handler

function, you need to pass some object reference so that the function knows where to go to work on the object.

Tip

Many programmers with experience in other languages expect parameters to be passed either by reference or by value, but not both ways. The rule of thumb in JavaScript, however, is fairly simple: object references are passed by reference; property values are passed by value. ■

Here are some guidelines to follow when deciding what kind of value to pass to an event handler function:

- Pass the entire form control object (`this`) when the function needs to make subsequent access to that same element (perhaps reading an object's `value` property, converting the value to all uppercase letters, and then writing the result back to the same object's `value` property).
- Pass only one property (`this.propertyName`) when the function needs read-only access to that property.
- Pass the entire `form` element object (`this.form`) for the function to access multiple elements inside a form (for example, a button click means that the function must retrieve a field's content). Remember, too, that control objects all have a `form` property, which is a reference to the containing `form` object. Therefore, if you pass only `this`, the function can still obtain the form reference.

Also be aware that you can submit multiple parameters (for example, `onclick="someFunction(this.form, this.name)"`) or even an entirely different object from the same form (for example, `onclick="someFunction(this.form.emailAddr.value)"`). Simply adjust your function's incoming parameters accordingly. (See Chapter 23 for more details about custom functions.)

Emailing forms

A common request among scripters is how to send a form via email to the page's author. This includes the occasional desire to send "secret" email to the author whenever someone visits the web site. Let us address the privacy issue first.

A site visitor's email address is valuable personal information that you should not retrieve without the visitor's permission or knowledge. Early browser versions employed various approaches to help safeguard email addresses. Some were more effective than others. Due to the unreliable nature and occasionally awkward user interface of mailing a form via the `mailto:` URL, we do not recommend its use.

Many ISPs that host web sites provide standard CGIs for forwarding forms to an email address of your choice. Search the Web for "formmail service" to locate third-party suppliers of this feature if you don't have access to server programming for yourself.

The remaining discussion about mailing forms focuses primarily on modern browsers and assumes that an ideally-configured email program is installed. You should be aware that mailing forms in the following ways is controversial in some web standards circles, since making an assumption about what the user has installed is somewhat of a leap of faith. Consequently, the W3C HTML specification does not endorse these techniques specifically. Use these facilities judiciously, and only after extensive testing on the client browsers you intend to support.

If you want to have forms submitted as email messages, you must attend to three `<form>` tag attributes. The first is the `method` attribute. You must set it to `post`. Next comes `enctype`. If you omit this attribute, the email client sends the form data as an attachment consisting of escaped name-value pairs, as in this example:

```
name=Danny+Goodman&rank=Scripter+First+Class&serialNumber=042
```

But, if you set the `enctype` attribute to text/plain, the form name-value pairs are placed in the body of the mail message in a more human-readable format:

```
name=Danny Goodman
rank=Scripter First Class
serialNumber=042
```

The last attribute of note is the `action` attribute, which is normally the spot to place a URL pointing to another file or server CGI. Substitute the URL with the special `mailto:` URL, followed by an optional parameter for the subject. Here is an example:

```
action="mailto:prez@whitehouse.gov?subject=Opinion%20Poll"
```

To sum up, the following example shows the complete `<form>` tag for emailing the form:

```
<form name="entry"
   method="post"
   enctype="text/plain"
   action="mailto:prez@whitehouse.gov?subject=Opinion Poll">
```

None of this requires any JavaScript at all. But seeing how you can use the attributes — and the fact that these attributes are exposed as properties of the `form` element object — you might see some extended possibilities for script control over forms.

Changing form attributes

All modern browsers expose `form` element attributes as modifiable properties. Therefore, you can change, say, the action of a form via a script in response to user interaction on your page. For example, you can have two different CGI programs invoked on your server depending on whether a form's check box is checked.

Tip

The best opportunity to change the properties of a `form` element object is in a function invoked by the form's `onsubmit` event handler. The modifications are performed at the last instant prior to actual submission, leaving no room for user-induced glitches to get in the way. ■

Buttons in forms

A common mistake that newcomers to scripting make is defining all clickable buttons as the submit type of `input` object (`<input type="submit">`). The Submit button does exactly what it says — it submits the form. If you don't set any `method` or `action` attributes of the `<form>` tag, the browser inserts its default values for you: `method="get"` and `action="pageURL"`. When you submit a form with these attributes, the page reloads itself and resets all field values to their initial values.

Use a Submit button only when you want the button to actually submit the form. If you want a button for other types of action, use the button style (`<input type="button">`). A regular button can invoke a function that performs some internal actions, and then invokes the `form` element object's `submit()` method to submit the form under script control.

Redirection after submission

Undoubtedly, you have submitted a form to a site and seen a "Thank You" page come back from the server to verify that your submission was accepted. This is warm and fuzzy, if not logical, feedback for the submission action. It is not surprising that you would want to re-create that effect, even if the submission is to a `mailto:` URL. Unfortunately, a problem gets in the way.

A common sense approach to the situation calls for a script to perform the submission (via the `form.submit()` method) and then navigate to another page that does the "Thank You." Here is such a scenario from inside a function triggered by clicking a link surrounding a nice, graphical Submit button:

```
function doSubmit()
{
   document.forms[0].submit();
   location.href = "thanks.html";
}
```

The problem is that when another statement executes immediately after the `form.submit()` method, the submission is canceled. In other words, the script does not wait for the submission to complete itself and verify to the browser that all is well (even though the browser appears to know how to track that information, given the status bar feedback during submission). The point is, because JavaScript does not provide an event that is triggered by a successful submission, there is no sure-fire way to display your own "Thank You" page.

Don't be tempted by the `window.setTimeout()` method to change the location after some number of milliseconds following the `form.submit()` method. You cannot predict how fast the network and/or server is for every visitor. If the submission does not complete before the timeout ends, the submission is still canceled — even if it is partially complete.

Form element arrays

Document object models provide a feature that is beneficial to a lot of scripters. If you create a series of like-named objects, they automatically become an array of objects accessible via array syntax (see Chapter 9). This is particularly helpful when you create forms with columns and rows of fields, such as in an order form. By assigning the same name to all fields in a column, you can employ `for` loops to cycle through each row, using the loop index as an array index.

As an example, the following code shows a typical function that calculates the total for an order form row (and calls another custom function to format the value):

```
function extendRows(form)
{
   for (var i = 0; i < Qty.length; i++)
   {
      var rowSum = form.Qty[i].value * form.Price[i].value;
      form.Total[i].value = formatNum(rowSum,2);
   }
}
```

All fields in the Qty column are named `Qty`. The item in the first row has an array index value of zero and is addressed as `form.Qty[i]`.

About <input> element objects

Whereas this chapter focuses strictly on the `form` element as a container of controls, the next three chapters discuss different types of controls that nest inside a form. Many of these controls share the same HTML tag: `<input>`. Only the `type` attribute of the `<input>` tag determines whether the browser shows you a clickable button, a check box, a text field, or other input field. The fact that one element has so many guises makes the system seem illogical at times.

An `input` element has some attributes (and corresponding scriptable object properties) that simply don't apply to every type of form control. For example, although the `maxLength` property of a text box makes perfect sense in limiting the number of characters that a user can type into it, the property has no bearing whatsoever on form controls that act as clickable buttons. Similarly, you can switch a radio button or check box on or off by adjusting the `checked` property; however, that property simply doesn't apply to a text box.

As the document object models have evolved, they have done so in an increasingly object-oriented way. The result in this form-oriented corner of the model is that all elements created via the `<input>` tag have a long list of characteristics that they all share by virtue of being types of `input` elements — they inherit the properties and methods that are defined for any `input` element. To try to limit the confusion, we divide the chapters in this book that deal

with `input` elements along functional lines (clickable buttons in one chapter, text fields in the other), and only list and discuss those `input` element properties and methods that apply to the specific control type.

In the meantime, this chapter continues with details of the `form` element object.

Properties

acceptCharset

Value: String — Read/Write

Compatibility: WinIE5+, MacIE5+, NN6+, Moz+, Safari+, Opera+, Chrome+

The `acceptCharset` property represents the `acceptcharset` attribute of the `form` element in HTML 4.0. The value is a list of one or more recognized character sets that the server receiving the form must support. For a list of registered character set names, see `http://www.iana.org/assignments/character-sets`.

Related Items: None

action

Value: URL string — Read/Write (see text)

Compatibility: WinIE3+, MacIE3+, NN2+, Moz+, Safari+, Opera+, Chrome+

The `action` property (along with the `method` and `target` properties) primarily functions for HTML authors whose pages communicate with server-based CGI scripts. This property is the same as the value you assign to the `action` attribute of a `<form>` tag. The value is typically a URL on the server where queries or postings are sent for submission.

User input may affect how you want your page to access a server. For example, a checked box in your document may set a form's `action` property so that a CGI script on one server handles all the input, whereas an unchecked box means the form data goes to a different CGI script, or a CGI script on an entirely different server. Or, one setting may direct the action to one `mailto:` address, whereas another setting sets the `action` property to a different `mailto:` address.

Although the specifications for all three related properties indicate that you can set them on the fly, such changes are ephemeral. A soft reload eradicates any settings you make to these properties, so you should make changes to these properties only in the same script function that submits the form (see `form.submit()` later in this chapter).

Related Items: `form.method`, `form.target`, `form.encoding` properties

autocomplete

Value: String — Read/Write

Compatibility: WinIE5+, MacIE-, NN-, Moz-, Safari-, Opera-, Chrome-

Microsoft added a feature to forms, starting with WinIE5, that allows the browser to supply hints for filling out form controls, if the controls' names map to a set of single-line text controls

defined via some additional attributes linked to the vCard XML schema. For details on implementing this browser feature, see `http://msdn.microsoft.com/library/default.asp?url=/workshop/author/dhtml/reference/properties/autocomplete.asp`. Values for the `autoComplete` property are your choice of two strings: `on` or `off`. In either case, the `form` element object does not report knowing about this property unless you set the `autocomplete` attribute in the form's tag.

Related Items: None

elements[]

Value: Array of form control elements Read-Only

Compatibility: WinIE3+, MacIE3+, NN2+, Moz+, Safari+, Opera+, Chrome+

Elements include all the user interface elements defined for a form: text fields, buttons, radio buttons, check boxes, selection lists, and more. The `elements` property is an array of all form control items defined within the current form. For example, if a form defines three `<input>` items, the `elements` property for that form is an array consisting of three entries (one for each item in source code order). Each entry is a valid reference to that element; so, to extract properties or call methods for those elements, your script must dig deeper in the reference. Therefore, if the first element of a form is a text field, and you want to extract the string currently showing in the field (a text element's `value` property), the reference looks like this:

```
document.forms[0].elements[0].value
```

Notice that this reference summons two array-oriented properties along the way: one for the document's `forms` property and one for the form's `elements` property.

In practice, we suggest you refer to form controls (and forms) by their names (or IDs if you prefer to use `document.getElementById()`). This allows you the flexibility to move controls around the page as you fine-tune the design, without worrying about the source code order of the controls. The `elements` array comes in handy when you need to iterate through all of the controls within a form. If your script needs to loop through all elements of a form in search of particular kinds of elements, use the `type` property of every `form` control object to identify which kind of object it is. The `type` property consists of the same string used in the `type` attribute of an `<input>` tag.

Overall, we prefer to generate meaningful names for each form control element and use those names in references throughout our scripts. The `elements` array helps with form control names as well. Instead of a numeric index to the `elements` array, you can use the string name of the control element as the index. Thus, you can create a generic function that processes any number of form control elements, and simply pass the string name of the control as a parameter to the function. Then, use that parameter as the `elements` array index value. For example:

```
function putVal(controlName, val)
{
   document.forms[0].elements[controlName].value = val;
}
```

***formObject*.elements[]**

If you want to modify the number of controls within a form, you should use the element and/or node management facilities of the browser(s) of your choice. For example, in modern browsers you can assemble the HTML string for an entirely new set of form controls, and then assign that string to the `innerHTML` property of the `form` element object.

Note

As handy as it may be, in a strict W3C approach to JavaScript, you wouldn't use the `innerHTML` property, since it isn't officially part of the W3C standard. However, it is often too powerful a convenience property to ignore, as much of the code throughout this book is a testament. The book does show the W3C node manipulation alternative to `innerHTML` in some examples. Refer to Chapter 29 for a thorough explanation and examples of the W3C alternative to `innerHTML`. ■

The document in Listing 34-2 demonstrates a practical use of the `elements` property. A form contains four fields and some other elements mixed in between (see Figure 34-2). The first part of the function that acts on these items repeats through all the elements in the form to find out which ones are text box objects, and which text box objects are empty. Notice how we use the `type` property to separate text box objects from the rest, even when radio buttons appear amid the fields. If one field has nothing in it, we alert the user and use that same index value to place the insertion point in the field with the field's `focus()` method.

LISTING 34-2

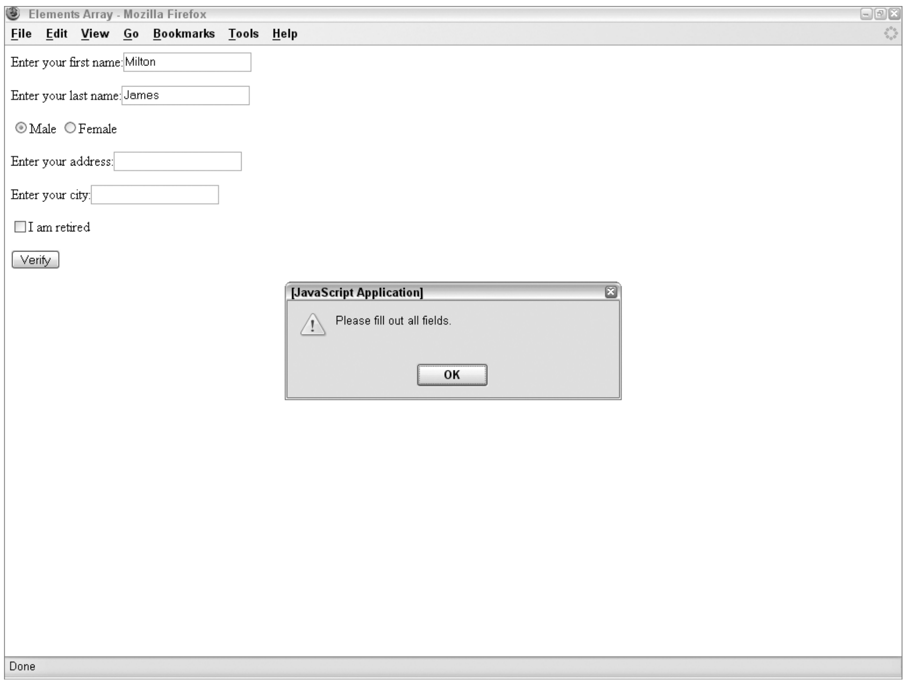
Using the form.elements Array

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Elements Array</title>
      <script type="text/javascript">
      function verifyIt()
      {
         var form = document.forms[0];
         for (i = 0; i < form.elements.length; i++)
         {
            if (form.elements[i].type == "text"
                && form.elements[i].value == "")
            {
               alert("Please fill out all fields.");
               form.elements[i].focus();
               break;
            }
            // more tests
         }
         // more statements
      }
      </script>
   </head>
   <body>
    <form>
```

```
        Enter your first name:<input type="text" name="firstName"
                                     id="firstName" />
        <p>Enter your last name:<input type="text" name="lastName"
                                       id="lastName" />
        </p>
        <p><input type="radio" name="gender" id="gender1" />Male
           <input type="radio" name="gender" id="gender2" />Female
        </p>
        <p>Enter your address:<input type="text" name="address" id="address"/></p>
        <p>Enter your city:<input type="text" name="city" id="city" /></p>
        <p><input type="checkbox" name="retired" id="retired" />I am
           retired
        </p>
      </form>
      <form>
         <input type="button" name="act" id="act" value="Verify"
                onclick="verifyIt()" />
       </form>
    </body>
</html>
```

FIGURE 34-2

The elements array helps find text fields for validation.



Related Items: `text`, `textarea`, `button`, `radio`, `checkbox`, `select` objects

encoding
enctype

Value: MIME type string Read/Write (see text)

Compatibility: WinIE6+, MacIE5+, NN6+, Moz+, Safari+, Opera+, Chrome+

You can define a form to alert a server when the data you submit is in a MIME type. The `encoding` property reflects the setting of the `enctype` attribute in the form definition. The `enctype` property name is defined for `form` element objects in the W3C DOM (with `encoding` removed), but NN6+ provides both properties for backward and forward compatibility.

For `mailto:` URLs, we recommend setting this value (in the tag or via script) to `"text/plain"` to have the form contents placed in the mail message body. If the definition does not have an `enctype` attribute, this property is an empty string.

Related Items: `form.action`, `form.method` properties

length

Value: Integer Read-Only

Compatibility: WinIE3+, MacIE3+, NN2+, Moz+, Safari+, Opera+, Chrome+

The `length` property of a `form` element object provides the same information as the `length` property of the form's `elements` array. The property provides a convenient, if not entirely logical, shortcut to retrieving the number of controls in a form.

Related Items: `form.elements` property

method

Value: String (`get` or `post`) Read/Write (see text)

Compatibility: WinIE3+, MacIE3+, NN2+, Moz+, Safari+, Opera+, Chrome+

A form's `method` property is either the `get` or `post` value (not case-sensitive) assigned to the `method` attribute in a `<form>` definition. Terminology overlaps here a bit, so be careful to distinguish a form's method of transferring its data to a server from the object-oriented method (action or function) that all JavaScript forms have.

The `method` property is of primary importance to HTML documents that submit a form's data to a server-based CGI script, because it determines the format used to convey this information. For example, to submit a form to a `mailto:` URL, the `method` property must be `post`. Details of forms posting and CGI processing are beyond the scope of this book. Consult HTML or CGI documentation to determine which is the appropriate setting for this attribute in your web server environment. If a form does not have a `method` attribute explicitly defined for it, the default value is `get`.

Related Items: `form.action`, `form.target`, `form.encoding` properties

name

Value: Identifier string Read/Write

Compatibility: WinIE3+, MacIE3+, NN2+, Moz+, Safari+, Opera+, Chrome+

Assigning a name to a form via the `name` attribute is optional, but highly recommended when your scripts need to reference a form or its elements. This attribute's value is retrievable as the `name` property of a form. You don't have much need to read this property unless you inspect another source's document for its form construction, as in:

```
var formName = parent.frameName.document.forms[0].name;
```

Moreover, because server-side CGI programs frequently rely on the name of the form for validation purposes, it is unlikely that you will need to change this property.

target

Value: Identifier string Read/Write (see text)

Compatibility: WinIE3+, MacIE3+, NN2+, Moz+, Safari+, Opera+, Chrome+

Whenever an HTML document submits a query to a server for processing, the server typically sends back an HTML page — whether it is a canned response or, more likely, a customized page based on the input provided by the user. You see this situation all the time when you perform a search on web sites. In a multiframe or multiwindow environment, you may want to keep the form part of this transaction in view for the user, but leave the responding page in a separate frame or window for viewing. The purpose of the `target` attribute of a `<form>` definition is to enable you to specify where the output from the server's query should be displayed.

The value of the `target` property is the name of the window or frame. For instance, if you define a frameset with three frames and assign the names `Frame1`, `Frame2`, and `Frame3` to them, you need to supply one of these names (as a quoted string) as the parameter of the `target` attribute of the `<form>` definition. Browsers also observe four special window names that you can use in the `<form>` definition: `_top`, `_parent`, `_self`, and `_blank`. To set the target as a separate subwindow opened via a script, use the window name from the `window.open()` method's second parameter, and not the window object reference that the method returns.

If you code your page to validate according to strict XHTML, you won't be able to include a `target` attribute for a form. But you can still use a script to assign a value to the property without interfering with the validation.

Related Items: `form.action`, `form.method`, `form.encoding` properties

Methods

reset()

Returns: Nothing

Compatibility: WinIE4+, MacIE4+, NN3+, Moz+, Safari+, Opera+, Chrome+

***formObject*.reset()**

A common practice, especially with a long form, is to provide a button that enables the user to return all the form elements to their default settings. The standard Reset button (a separate object type described in Chapter 34) does that task just fine. But if you want to clear the form using script control, you must do so by invoking the `reset()` method for the form. More than likely, such a call is initiated from outside the form, perhaps from a function or graphical button. In such cases, make sure that the reference to the `reset()` method includes the complete reference to the form you want to reset — even if the page only has one form defined for it.

In Listing 34-3, we assign the act of resetting the form to the `href` attribute of a link object (that is attached to a graphic called `reset.jpg`). We use the `javascript:` URL to invoke the `reset()` method for the form directly (in other words, without doing it via a function). Note that the form's action in this example is to a nonexistent URL. If you click the Submit icon, you receive an "unable to locate" error from the browser.

LISTING 34-3

form.reset() and form.submit() Methods

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Registration Form</title>
   </head>
   <body>
    <form name="entries" method="post"
          action="http://www.u.edu/pub/cgi-bin/register">
      Enter your first name:<input type="text" name="firstName"
                             id="firstName" />
      <p>Enter your last name:<input type="text" name="lastName"
                               id="lastName" />
      </p>
      <p>Enter your address:<input type="text" name="address"
                             id="address" />
      </p>
      <p>Enter your city:<input type="text" name="city" id="city" /></p>
      <p><input type="radio" name="gender" id="gender1" checked="checked" />Male
         <input type="radio" name="gender" id="gender2" />Female
      </p>
      <p><input type="checkbox" name="retired" id="retired" />I am retired</p>
     </form>
     <p><a href="javascript:document.forms[0].submit()">
        <img alt="image" src="submit.jpg" height="25" width="100" border="0" />
      </a>
      <a href="javascript:document.forms[0].reset()">
        <img alt="image" src="reset.jpg" height="25" width="100" border="0" />
       </a>
    </p>
   </body>
</html>
```

Related Items: `onreset` event handler, `reset` object

submit()

Returns: Nothing

Compatibility: WinIE3+, MacIE3+, NN2+, Moz+, Safari+, Opera+, Chrome+

The most common way to send a form's data to a server's CGI program for processing is to have a user click a Submit button. The standard HTML Submit button is designed to send data from all named elements of a form according to the specifications listed in the `<form>` definition's attributes. But if you want to submit a form's data to a server automatically for the user, or you want to use a graphical button for submission, you can accomplish the submission with the `form.submit()` method.

Invoking this method is almost the same as a user clicking a form's Submit button (except that the `onsubmit` event handler is not triggered). Therefore, you may have an image on your page that is a graphical submission button. If that image is surrounded by a link object, you can capture a mouse click on that image, and trigger a function whose content includes a call to a form's `submit()` method (see Listing 34-3).

In a multiple-form HTML document, however, you must reference the proper form, either by name or according to its position in a `document.forms` array. Always make sure that the reference you specify in your script points to the desired form before you submit any data to a server.

As a security and privacy precaution for people visiting your site, JavaScript ignores all `submit()` methods whose associated form actions are set to a `mailto:` URL. Many web page designers would love to have secret email addresses captured from visitors. Because such a capture can be considered an invasion of privacy, that power has been disabled since early browser versions. You can, however, still use an explicit Submit button object to mail a form to yourself from the browser. (See the section "Emailing forms" earlier in this chapter.)

Because the `form.submit()` method does not trigger the form's `onsubmit` event handler, you must perform any presubmission processing and forms validation in the same script that ends with the `form.submit()` statement. You also do not want to interrupt the submission process after the script invokes the `form.submit()` method. Script statements inserted after one that invokes `form.submit()` — especially those that navigate to other pages or attempt a second submission — cause the first submission to cancel itself.

Related Item: `onsubmit` event handler

Event handlers

onreset

Compatibility: WinIE4+, MacIE4+, NN3+, Moz+, Safari+, Opera+, Chrome+

Immediately before a Reset button returns a form to its default settings, JavaScript sends a `reset` event to the form. By including an `onreset` event handler in the form definition, you can trap that event before the reset takes place.

A friendly way of using this feature is to provide a safety net for a user who accidentally clicks the Reset button after filling out a form. The event handler can run a function that asks the user to confirm the action.

The `onreset` event handler must evaluate to `return true` for the event to continue to the browser. This may remind you of the way `onmouseover` and `onmouseout` event handlers work for links and image areas. This requirement is far more useful here because your function can control whether the reset operation ultimately proceeds to conclusion.

Listing 34-4 demonstrates one way to prevent accidental form resets or submissions. Using standard Reset and Submit buttons as interface elements, the `<form>` object definition includes both event handlers. Each event handler calls its own function that offers a choice for users. Notice how each event handler includes the word `return` and takes advantage of the Boolean values that come back from the `confirm()` method dialog boxes in both functions.

LISTING 34-4

The Onreset and Onsubmit Event Handlers

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Submit and Reset Confirmation</title>
      <script type="text/javascript">
         function allowReset()
         {
            return window.confirm("Go ahead and clear the form?");
         }
         function allowSend()
         {
            return window.confirm("Go ahead and mail this info?");
         }
      </script>
   </head>
   <body>
     <form method="post" enctype="text/plain"
           action="mailto:trash4@dannyg.com" onreset="return allowReset()"
           onsubmit="return allowSend()">
      Enter your first name:<input type="text" name="firstName"
                                 id="firstName" />
      <p>Enter your last name:<input type="text" name="lastName"
                                   id="lastName" />
      </p>
      <p>Enter your address:<input type="text" name="address"
                                 id="address" />
      </p>
      <p>Enter your city:<input type="text" name="city" id="city" /></p>
      <p><input type="radio" name="gender" id="gender1" checked="checked" />Male
```

```
        <input type="radio" name="gender" id="gender2" />Female
      </p>
      <p><input type="checkbox" name="retired" id="retired" />I am retired</p>
      <p><input type="reset" /> <input type="submit" /></p>
    </form>
  </body>
</html>
```

onsubmit

Compatibility: WinIE3+, MacIE3+, NN2+, Moz+, Safari+, Opera+, Chrome+

No matter how a form's data is actually submitted (by a user clicking a Submit button or by a script invoking the `form.submit()` method), you may want your JavaScript-enabled HTML document to perform some data validation on the user input, especially with text fields, before the submission heads for the server. You have the option of doing such validation while the user enters data (see Chapter 46 on the CD-ROM), or in batch mode before sending the data to the server (or both). The place to trigger this last-ditch data validation is the form's `onsubmit` event handler. Note, however, that this event fires only from a genuine Submit type `<input>` element, and not from the form's `submit()` method.

When you define an `onsubmit` event handler as an attribute of a `<form>` definition, JavaScript sends the `submit` event to the form just before it dashes off the data to the server. Therefore, any script or function that is the parameter of the `onsubmit` attribute executes before the data is actually submitted. Note that this event handler fires only in response to a genuine Submit-style button, and not from a `form.submit()` method.

Any code executed for the `onsubmit` event handler must evaluate to an expression consisting of the word `return`, plus a Boolean value. If the Boolean value is `true`, the submission executes as usual; if the value is `false`, no submission is made. Therefore, if your script performs some validation prior to submitting data, make sure that the event handler calls that validation function as part of a return statement (as shown in Listing 34-4).

Even after your `onsubmit` event handler traps a submission, JavaScript's security mechanism can present additional alerts to the user, depending on the server location of the HTML document and the destination of the submission.

fieldset and legend Element Objects

For HTML element properties, methods, and event handlers, see Chapter 26.

Properties	Methods	Event Handlers
`align`		
`form`		

Syntax

Accessing `fieldset` or `legend` element object properties or methods:

```
(IE4+)      [window.]document.all.elemID.property | method([parameters])
(IE5+/W3C)  [window.]document.getElementById("elemID").property |
            method([parameters])
```

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

About these objects

The `fieldset` and `legend` elements go hand in hand to provide some visual context to a series of form controls within a form. Browsers that implement the `fieldset` element draw a rectangle around the document space occupied by the form controls nested inside the `fieldset` element. The rectangle renders the full width of the body, unless its width is controlled by appropriate stylesheet properties (for example, `width`). To that rectangle is added a text label that is assigned via the `legend` element nested inside the `fieldset` element. None of this HTML-controlled grouping is necessary if you design a page layout that already provides graphical elements to group the form controls together.

Nesting the elements properly is essential to obtaining the desired browser rendering. A typical HTML sequence looks like the following:

```
<form>
  <fieldset>
    <legend>Legend Text</legend>
    All your form controls and their labels go here.
  </fieldset>
</form>
```

You can have more than one `fieldset` element inside a form. Each set has a rectangle drawn around it. This can help organize a long form into more easily digestible blocks of controls for users — yet the single form retains its integrity for submission to the server.

A `fieldset` element acts like any HTML container with respect to stylesheets and the inheritance thereof. For example, if you set the `color` style property of a `fieldset` element, the color affects the text of elements nested within; however, the color of the border drawn by the browser is unaffected. Assigning a color to the `fieldset` style's `border-color` property colors just the border and not the textual content of nested elements.

Note that the content of the `legend` element can be any HTML. Alternatively, you can assign a distinctive stylesheet rule to the `legend` element. If your scripts need to modify the text of the legend, you can accomplish this in modern browsers with the `nodeValue` properties of HTML element objects.

Only two element-specific properties are assigned to this object pair. The first is the `align` property of the `legend` object. This property matches the capabilities of the `align` attribute for the element as specified in the HTML 4.0 recommendation (albeit the property is deprecated in favor

of stylesheet rules). MacIE5+ and WinIE5.5+ enable you to adjust this property on the fly (generally between your choices of “right” and “left”) to alter the location of the legend at the top of the fieldset rectangle.

Because these elements are children of a `form` element, it makes sense that the DOM Level 2 specification supplies the read-only `form` property to both of these objects. That property returns a reference to the `form` element object that encloses either element. The `form` property for the `fieldset` and `legend` objects is implemented in modern browsers.

label Element Object

For HTML element properties, methods, and event handlers, see Chapter 26.

Properties	Methods	Event Handlers
`form`		
`htmlFor`		

Syntax

Accessing `label` element object properties or methods:

```
(IE4+)      [window.]document.all.elemID.property | method([parameters])
(IE5+/W3C)  [window.]document.getElementById("elemID").property |
            method([parameters])
```

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

About this object

With the push in the HTML 4.0 specification to provide context-oriented tags for just about every bit of content on the page, the W3C HTML working group filled a gap with respect to text that usually hangs in front of, or immediately after, `input`, `select`, and `textarea` form control elements. You use these text chunks as labels for the items to describe the purpose of the control. The only `input` element that had an attribute for its label was the button input type. But even the newer `button` element did away with that.

A `label` element enables you to surround a control’s label text with a contextual tag. In addition, one of the element’s attributes — `for` — enables you to associate the label with a particular form control element. In the HTML, the `for` attribute is assigned the ID of the control with which the label is associated. A `label` element can be associated with a form control if the form control’s tag is contained between the `label` element’s start and end tags.

At first glance, browsers do nothing special (from a rendering point of view) for a `label` element. But for some kinds of elements, especially check box and radio input type elements,

browsers help restore to users a vital user-interface convention: clicking the label is the same as clicking the control. For text elements, focus events are passed to the text input element associated with the label. In fact, all events that are directed at a label bubble upward to the form control associated with it. The following page fragment demonstrates how `fieldset`, `legend`, and `label` elements look in a form consisting of two radio buttons:

```
<form ...>
   <fieldset id="form1set1">
      <legend id="form1set1legend">Choose the Desired Performance</legend>
      <input type="radio" name="speed" id="speed1" />
         <label for="speed1">Fastest (lower quality)</label><br />
      <input type="radio" name="speed" id="speed2" />
         <label for="speed2">Slower (best quality)</label>
   </fieldset>
</form>
```

Even so, a `label` and its associated form control element do not have to be adjacent to each other in the source code. For example, you can have a label in one cell of a table row, with the form control in another cell (in the same or different row).

Properties

htmlFor

Value: Element object reference — Read/Write

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

The `htmlFor` property is the scripted equivalent of the `for` attribute of the `label` element. An acceptable value is a full reference to a form control element (`input`, `textarea`, or `select` element objects). It is highly unlikely that you would modify this property for an existing `label` element. However, if your script is creating a new `label` element (perhaps a replacement form), use this property to associate the label with a form control.

Scripting and Web Forms 2.0

Standard HTML forms have been with us for some time, which means it shouldn't necessarily come as a surprise that there is an effort to revamp the handling of forms on the Web. Although there are several form technologies vying for the limelight, Web Forms 2.0 is the one with the most momentum, at least for now.

It's difficult to mention Web Forms 2.0 without also mentioning XForms, which is in many ways a competing form technology. Unlike Web Forms 2.0, which is based on traditional HTML code and scripting, XForms relies on a specialized XML syntax to describe form components. This doesn't necessarily make either technology better or worse than the other, except for the fact that Web Forms 2.0 is already supported by a production web browser (Opera 9). And, as history has shown, early adoption is one of the best ways for an emerging technology to take hold.

Speaking of browsers and forms, it's worth noting that Microsoft doesn't appear to be very eager to back either Web Forms 2.0 or XForms. They have their own XML-based form technology known as XAML that has ties to the next generation Windows operating system. The Mozilla Foundation was involved in developing the Web Forms 2.0 specification and has stated a commitment to supporting it natively in the Gecko browser engine that forms the core of Mozilla-based browsers. The same goes for Apple and its desire to support Web Forms 2.0 in its Safari browser. So, expect to see much broader support for Web Forms 2.0 in the very near future, and in the meantime there are plug-ins available for several major browsers.

What Is Web Forms 2.0?

Like HTML and JavaScript, Web Forms 2.0 is a specification that describes how a technology is supposed to work. In this case, the specification describes a set of rich user interface components that carry out common information gathering tasks such as allowing the user to select a date or enter text. Traditional HTML supports such tasks via forms as well, but those forms are fairly limited and require a decent amount of scripting in order to provide any significant degree of user friendliness. Thanks to modern user interface design, web users have come to expect form validation, an auto-completion feature for frequently-entered text, and visual user interfaces for common data entry tasks such as the selection of a date.

Web Forms 2.0 addresses the limitations of HTML forms by offering advanced form validation without scripting: auto-completion, careful control over the input focus, and a host of new form control types, such as specialized date/time and URL input controls, among others. You can begin using Web Forms 2.0 right away in the Opera browser (as of version 9); there are browser add-ons that support it in other major browsers.

Web Forms 2.0 and JavaScript

Scripting's role in Web Forms 2.0 is primarily that of a programmatic equivalent to operations and behaviors that are built into any browser that supports the standard. For example, Web Forms 2.0 includes a facility known as *repetition blocks*, whereby a form can grow or shrink as needed, such as a tabular order form whose rows of input elements grow as the user adds new items to the order. The Web Forms mechanism takes care of adding repeated rows of form controls and assigning names to elements indicating row numbers (there is a special button type whose tag attributes link it to the template that is repeated). The DOM interface provided with Web Forms allows script access to adding and subtracting rows if the page design requires it.

If it sounds as though Web Forms 2.0 is trying to displace JavaScript, in a way that's true. The purpose is to rely less on scripting for form validation and other Dynamic HTML surrounding forms, offering page authors a more standardized way of handling these common tasks. But the creators of Web Forms 2.0 also know that scripters will want to get their hands on the new forms "stuff," and they therefore provide ample script access.

For more information about Web Forms 2.0, visit `http://www.whatwg.org/specs/web-forms/current-work`. Web Forms 2.0 has recently been subsumed by the ongoing HTML5 working draft. This link will give you more information: `http://www.whatwg.org/specs/web-apps/current-work/multipage/`.

CHAPTER

35

Button Objects

IN THIS CHAPTER

Triggering action from a user's click of a button

Assigning hidden values to radio and checkbox buttons

Distinguishing between radio button families and their individual buttons

This chapter is devoted to those lovable buttons that invite users to initiate action and make choices with a single click of the mouse. In this category fall the standard system-looking buttons with labels on them, as well as radio buttons and checkboxes. For such workhorses of the HTML form, these objects have a limited vocabulary of object-specific properties, methods, and event handlers.

We group together the button, submit, and reset objects for an important reason: They look alike, yet they are intended for very different purposes. Knowing when to use which button is important — especially in choosing between the button and submit objects. Many a newcomer get the two confused and wind up with scripting error headaches. That confusion won't be an issue for you by the time you finish this chapter.

The button Element Object, and the Button, Submit, and Reset Input Objects

For HTML element properties, methods, and event handlers, see Chapter 26, "Generic HTML Element Objects".

Properties	Methods	Event Handlers
form	click()	onclick
name		onmousedown
type		onmouseup
value		

Syntax

Accessing button object properties or methods:

```
(All)       [window.]document.formName.buttonName.property |
            method([parameters])
(All)       [window.]document.formName.elements[index].property |
            method([parameters])
(All)       [window.]document.forms[index].buttonName.property |
            method([parameters])
(All)       [window.]document.forms["formName"].buttonName.property |
            method([parameters])
(All)       [window.]document.forms["formName"].elements[index].property |
            method([parameters])
(IE4+)      [window.]document.all.elemID.property | method([parameters])
(IE5+/W3C)  [window.]document.getElementById("elemID").property |
            method([parameters])
```

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

About these objects

Button objects generate standard, pushbutton-style user interface elements on the page, depending on the operating system on which the particular browser runs. In the early days, the browsers called upon the operating systems to generate these standard interface elements. In more recent versions, the browsers define their own look, albeit frequently still different for each operating system. More recently, the appearance of a button may also be influenced by browser-specific customizations that manufacturers put into their products. Even so, any computer user will recognize a button when the browser produces it on the page.

There are two ways to put standard buttons into a page. The first, and completely backward-compatible way, is to use `input` elements nested inside a `form` container. The modern approach involves the `button` HTML element, which provides a slightly different way of specifying a button in a page, including the option of putting a button outside of a `form` (presumably for some client-side script execution, independent of form submission). From an

HTML point of view, the difference between the two concerns itself with the way the label of the button is specified. With an `input` element, the string assigned to the `value` attribute becomes the label of the button; but a `button` element is a container (meaning it has an end tag), whose content becomes the button's label. You can still assign a value to the `value` attribute, which, if a form contains the button, gets submitted to the server, independent of the label text.

Always give careful thought to the label that you assign to a button. Because a button initiates some action, make sure that the verb in the label clearly defines what happens after you click it. Also, take cues from experienced user interface designers who craft operating system and commercial software buttons: be concise. If you find your button labels going longer than two or three words, reconsider the design of your page so that the user can clearly understand the purpose of any button from a shorter label.

Browsers automatically display a button sized to accommodate the label text. But only modern browsers (IE4+/Moz+/W3C) allow you to control more visual aspects of the button, such as size, label font, and coloration. As for the position on the page, buttons, like all in-line elements, appear where they occur in the source code. You can, of course, use element positioning (see Chapter 43, "Positioned Objects," on the CD-ROM) to make a button appear wherever you want it. But if your pages run on multiple operating systems and generations of browsers, be aware that the appearance (and size) of a button will not be identical on all screens. Check out the results on as many platforms as possible.

Buttons in the Windows environment follow normal behavior, in that they indicate the focus with highlighted button-label text (usually with a dotted rectangle). Some newer browsers running on other operating systems offer this kind of highlighting and selection as a user option. IE5+ provides additional `input` element features that prevent buttons from receiving this kind of visible focus.

The lone `button` object event handler that works on all browser versions is one that responds to a user clicking the pointer on the button: the `onclick` event handler. Virtually all action surrounding a `button` object comes from this event handler. You rarely need to extract property values or invoke the `click()` method. Modern browsers include support for the individual component events of a click: `mousedown` and `mouseup`; there's also a plethora of user-initiated events for buttons that you can use.

Two special variants of the `button` input object are the *submit* and *reset* input objects. With heritages going back to early incarnations of HTML, these two button types perform special operations on their own. The submit-style button automatically sends the data within the same form object to the URL listed in the `action` attribute of the `<form>` definition. The `method` attribute dictates the format in which the button sends the data. Therefore, you don't have to script this action if your HTML page is communicating with a program (often a CGI script) on the server.

If the form's `action` attribute is set to a `mailto:` URL, you must provide the page visitor with a Submit button to carry out the action. Setting the form's `enctype` attribute to `text/plain` is also helpful so that the form data arrives in a more readable form than the normal encoded name-value pairs. See Chapter 33, "Form and Related Objects," for details about submitting form content via email.

The partner of the Submit button is the Reset button. This button, too, has special powers. A click of this button type restores all elements within the form to their default values. That goes for text objects, radio button groups, checkboxes, and selection lists. The most common application of the button is clearing entry fields of any data entered by the user.

All that distinguishes these three types of buttons from each other in the `<input>` tag or `<button>` tag is the parameter of the `type` attribute. For buttons not intended to send data to a server, use the "button" style (this is the default value for the `button` element). You should reserve "submit" and "reset" for when you need their special powers.

If you want an image to behave like a button in all scriptable browsers, consider either associating a link with an image (see the discussion on the link object in Chapter 31, "Link and Anchor Objects") or creating a client-side image map (see the `area` object discussion in Chapter 32, "Image, Area, Map, and Canvas Objects"). An even better idea that applies solely to modern browsers is to use the `input` element with a `type` attribute set to `image` (discussed later in this chapter).

Probably the biggest mistake scripters make with these buttons is using a Submit button to do the work of a plain button. Because these two buttons look alike, and the submit type of input element has a longer tradition than the plain button, confusing the two is easy. But if all you want is to display a button that initiates client-side script execution, use a plain button. The Submit button attempts to submit the form. If no `action` attribute is set, then the page reloads, and all previous processing and field entries are erased. The plain button does its job quietly without reloading the page (unless the script it triggers intentionally does so).

Properties

`form`

Value: Form object reference Read-Only

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

A property of every `input` element object is a reference to the `form` element that contains the control. This property can be very convenient in a script when you are dealing with one form control that is passed as a parameter to the function, and you want to either access another control in the same form or invoke a method of the form. An event handler of any `input` element can pass `this` as the parameter, and the function can still get access to the form without having to hard-wire the script to a particular form name or document layout.

Related Items: `form` object

`name`

Value: Identifier string Read/Write (see text)

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

A button's name is fixed in the `input` or `button` element's `name` attribute and can be adjusted via scripting in modern browsers. You may need to retrieve this property in a general-purpose

function handler called by multiple buttons in a document. The function can test for a button name and perform the necessary statements for that button:

```
if (button.name == "Calculate")
{
   // Perform calculation
}
```

If you change the name of the object, even a soft reload or window resize restores its original name.

Related Items: `name` property of all `form` elements

type

Value: String Read-Only

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

The precise value of the `type` property echoes the setting of the `type` attribute of the `<input>` or `<button>` tag that defines the object: `button`, `submit`, or `reset`.

value

Value: String Read/Write (see text)

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

Both `input` and `button` elements have the `value` attribute, which is represented by the `value` property in the object model. But the purpose of the attribute/property in the two elements differs. For the `input` element, the `value` property represents the label displayed on the button. For a `button` element, however, the label text is created by the HTML text between the element's start and end tags. When the `input` element has a `name` value associated with it, the name-value pair is submitted along with the form.

If you do not assign a `value` attribute to a reset or submit style button, browsers automatically assign the labels `Reset` and `Submit` without assigning a value. A value property can be any string, including multiple words.

You can modify this text on the fly in a script. Modern browsers are smart enough to resize the button and reflow the page to meet the new space needs; the new label survives a window resizing, but not a soft reload of the page.

Related Items: `value` property of text object

Methods

click()

Returns: Nothing

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

A button's `click()` method simulates, via scripting, the human action of clicking that button, triggering a `click` event.

Related Items: `onclick` event handler

Event handlers

onclick

Compatibility: WinIE3+, MacIE3+, NN2+, Moz+, Safari+, Opera+, Chrome+

Virtually all button action takes place in response to the `onclick` event handler. A *click* is defined as a press and release of the mouse button while the screen pointer rests on the button. The event goes to the button only after the user releases the mouse button.

For a Submit button, you should probably omit the `onclick` event handler and allow the form's `onsubmit` event handler to take care of last-minute data entry validation before sending the form. By triggering validation with the `onsubmit` event handler, your scripts can cancel the submission if something is not right (see the `form` object discussion in Chapter 33).

Listing 35-1 demonstrates not only the `onclick` event handler of a button, but also how you may need to extract a particular button's `name` or `value` properties from a general-purpose function that services multiple buttons. In this case, each button passes its own object as a parameter to the `displayTeam()` function. The function then displays the results in an alert dialog box. A real-world application would probably perform more sophisticated actions based on the button clicked.

LISTING 35-1

Three Buttons Sharing One Function

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Button Click</title>
      <script type="text/javascript">
      function displayTeam(btn)
      {
         switch (btn.value)
         {
            case "Starsky":
               alert("Starsky & Hutch");
               break;
            case "Tango":
               alert("Tango & Cash");
               break;
            case "Turner":
               alert("Turner & Hooch");
               break;
```

continued

LISTING 35-1 *(continued)*

```
         }
      }
      </script>
   </head>
   <body>
      Click on your favorite half of a popular crime fighting team:
      <form>
         <input type="button" value="Starsky" onclick="displayTeam(this)" />
         <input type="button" value="Tango" onclick="displayTeam(this)" />
         <input type="button" value="Turner" onclick="displayTeam(this)" />
      </form>
   </body>
</html>
```

Note

The property assignment event handling technique used in the previous example and throughout the chapter is a deliberate simplification to make the code more readable. It is generally better to use the more modern approach of binding events using the `addEventListener()` (NN6+/Moz/W3C) or `attachEvent()` (IE5+) methods. A modern cross-browser event handling technique is explained in detail in Chapter 37, "Event Objects." Several other chapters use the modern technique extensively. ■

Related Items: `button.onmousedown`, `button.onmouseup`, `form.onsubmit` event handlers

onmousedown
onmouseup

Compatibility: WinIE4+, MacIE4+, NN4+, Moz+, Safari+, Opera+, Chrome+

Modern browsers have event handlers for the components of a `click` event: the `onmousedown` and `onmouseup` event handlers. These events fire in addition to the `onclick` event handler.

The system-level buttons provided by the operating system perform their change of appearance while a button is being pressed. Therefore, trapping for the components of a click action won't help you in changing the button's appearance via scripting. Remember that a user can roll the cursor off the button while the button is still down. When the cursor leaves the region of the button, the button's appearance returns to its unpressed look, but any setting you make with the `onmousedown` event handler won't undo itself with an `onmouseup` counterpart, even after the user releases the mouse button elsewhere. On the other hand, you can use these events to fire pre-cached click-on and click-off sounds in response to the respective mouse button actions.

Related Items: `button.onclick` event handler

checkbox Input Object

For HTML element properties, methods, and event handlers, see Chapter 26, "Generic HTML Element Objects."

Properties	Methods	Event Handlers
`checked`	`click()`†	`onclick`†
`form`†		
`name`†		
`type`		
`value`		

†See Button object.

Syntax

Accessing checkbox properties or methods:

```
(All)        [window.]document.formName.boxName.property |
             method([parameters])
(All)        [window.]document.formName.elements[index].property |
             method([parameters])
(All)        [window.]document.forms[index].boxName.property |
             method([parameters])
(All)        [window.]document.forms["formName"].boxName.property |
             method([parameters])
(All)        [window.]document.forms["formName"].elements[index].property |
             method([parameters])
(IE4+)       [window.]document.all.elemID.property | method([parameters])
(IE5+/W3C)   [window.]document.getElementById("elemID").property |
             method([parameters])
```

Compatibility: WinIE3+, MacIE3+, NN2+, Moz+, Safari+, Opera+, Chrome+

About this object

Checkboxes have a very specific purpose in modern graphical user interfaces: to toggle between "on" and "off" settings. As with a checkbox on a printed form, a mark in the box indicates that the label text is true, or should be included for the individual who made that mark. When the box is unchecked or empty, the text is false or should not be included. If two or more check-boxes are physically grouped together, they should have no interaction: Each is an independent setting (see the discussion on the radio object for interrelated buttons).

We make these user-interface points at the outset because, in order to present a user interface in your HTML pages consistent with the user's expectations based on exposure to other programs,

you must use checkbox objects only for on/off choices that the user makes. Using a checkbox as an action button that, for example, navigates to another URL, is not good form. Just as they do in a Windows or Mac dialog box, users make settings with checkboxes and radio buttons, and initiate action by clicking a standard button or image map.

That's not to say that a checkbox object cannot perform some limited action in response to a user's click, but such actions are typically related to the context of the checkbox button's label text. For example, in some Windows and Macintosh dialog boxes, turning on a checkbox may activate a bunch of otherwise inactive settings elsewhere in the same dialog box. Modern browsers allow disabling (dimming) or hiding of form elements, so a checkbox may control those visible attributes of related controls. Or, in a two-frame window, a checkbox in one frame may control whether the viewer is an advanced user. If so, the content in the other frame may be more detailed. Toggling the checkbox changes the complexity level of a document showing in the other frame (using different URLs for each level). The bottom line, then, is that you should use checkboxes for toggling between on/off settings. Provide regular buttons for users to initiate processing.

In the `<input>` tag for a checkbox, you can preset the checkbox to be checked when the page appears by adding the constant `checked` attribute to the definition. If you omit this attribute, the button takes on its default, unchecked appearance. As for the checkbox label text, its definition lies outside the `<input>` tag, usually as text that appears next to the tag. If you look at the way checkboxes behave in HTML browsers, this location makes sense: The label is not an active part of the checkbox (as it typically is in Windows and Macintosh user interfaces, where clicking the label is the same as clicking the box).

Naming a checkbox can be an important part of the object definition, depending on how you plan to use the information in your script or document. For forms whose content goes to a program running on the server, you must word the box name as needed for use by the server program so that the program can parse the form data and extract the setting of the checkbox. For JavaScript client-side use, you can assign not only a name that describes the button, but also a value useful to your script for making `if...else` decisions, or for assembling strings that are eventually displayed in a window or frame.

Properties

checked

Value: Boolean Read/Write

Compatibility: WinIE3+, MacIE3+, NN2+, Moz+, Safari+, Opera+, Chrome+

The simplest property of a checkbox reveals (or lets you set) whether or not a checkbox is checked. The value is `true` for a checked box and `false` for an unchecked box. To check a box via a script, simply assign `true` to the checkbox's `checked` property:

```
document.forms[0].boxName.checked = true;
```

Setting the `checked` property from a script does not trigger a `click` event for the checkbox object. So, the `onclick` event handler won't get called in response to a checkbox being checked via the `checked` property.

You may need an instance in which one checkbox automatically checks or unchecks another checkbox elsewhere in the same form (or in another form) of the document. To accomplish this task, create an `onclick` event handler for the one checkbox and build a statement, similar to the preceding one, to set the other related checkbox to `true`. Don't get too carried away with this feature, however: For a group of interrelated, mutually exclusive choices, use a group of radio buttons instead.

If your page design requires that a checkbox be checked after the page loads, don't bother trying to script this checking action. Simply add the one-word `checked` attribute to the `<input>` tag. Because the `checked` property is a Boolean value, you can use its results as an argument for an `if` clause, as shown in the next example.

The simple example in Listing 35-2 passes a form object reference to the JavaScript function. The function, in turn, reads the `checked` value of the form's checkbox object (`checkThis.checked`) and uses its Boolean value as the test result for the `if...else` construction.

LISTING 35-2

The checked Property as a Conditional

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Checkbox Inspector</title>
      <script type="text/javascript">
         function inspectBox(form)
         {
            if (form.checkThis.checked)
            {
               alert("The box is checked.");
            }
            else
            {
               alert("The box is not checked at the moment.");
            }
         }
      </script>
   </head>
   <body>
      <form>
         <input type="checkbox" name="checkThis" />Check here
         <p><input type="button" name="boxChecker" value="Inspect Box"
                  onclick="inspectBox(this.form)" />
         </p>
      </form>
   </body>
</html>
```

Related Items: `defaultChecked`, `value` properties

defaultChecked

Value: Boolean Read-Only

Compatibility: WinIE3+, MacIE3+, NN2+, Moz+, Safari+, Opera+, Chrome+

Sometimes you may find it beneficial to know if the initial setting of a checkbox has changed. The `checked` property alone can't tell you this because it reflects only the current state of a checkbox. Another property, `defaultChecked`, keeps up with the initial state of a checkbox.

If you add the `checked` attribute to the `<input>` definition for a checkbox, the `defaultChecked` property for that object is `true`; otherwise, the property is `false`. Having access to this property enables your scripts to examine checkboxes to see if they have been adjusted (presumably by the user, if your script does not set properties).

The following function is designed to compare the current setting of a checkbox against its default value:

```
function compareBrowser(thisBox)
{
   if (thisBox.checked != thisBox.defaultChecked)
   {
      // statements about using a different set of HTML pages
   }
}
```

The `if` construction compares the current status of the box against its default status. Both are Boolean values, so they can be compared against each other. If the current and default settings don't match, the function goes on to handle the case in which the current setting is other than the default.

Related Items: `checked`, `value` properties

type

Value: String (`checkbox`) Read-Only

Compatibility: WinIE4+, MacIE4+, NN3+, Moz+, Safari+, Opera+, Chrome+

Use the `type` property to help you identify a checkbox object from an unknown group of form elements. Just look for the string `checkbox` as the type of a form element to determine if it is indeed a checkbox.

Related Items: `form.elements` property

value

Value: String Read/Write

Compatibility: WinIE3+, MacIE3+, NN2+, Moz+, Safari+, Opera+, Chrome+

A checkbox object's `value` property is a string of any text that you want to associate with the box. Note that the checkbox's `value` property is not the label, as it is for a regular button, but hidden text associated with the checkbox. For instance, the label that you attach to a checkbox may not be worded in a way that is useful to your script. But if you place useful wording in the `value` attribute of the checkbox tag, you can extract that string via the `value` property.

When a checkbox object's data is submitted to a CGI program, the `value` property is sent as part of the name-value pair if the box is checked (nothing about the checkbox is sent if the box is unchecked). If you omit the `value` attribute in your definition, the property always yields the string "on," which is submitted to a CGI program when the box is checked. From the JavaScript side, don't confuse this string with the on and off settings of the checkbox: Use the `checked` property to determine a checkbox's status.

The scenario for the skeleton HTML page in Listing 35-3 is a form with a checkbox whose selection determines which of two actions to follow for submission to the server. After the user clicks the Submit button, a JavaScript function examines the checkbox's `checked` property. If the property is `true` (the button is checked), the script sets the `action` property for the entire form to the content of the value property — thus influencing where the form goes on the server side. If you try this listing on your computer, the result you see varies widely with the browser version you use. For most browsers, you see some indication (an error alert or other screen notation) that a file with the name `primaryURL` or `alternateURL` doesn't exist. The names and the error message come from the submission process for this demonstration.

LISTING 35-3

Adjusting a Server Submission Action

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Checkbox Submission</title>
      <script type="text/javascript">
         function setAction(form)
         {
            if (form.checkThis.checked)
            {
               form.action = form.checkThis.value;
            }
            else
            {
               form.action = "file://primaryURL";
            }
            return true;
         }
      </script>
   </head>
   <body>
      <form method="POST" action="">
         <input type="checkbox" name="checkThis"
```

continued

LISTING 35-3 *(continued)*

```
            value="file://alternateURL" />Use alternate
        <p><input type="submit" name="boxChecker"
            onclick="return setAction(this.form)" /></p>
      </form>
   </body>
</html>
```

Related Items: `checked` property

Methods

`click()`

Returns: Nothing

Compatibility: WinIE3+, MacIE3+, NN2+, Moz+, Safari+, Opera+, Chrome+

The intention of the `click()` method is to enact, via script, the physical act of clicking a checkbox (but without triggering the `onclick` event handler). However, your scripts are better served by setting the `checked` property so that you know exactly what the setting of the box is at any time.

Related Items: `checked` property; `onclick` event handler

Event handlers

`onclick`

Compatibility: WinIE3+, MacIE3+, NN2+, Moz+, Safari+, Opera+, Chrome+

Because users regularly click checkboxes, the objects have an event handler for the `click` event. Use this event handler only if you want your page (or variable values hidden from view) to respond in some way to the action of clicking a checkbox. Most user actions, as mentioned earlier, are initiated by clicking standard buttons rather than checkboxes, so be careful not to overuse event handlers in checkboxes.

The page in Listing 35-4 shows how to trap the click event in one checkbox to influence the visibility and display of other form controls. After you turn on the Monitor checkbox, a list of radio buttons for monitor sizes appears. Similarly, engaging the Communications checkbox makes two radio buttons visible. Your choice of radio button brings up one of two further choices within the same table cell (see Figure 35-1).

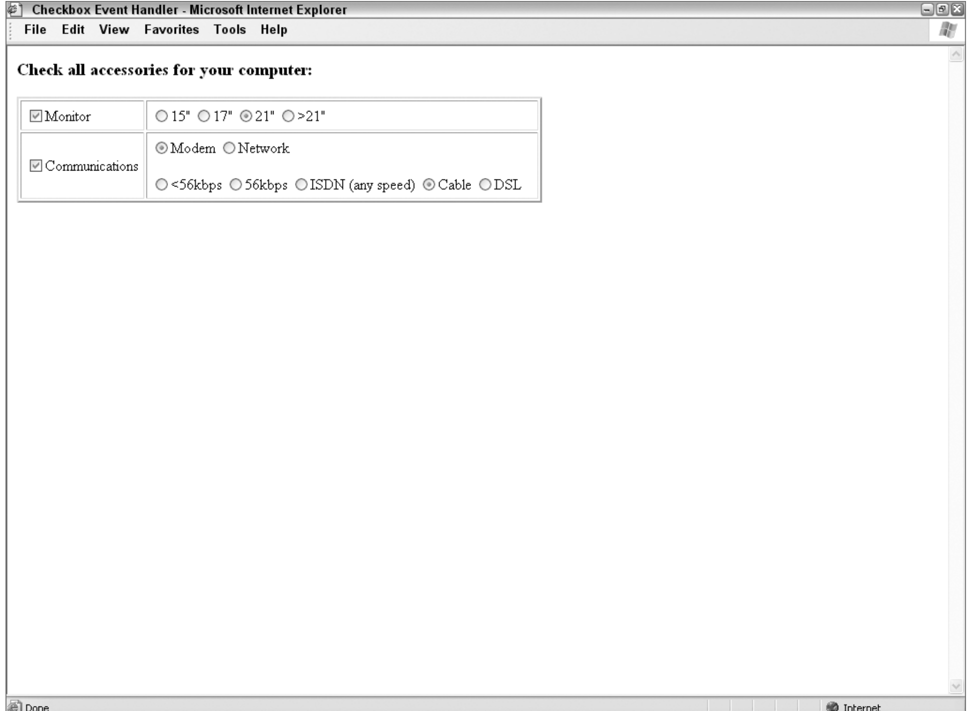
Notice how the `toggle()` function was written as a generalizable function. This function can accept a reference to any checkbox object and any related span. If five more groups like this were added to the table, no additional functions would be needed.

In the `swap()` function, a nested `if...else` shortcut construction is used to convert the Boolean values of the `checked` property to the strings needed for the `display` style property.

The nesting is used to allow a single statement to take care of two conditions: the group of buttons to be controlled and the `checked` property of the button invoking the function. This function is not generalizable, because it contains explicit references to objects in the document. The `swap()` function can be made generalizable, but due to the special relationships between pairs of span elements (one has to be hidden while the other is displayed in its place), the function would require more parameters to fill in the blanks where explicit references are needed.

FIGURE 35-1

Clicking each checkbox reveals additional relevant choices.



LISTING 35-4

A Checkbox and an onclick Event Handler

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Checkbox Event Handler</title>
      <style type="text/css">
```

continued

LISTING 35-4 *(continued)*

```
         #monGroup {visibility:hidden}
         #comGroup {visibility:hidden}
      </style>
      <script type="text/javascript">
         // toggle visibility of a main group spans
         function toggle(chkbox, group)
         {
            var visSetting = (chkbox.checked) ? "visible" : "hidden";
            document.getElementById(group).style.visibility = visSetting;
         }
         // swap display of communications sub group spans
         function swap(radBtn, group)
         {
            var modemsVisSetting = (group == "modems") ? ((radBtn.checked)
               ? "" : "none") : "none";
            var netwksVisSetting = (group == "netwks") ? ((radBtn.checked)
               ? "" : "none") : "none";
            document.getElementById("modems").style.display = modemsVisSetting;
            document.getElementById("netwks").style.display = netwksVisSetting;
         }
      </script>
   </head>
   <body>
      <form>
         <h3>Check all accessories for your computer:</h3>
         <table border="2" cellpadding="5">
            <tr>
               <td><input type="checkbox" name="monitor"
                          onclick="toggle(this, 'monGroup')" />Monitor</td>
               <td>
                  <span id="monGroup">
                     <input type="radio" name="monitorType" />15"
                     <input type="radio" name="monitorType" />17"
                     <input type="radio" name="monitorType" />21"
                     <input type="radio" name="monitorType" />&gt;21"
                  </span>
               </td>
            </tr>
            <tr>
               <td>
                  <input type="checkbox" name="comms" onclick="toggle(this,
                   'comGroup')" />Communications
               </td>
               <td>
                  <span id="comGroup">
                     <p>
                        <input type="radio" name="commType" onclick="swap(this,
                         'modems')" />Modem
                        <input type="radio" name="commType" onclick="swap(this,
                         'netwks')" />Network
```

```
                </p>
                <p>
                    <span id="modems" style="display:none">
                        <input type="radio" name="modemType" />&lt;56kbps
                        <input type="radio" name="modemType" />56kbps
                        <input type="radio" name="modemType" />ISDN (any speed)
                        <input type="radio" name="modemType" />Cable
                        <input type="radio" name="modemType" />DSL
                    </span>
                    <span id="netwks" style="display:none">
                        <input type="radio" name="netwkType" />Ethernet 10Mbps
                         (10-Base T)
                        <input type="radio" name="netwkType" />Ethernet 100Mbps
                         (10/100)
                        <input type="radio" name="netwkType" />T1 or greater
                    </span>
                     
                </p>
            </span>
          </td>
        </tr>
      </table>
    </form>
  </body>
</html>
```

Related Items: `checkbox` mouse-related event handler

radio Input Object

Properties	Methods	Event Handlers
See `checkbox` object.		

Syntax

Accessing radio object properties or methods:

```
(All)    [window.]document.formName.buttonGroupName[index].property |
         method([parameters])
(All)    [window.]document.formName.elements[index] [index].property |
         method([parameters])
(All)    [window.]document.forms[index]. buttonGroupName[index].property |
         method([parameters])
(All)    [window.]document.forms["formName"]. buttonGroupName[index].property
          | method([parameters])
```

radioObject

```
(All)        [window.]document.forms["formName"].elements[index].property |
             method([parameters])
(IE4+)        [window.]document.all.elemID[index].property | method([parameters])
(IE5+/W3C)  [window.]document.getElementById("elemID")[index].property |
             method([parameters])
```

Compatibility: WinIE3+, MacIE3+, NN2+, Moz+, Safari+, Opera+, Chrome+

About this object

A radio button is an unusual object within the body of JavaScript applications. In every other case of form control elements, one object equals one visual element on the screen. But a radio object actually consists of a group of radio buttons. Because of the nature of radio buttons — a mutually exclusive choice among two or more selections — a group always has multiple visual elements. All buttons in the group share the same name — which is how the browser knows to group buttons together and to let the clicking of a button deselect any other selected button within the group. Beyond that, however, each button can have unique properties, such as `value` or `checked` properties.

Use JavaScript array syntax to access information about an individual button within the button group. Look at the following example of defining a button group, and see how to reference each button. This button group lets the user select an image size from a group of standard sizes, which includes as the value the number of megapixels used by that size:

```
<form>
   <b>Select your desired image size:</b><br />
   <input type="radio" name="sizes" value="0.073" checked="checked" />320x240
   <input type="radio" name="sizes" value="0.293" />640x480
   <input type="radio" name="sizes" value="0.75" />1024x768
   <input type="radio" name="sizes" value="1.25" />1280x1024
</form>
```

After this group displays on the page, the first radio button is preselected for the user. Only one property of a radio button object (`length`) applies to all members of the group. However, the other properties apply to individual buttons within the group. To access any button, use an array index value as part of the button group name. For example:

```
firstBtnValue = document.forms[0].sizes[0].value;  // "0.073"
secondBtnValue = document.forms[0].sizes[1].value; // "0.293"
```

Any time you access the `checked`, `defaultChecked`, `type`, or `value` property, you must point to a specific button within the group according to its order in the array (or each button can also have a unique ID). The order of buttons in the group depends on the sequence in which the individual buttons are defined in the HTML document. In other words, to uncover the currently selected radio button, your script has to iterate through all radio buttons in the radio group. Examples of this come later in the discussion of this object.

Supplying a `value` attribute to a radio button can be very important in your script. Although the text label for a button is defined outside the `<input>` tag, the `value` attribute lets you store

any string in the button's hip pocket. In the earlier example, the radio button labels were just first names, whereas the `value` properties were set in the definition to the full names of the actors. The values could have been anything that the script needed, such as birth dates, shoe sizes, URLs, or the first names again (because a script has no way to retrieve the labels except through `innerHTML` in IE or node property access in more modern browsers). The point is that the `value` attribute should contain whatever string the script needs to derive from the selection made by the user. The `value` attribute contents are also what is sent to a server-side program in a submit action for the form.

How you decide to orient a group of buttons on the screen is entirely up to your design and the real estate available within your document. You can string them in a horizontal row (as shown earlier), place `<br />` tags after each one to form a column, or place `<br />` tags after every other button to form a double column. Numeric order within the array is determined only by the order in which the buttons are defined in the source code, not by where they appear. To determine which radio button of a group is checked before doing processing based on that choice, you need to construct a repeat loop (shown in the next example) to cycle through the buttons in the group. For each button, your script examines the checked property.

Properties

checked

Value: Boolean Read/Write

Compatibility: WinIE3+, MacIE3+, NN2+, Moz+, Safari+, Opera+, Chrome+

Only one radio button in a group can be highlighted (checked) at a time (the browser takes care of highlighting and unhighlighting buttons in a group for you). That one button's `checked` property is set to `true`, whereas all others in the group are set to `false`. By setting the `checked` property of one button in a group to `true`, all other buttons automatically uncheck themselves.

Listing 35-5 uses a repeat loop in a function to look through all buttons in a group of image sizes, in search of the checked button. After the loop finds the one whose checked property is `true`, it returns the value of the index. In one instance, that index value is used to extract the `value` property for display in the alert dialog box; in the other instance, the value helps determine which button in the group is next in line to have its `checked` property set to `true`.

LISTING 35-5

Finding the Selected Button in a Radio Group

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Extracting Highlighted Radio Button</title>
      <script type="text/javascript">
```

continued

***radioObject*.checked**

LISTING 35-5 *(continued)*

```
         function getSelectedButton(buttonGroup)
         {
            for (var i = 0; i < buttonGroup.length; i++)
            {
               if (buttonGroup[i].checked)
               {
                  return i;
               }
            }
            return 0;
         }
         function showMegapixels(form)
         {
            var i = getSelectedButton(form.sizes);
            alert("That image size requires " + form.sizes[i].value
              + " megapixels.");
         }
         function cycle(form)
         {
            var i = getSelectedButton(form.sizes);
               if (i+1 == form.sizes.length)
               {
                  form.sizes[0].checked = true;
               }
               else
               {
                  form.sizes[i+1].checked = true;
               }
         }
      </script>
   </head>
   <body>
      <form>
         <b>Select your desired image size:</b>
         <p><input type="radio" name="sizes" value="0.073"
               checked="checked" />320x240
            <input type="radio" name="sizes" value="0.293" />640x480
            <input type="radio" name="sizes" value="0.75" />1024x768
            <input type="radio" name="sizes" value="1.25" />1280x1024
         </p>
         <p><input type="button" name="Viewer" value="View Megapixels..."
                 onclick="showMegapixels(this.form)" />
         </p>
         <p><input type="button" name="Cycler" value="Cycle Buttons"
                 onclick="cycle(this.form)" />
         </p>
      </form>
   </body>
</html>
```

Related Items: `defaultChecked` property

defaultChecked

Value: Boolean Read-Only

Compatibility: WinIE3+, MacIE3+, NN2+, Moz+, Safari+, Opera+, Chrome+

If you add the `checked` attribute to the `<input>` definition for a radio button, the `defaultChecked` property for that object is `true`; otherwise, the property is `false`. Having access to this property enables your scripts to examine individual radio buttons to see if they have been adjusted (presumably by the user, if your script does not perform automatic clicking).

In the following script fragment, a function is passed a reference to a form containing radio buttons for choosing between image sizes:

```
function groupChanged(form)
{
   for (var i = 0; i < form.sizes.length; i++)
   {
      if (form.sizes[i].defaultChecked)
      {
         if (!form.sizes[i].checked)
         {
            alert("This radio group has been changed.");
         }
      }
   }
}
```

The goal in this code is to see, in as general a way as possible (supplying the radio button group name where needed), if the user changed the default setting. Looping through each of the radio buttons, you look for the one whose `checked` attribute is set in the `<input>` definition. With that index value (`i`) in hand, you then look to see if that entry is still checked. If not (notice the `!` negation operator), you display an alert dialog box about the change.

Related Items: `checked`, `value` properties

length

Value: Integer Read-Only

Compatibility: WinIE3+, MacIE3+, NN2+, Moz+, Safari+, Opera+, Chrome+

A radio button group has a *length* — the number of individual radio buttons defined for that group. Attempting to retrieve the length of an individual button yields a `null` value. The `length` property is valuable for establishing the maximum range of values in a repeat loop that must cycle through every button within that group. If you specify the `length` property to fill that value (rather than hard-wiring the value), the loop construction will be easier to maintain — as you make changes to the number of buttons in the group during page construction, the loop adjusts to the changes automatically. Just remember that while the `length` property is the number of radio buttons in the array, the array itself (as usual) is zero-based.

Related Items: None

name

Value: Identifier string　　Read-Only

Compatibility: WinIE3+, MacIE3+, NN2+, Moz+, Safari+, Opera+, Chrome+

The `name` property, although associated with an entire radio button group, can be read only from individual buttons in the group, such as

```
btnGroupName = document.forms[0].groupName[2].name;
```

In that sense, each radio button element in a group inherits the name of the group. Your scripts have little need to extract the `name` property of a button or group. More often than not, you will hard-wire a button group's name into your script to extract other properties of individual buttons. Getting the `name` property of an object whose name you know is obviously redundant. But understanding the place of radio button group names in the scheme of JavaScript objects is important for all scripters.

Related Items: `value` property

type

Value: String (`radio`)　　Read-Only

Compatibility: WinIE4+, MacIE4+, NN3+, Moz+, Safari+, Opera+, Chrome+

Use the `type` property to help identify a radio object from an unknown group of form elements. To find out if a form element is a radio object, just look for the string `radio` as the type of the element.

Related Items: `form.elements` property

value

Value: String　　Read/Write

Compatibility: WinIE3+, MacIE3+, NN2+, Moz+, Safari+, Opera+, Chrome+

As described earlier in this chapter in the discussion of the checkbox object, the `value` property contains arbitrary information that you assign when mapping out the `<input>` definition for an individual radio button. Using this property is a handy shortcut to correlating a radio button label with detailed or related information of interest to your script or server-side application. If you like, the `value` property can contain the same text as the label.

Related Items: `name` property

Methods

click()

Returns: Nothing

Compatibility: WinIE3+, MacIE3+, NN2+, Moz+, Safari+, Opera+, Chrome+

The intention of the `click()` method is to enact, via a script, the physical act of clicking a radio button. However, you better serve your scripts by setting the `checked` properties of all buttons in a group so that you know exactly what the setting of the group is at any time.

Related Items: `checked` property; `onclick` event handler

Event handlers

onclick

Compatibility: WinIE3+, MacIE3+, NN2+, Moz+, Safari+, Opera+, Chrome+

Radio buttons, more than any user interface element available in HTML, are intended for use in making choices that other objects, such as submit or standard buttons, act upon later. You may see cases in Windows or Mac programs in which highlighting a radio button — at most — activates or brings into view additional, related settings (see Listing 35-4).

We strongly advise you not to use scripting handlers that perform significant actions at the click of any radio button. At most, you may want to use knowledge about a user's clicking of a radio button to adjust a global variable or `document.cookie` setting that influences subsequent processing. Be aware, however, that if you script such a hidden action for one radio button in a group, you must also script similar actions for others in the same group. That way, if a user changes the setting back to a previous condition, the global variable is reset to the way it was. JavaScript, however, tends to run fast enough that a batch operation can make such adjustments after the user clicks a more action-oriented button.

Every time a user clicks one of the radio buttons in Listing 35-6, he or she sets a global variable to `true` or `false`, depending on whether the person chose the smallest image size. This action is independent of the action that is taking place if the user clicks on the View Megapixels action button. An `onunload` event handler in the `<body>` definition triggers a function that displays an informational warning message just before the page clears (click the browser's Reload button to leave the current page prior to reloading). Here we use an initialize function triggered by `onload` so that the current radio button selection sets the global value upon a reload.

LISTING 35-6

An onclick Event Handler for Radio Buttons

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Radio Button onClick Handler</title>
      <script type="text/javascript">
         var LoRes = false
         function initValue()
         {
            LoRes = document.forms[0].sizes[0].checked;
```

continued

LISTING 35-6 *(continued)*

```
         }
         function showMegapixels(form)
         {
            for (var i = 0; i < form.sizes.length; i++)
            {
               if (form.sizes[i].checked)
               {
                  break;
               }
            }
            alert("That image size requires " + form.sizes[i].value + "
               megapixels.");
         }
         function setLoRes(setting)
         {
            LoRes = setting;
         }
         function exitMsg()
         {
            if (LoRes)
            {
               alert("You should probably use a higher resolution image.");
            }
         }
      </script>
   </head>
   <body onload="initValue()" onunload="exitMsg()">
      <form>
         <b>Select your desired image size:</b>
         <p><input type="radio" name="sizes" value="0.073"
                  checked="checked" onclick="setLoRes(true)" />320x240
            <input type="radio" name="sizes" value="0.293"
                  onclick="setLoRes(false)" />640x480
            <input type="radio" name="sizes" value="0.75"
                  onclick="setLoRes(false)" />1024x768
            <input type="radio" name="sizes" value="1.25"
                  onclick="setLoRes(false)" />1280x1024
         </p>
         <p><input type="button" name="Viewer" value="View Megapixels..."
                  onclick="showMegapixels(this.form)" />
         </p>
      </form>
   </body>
</html>
```

image Input Object

For HTML element properties, methods, and event handlers, see Chapter 26, "Generic HTML Element Objects."

Properties	Methods	Event Handlers
`complete`		
`form`†		
`name`†		
`src`		
`type`		

†See `Button` object.

Syntax

Accessing `image` input object properties or methods:

```
(All)       [window.]document.formName.imageName.property |
            method([parameters])
(All)       [window.]document.formName.elements[index].property |
            method([parameters])
(All)       [window.]document.forms[index].imageName.property |
            method([parameters])
(All)       [window.]document.forms["formName"].imageName.property |
            method([parameters])
(All)       [window.]document.forms["formName"].elements[index].property |
            method([parameters])
(IE4+)      [window.]document.all.elemID.property | method([parameters])
(IE5+/W3C)  [window.]document.getElementById("elemID").property |
            method([parameters])
```

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

About this object

Modern browsers support the `image` input element among scriptable objects. The `image` input object most closely resembles the button input object, but replaces the `value` property (which defines the label for the button) with the `src` property, which defines the URL for the image that is to be displayed in the form control. This is a much simpler way to define a clickable image icon, for example, than the way required for compatibility with older browsers: wrapping an `img` element inside an `a` element so that you can use the `a` element's event handlers.

Although this element loads a regular Web image in the document, you have virtually no control over the image (unlike when you use the `img` element). Be sure the rendering is as you predict.

Properties

`complete`

Value: Boolean Read-Only

Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-, Opera-, Chrome-

The `complete` property works as it does for an `img` element, reporting `true` if the image has finished loading. Otherwise the property returns `false`. Interestingly, there is no `onload` event handler for this object.

Related Items: `image.complete` property

`src`

Value: URL string Read/Write

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

Like the `img` element object, the `image` input element's `src` property controls the URL of the image being displayed in the element. The property can be used for image swapping in a form control, just as it is for a regular `img` element. Because the `image` input element has all necessary mouse event handlers available (for example, `onmouseover`, `onmouseout`, `onmousedown`) you can script rollovers, click-downs, or any other user interface technique that you feel is appropriate for your buttons and images. To adapt code written for link-wrapped images, move the event handlers from the `a` element to the `image` input element, and make sure the name of the `image` input element is the same as your old `img` element.

Older browsers load images into an `image` input element, but no event handlers are recognized.

Related Items: `image.src` property

`type`

Value: String (`image`) Read-Only

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

Use the `type` property to help you identify an image input object from an unknown group of form elements. Just look for the string `image` as the type of a form element to know if it is indeed an image input object.

Related Items: `form.elements` property

CHAPTER

36

Text-Related Form Objects

IN THIS CHAPTER

Capturing and modifying text field contents

Triggering action by entering text

Capturing individual keystroke events

The document object model for forms includes four text-related user interface objects — `text`, `password`, and `hidden` input element objects, plus the `textarea` element object. All four of these objects are used for entry, display, or temporary storage of text data. Although all these objects can have text placed in them by default as the page loads, scripts can also modify the contents of these objects. Importantly, all but the `hidden` objects retain their user- or script-modified content during a soft reload (for example, clicking the Reload button) in Mozilla and Internet Explorer; `hidden` objects revert to their default values on all reloads.

A more obvious difference between the `hidden` object and the rest is that its invisibility removes it from the realm of user events and actions. Therefore, the range of scripted possibilities is much smaller for the `hidden` object.

The persistence of `text` and `textarea` object data through reloads (and window resizes), however, is neither reliable enough nor consistent enough across all modern browsers to be used in lieu of a temporary cookie. This is a change from past implementations.

Text Input Object

For HTML element properties, methods, and event handlers, see Chapter 26, "Generic HTML Element Objects."

Properties	Methods	Event Handlers
defaultValue	select()	onafterupdate
form	blur()	onbeforeupdate
maxLength	focus()	onchange
name		onerrorupdate
readOnly		onselect
size		onblur
type		onfocus
value		

Syntax

Accessing `text` input object properties or methods:

```
(All)       [window.]document.formName.fieldName.property |
            method([parameters])
(All)       [window.]document.formName.elements[index].property |
            method([parameters])
(All)       [window.]document.forms[index].fieldName.property |
            method([parameters])
(All)       [window.]document.forms["formName"].fieldName.property |
            method([parameters])
(All)       [window.]document.forms["formName"].elements[index].property |
            method([parameters])
(IE4+)      [window.]document.all.elemID.property | method([parameters])
(IE5+/W3C)  [window.]document.getElementById("elemID").property |
            method([parameters])
```

Compatibility: WinIE3+, MacIE3+, NN2+, Moz+, Safari+, Opera+, Chrome+

About this object

The `text` input object is the primary medium for capturing single-line, user-entered text. By default, browsers tend to display entered text in a monospaced font (usually Courier or a derivative) so that you can easily specify the width (`size`) of a field based on the anticipated number of characters that a user may put into the field. Until you get to modern (IE4+ and W3C) browsers, the font is a fixed size and is always left-aligned in the field. In those later browsers, style sheets can control the font characteristics of a text field. If your design requires multiple lines of text, use the `textarea` object that comes later in this chapter.

Text object methods and event handlers use terminology that may be known to Windows users but not to Macintosh users. A field is said to have *focus* whenever the user clicks or tabs into the field. When a field has focus, either the text insertion pointer flashes, or any text in the

field may be selected. Only one text object on a page can have focus at a time. The inverse user action — clicking or tabbing away from a text object — is called a *blur*. Clicking another object, whether it is another field or a button of any kind, causes a field that currently has focus to blur.

If you don't want the contents of a field to be changed by the user, you have three options, depending on the vintage of browsers you need to support: forcing the field to lose focus; disabling the field; or setting the field's `readOnly` property.

The tactic that is completely backward compatible uses the following event handler in a field you want to protect:

```
onfocus="this.blur()"
```

Starting with IE4 and NN6/Moz1, the object model provides a `disabled` property for form controls. Setting the property to `true` leaves the element visible on the page, but the user cannot access the control. The same browsers provide a `readOnly` property, which doesn't dim the field, but prevents typing in the field.

Text fields and events

Focus and blur also interact with other possible user actions to a text object: selecting and changing. *Selecting* occurs when the user clicks and drags across any text in the field; *changing* occurs when the user makes any alteration to the content of the field and then either tabs or clicks away from that field.

When you design event handlers for fields, be aware that a user's interaction with a field may trigger more than one event with a single action. For instance, clicking a field to select text may trigger both a `focus` and `select` event. If you have conflicting actions in the `onfocus` and `onselect` event handlers, your scripts can do some weird things to the user's experience with your page. Displaying alert dialog boxes, for instance, also triggers `blur` events, so a field that has both an `onselect` handler (which displays the alert) and an `onblur` handler gets a nasty interaction from the two.

As a result, you should be very judicious with the number of event handlers you specify in any text object definition. If possible, pick one user action that you want to use to initiate some JavaScript code execution and deploy it consistently on the page. Not all fields require event handlers — only those you want to perform some action as the result of user activity in that field.

Many newcomers also become confused by the behavior of the `change` event. To prevent this event from being sent to the field for every character the user types, any change to a field is determined only *after* the field loses focus by the user's clicking or tabbing away from it. At that point, instead of a `blur` event being sent to the field, only a `change` event is sent, triggering an `onchange` event handler if one is defined for the field. This extra burden of having to click or tab away from a field may entice you to shift any `onchange` event handler tasks to a separate button that the user must click to initiate action on the field contents.

Starting with version 4 browsers, text fields also have event handlers for keyboard actions, namely `onkeydown`, `onkeypress`, and `onkeyup`. With these event handlers, you can intercept

keystrokes before the characters reach the text field. Thus, you can use keyboard events to prevent anything but numbers from being entered into a text box while the user types the characters.

To extract the current content of a text object, summon the property `document.formName.fieldName.value`. After you have the string value, you can use JavaScript's string object methods to parse or otherwise massage that text as needed for your script. If the field entry is a number and you need to pass that value to methods requiring numbers, you have to convert the text to a number with the help of the `parseInt()` or `parseFloat()` global functions.

Text Boxes and the Enter/Return Key

Early browsers established a convention that continues to this day. When a form consists of only one text box, a press of the Enter/Return key acts the same as clicking a Submit button for the form. You have probably experienced this many times when entering a value into a single search field of a form. Press the Enter/Return key, and the search request goes off to the server.

The flip side is that if the form contains more than one text box, the Enter/Return key does no submission from any of the text boxes (IE for the Mac and Safari are exceptions: they submit no matter how many text boxes there are). But with the advent of keyboard events, you can script this action (or the invocation of a client-side script) into any text boxes of the form you like. To make it work with all flavors of browsers capable of keyboard events requires a small conversion function that extracts the DOM-specific desired code from the keystroke. The following listing shows a sample page that demonstrates how to implement a function that inspects each keystroke from a text field and initiates processing if the key pressed is the Enter/Return key:

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Enter/Return Event Trigger</title>
      <script type="text/javascript">
         // Event object processor
         function isEnterKey(evt)
         {
            evt = (evt) ? evt : ((window.event) ? window.event : null);
            var keyCode;
               if (evt)
               {
                  keyCode = (evt.keyCode) ? evt.keyCode : evt.which;
               }
            return (keyCode == 13);
         }

         function processOnEnter(fld, evt)
         {
            if (isEnterKey(evt))
```

```
            {
               alert("Ready to do some work with the form.");
               return false;
            }
            return true;
         }
      </script>
   </head>
   <body>
      <h1>Enter/Return Event Trigger</h1>
      <hr />
      <form onsubmit="return false">
         Field 1: <input type="text" name="field1"
                         onkeydown="processOnEnter(this, event)" />
         Field 2: <input type="text" name="field2"
                         onkeydown="processOnEnter(this, event)" />
         Field 3: <input type="text" name="field3"
                         onkeydown="processOnEnter(this, event)" />
      </form>
   </body>
</html>
```

Properties

`defaultValue`

Value: String Read-Only

Compatibility: WinIE3+, MacIE3+, NN2+, Moz+, Safari+, Opera+, Chrome+

Though your users and your scripts are free to muck with the contents of a `text` object by assigning strings to the value property, you can always extract (and thus restore, if necessary) the string assigned to the text object in its `<input>` definition. The `defaultValue` property yields the string parameter of the `value` attribute.

Note

Listings 36-1, 36-2, and 36-3 feature a form with only one `text` input element. The rules of HTML forms say that such a form submits itself if the user presses the Enter key whenever the field has focus. Such a submission to a form whose action is undefined causes the page to reload, thus stopping any scripts that are running at the time. `form` elements for these example listings contain an `onsubmit` event handler that both blocks the submission and attempts to trigger the text box `onchange` event handler to run the demonstration script. In some browsers, such as MacIE5, you may have to press the Tab key or click outside of the text box to trigger the `onchange` event handler after you enter a new value. ■

Listing 36-1 has a simple form with a single field that has a default value set in its tag. A function (`resetField()`) restores the contents of the page's lone field to the value assigned to it in

the <input> definition. For a single-field page such as this, defining a `type="reset"` button or calling `form.reset()` works the same way because such buttons reestablish the default values of all elements of a form. But if you want to reset only a subset of fields in a form, follow the example button and function in Listing 36-1.

LISTING 36-1

Resetting a Text Object to Default Value

```
<html>
   <head>
      <title>Text Object DefaultValue</title>
      <script type="text/javascript">
      function upperMe(field) {
         field.value = field.value.toUpperCase();
      }
      function resetField(form) {
         form.converter.value = form.converter.defaultValue;
      }
      </script>
   </head>
   <body>
      <form onsubmit="window.focus(); return false">
         Enter lowercase letters for conversion to uppercase: <input
         type="text" id="convert" name="converter" value="sample"
         onchange="upperMe(this)" /> <input type="button" id="reset"
         value="Reset Field" onclick="resetField(this.form)" />
      </form>
   </body>
</html>
```

Note

The property assignment event handling technique used in this example and throughout the chapter is a deliberate simplification to make the code more readable. It is generally better to use the more modern approach of binding events using the `addEventListener()` **(NN6+/Moz/W3C) or** `attachEvent()` **(IE5+) methods. A modern cross-browser event handling technique is explained in detail in Chapter 32, "Event Objects." Several other chapters use the modern technique extensively. ■**

Related Items: `value` property

form

Value: Form object reference　　　Read-Only

Compatibility: WinIE3+, MacIE3+, NN2+, Moz+, Safari+, Opera+, Chrome+

A property of every input element object is a reference to the `form` element that contains the control. This property can be very convenient in a script when you are dealing with one form control that is passed as a parameter to the function, and you want to either access another control in the same form or invoke a method of the form. An event handler of any

input element can pass `this` as the parameter, and the function can still get access to the form without having to hard-wire the script to a particular form name or document layout.

The following function fragment receives a reference to a text element as the parameter. The text element reference is needed to decide which branch to follow; then the form is submitted.

```
function setAction(fld)
{
   if (fld.value.indexOf("@") != -1)
   {
      fld.form.action = "mailto:" + fld.value;
   }
   else
   {
      fld.form.action = "cgi-bin/normal.pl";
   }
   fld.form.submit();
}
```

Notice how this function doesn't have to worry about the form reference, because its job is to work with whatever form encloses the text field that triggers this function.

Related Items: `form` object

maxLength

Value: Integer — Read/Write

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

The `maxLength` property controls the maximum number of characters allowed to be typed into the field. There is no interaction between the `maxLength` and `size` properties. This value is normally set initially via the `maxlength` attribute of the input element.

Use The Evaluator (Chapter 4, "JavaScript Essentials") to experiment with the `maxLength` property. The top text field has no default value, but you can temporarily set it to only a few characters and see how it affects entering new values:

```
document.forms[0].input.maxLength = 3;
```

Try typing this into the field to see the results of the change. To restore the default value, reload the page.

Related Items: `size` property

name

Value: Identifier string — Read/Write

Compatibility: WinIE3+, MacIE3+, NN2+, Moz+, Safari+, Opera+, Chrome+

Text object names are important for two reasons. First, if your HTML page submits information to a CGI script or server application, the input device passes the name of the text object along

with the data to help the server program identify the data being supplied by the form. Second, you can use a text object's name in its reference within JavaScript coding. If you assign distinctive, meaningful names to your fields, these names will help you read and debug your JavaScript listings (and will help others follow your scripting tactics).

Be as descriptive about your text object names as you can. Borrowing text from the field's on-page label may help you mentally map a scripted reference to a physical field on the page. Like all JavaScript object names, text object names must begin with a letter and can be followed by any number of letters or numbers. Avoid punctuation symbols, with the exception of the very safe underscore character.

Although we urge you to use distinctive names for all objects you define in a document, you can make a case for assigning the same name to a series of interrelated fields — and JavaScript is ready to help. Within a single form, any reused name for the same object type is placed in an indexed array for that name. For example, if you define three fields with the name `entry`, the following statements retrieve the `value` property for each field:

```
data = document.forms[0].entry[0].value;
data = document.forms[0].entry[1].value;
data = document.forms[0].entry[2].value;
```

This construction may be useful if you want to cycle through all of a form's related fields to determine which ones are blank. Elsewhere, your script probably needs to know what kind of information each field is supposed to receive so that it can process the data intelligently. We don't often recommend reusing object names, but you should be aware of how the object model handles them in case you need this construction. See Chapter 34, "Form and Related Objects," for more details.

Consult Listing 36-2 later in this chapter, where we use the text object's name, `convertor`, as part of the reference when assigning a value to the field. To extract the name of a text object, you can use the property reference. Therefore, assuming that your script doesn't know the name of the first object in the first form of a document, the statement is

```
var objectName = document.forms[0].elements[0].name;
```

Related Items: `form.elements` property; all other `form` element objects' `name` properties

readOnly

Value: Boolean Read/Write

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

To display text in a text field, yet prevent users from modifying it, newer browsers offer the `readOnly` property (and tag attribute). When set to `true`, the property prevents users from changing or removing the content of the text field. Unlike a disabled text field, a read-only text field looks just like an editable one.

Use The Evaluator (Chapter 4, "JavaScript Essentials") to set the bottom text box to be read-only. Begin by typing anything you want in to the bottom text box. Then enter the following statement into the top text box:

```
document.forms[0].inspector.readOnly = true;
```

Although existing text in the box is selectable (and therefore can be copied into the clipboard), it cannot be modified or removed.

Related Items: `disabled` property

size

Value: Integer Read/Write

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

Unless otherwise directed, a text box is rendered to accommodate approximately 20 characters of text for the font family and size assigned to the element's style sheet. You can adjust this under script control (in case the `size` attribute of the tag wasn't enough) via the `size` property, whose value is measured in characters (not pixels). Be forewarned, however, that browsers don't always make completely accurate estimates of the space required to display a set number of characters. If you are setting the `maxlength` attribute of a text box, making the `size` one or two characters larger is often a safe bet.

Resize the bottom text box of The Evaluator by entering the following statements into the top text box:

```
document.forms[0].inspector.size = 20;
document.forms[0].inspector.size = 400;
```

To return the size back to normal, reload the page (or set the value to `80`).

Related Items: `maxLength` property

type

Value: String (`text`) Read-Only

Compatibility: WinIE4+, MacIE4+, NN3+, Moz+, Safari+, Opera+, Chrome+

Use the `type` property to help you identify a `text` input object from an unknown group of form elements.

Related Items: `form.elements` property

value

Value: String Read/Write

Compatibility: WinIE3+, MacIE3+, NN2+, Moz+, Safari+, Opera+, Chrome+

A text object's `value` property is the two-way gateway to the content of the field. A reference to an object's value property returns the string currently showing in the field. Note that all values coming from a text object are string values. If your field prompts a user to enter a number, your script may have to perform data conversion of the number-as-string value ("42" instead of

plain old 42) before a script can perform math operations on it. JavaScript tries to be as automatic about this data conversion as possible and follows some rules about it (see Chapter 15, "The String Object"). If you see an error message that says a value is not a number (for a math operation), the value is still a string.

Your script places text of its own into a field for display to the user by assigning a string to the `value` property of a text object. Use the simple assignment operator. For example:

```
document.forms[0].ZIP.value = "90210";
```

JavaScript is more forgiving about data types when assigning values to a text object. JavaScript does its best to convert a value to a string on its way to a text object display. Even Boolean values get converted to their string equivalents, `true` or `false`. Scripts can place numeric values into fields without a hitch. But remember that if a script later retrieves these values from the text object, they will come back as strings. About the only values that don't get converted are objects. They typically show up in text boxes as `[object]` or, in some browsers, a more descriptive label for the object.

Storing arrays in a field requires special processing. You need to use the `array.join()` method to convert an array into a string. Each array entry is delimited by a character you establish in the `array.join()` method. Later, you can use the `string.split()` method to turn this delimited string into an array.

As a demonstration of how to retrieve and assign values to a text object, Listing 36-2 shows how the action in an `onchange` event handler is triggered. Enter any lowercase letters into the field, and then click out of the field. We pass a reference to the entire form object as a parameter to the event handler. The function extracts the value, converts it to uppercase (using one of the JavaScript string object methods), and assigns it back to the same field in that form.

LISTING 36-2

Getting and Setting a Text Object's Value

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Text Object Value</title>
      <script type="text/javascript">
         function upperMe(form)
         {
            inputStr = form.converter.value;
            form.converter.value = inputStr.toUpperCase();
         }
      </script>
   </head>
   <body>
```

```
      <form onsubmit="window.focus(); return false">
         Enter lowercase letters for conversion to uppercase:
         <input type="text" name="converter" value="sample"
               onchange="upperMe(this.form)" />
      </form>
   </body>
</html>
```

We also show two other ways to accomplish the same task, each one more efficient than the previous example. Both utilize the shortcut object reference to get at the heart of the text object. Listing 36-3 passes the text object — contained in the `this` reference — to the function handler. Because that text object contains a complete reference to it (out of sight, but there just the same), you can access the `value` property of that object and assign a string to it in a simple assignment statement.

LISTING 36-3

Passing a Text Object (as this) to the Function

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Text Object Value</title>
      <script type="text/javascript">
         function upperMe(field)
         {
            field.value = field.value.toUpperCase();
         }
      </script>
   </head>
   <body>
      <form onsubmit="window.focus(); return false">
         Enter lowercase letters for conversion to uppercase:
         <input type="text" name="converter" value="sample"
               onchange="upperMe(this)" />
      </form>
   </body>
</html>
```

Yet another way is to deal with the field values directly in an embedded event handler — instead of calling an external function (which is possibly a little cleaner since there is just a single line of code in the function anyway). With the `upperMe()` function removed from the document, the event handler attribute of the `<input>` tag changes to do all the work:

```
<input type="text" name="converter" value="sample"
      onchange="this.value = this.value.toUpperCase()" />
```

The right-hand side of the assignment expression extracts the current contents of the field and (with the help of the `toUpperCase()` method of the string object) converts the original string to all uppercase letters. The result of this operation is assigned to the `value` property of the field.

The application of the `this` keyword in the previous examples may be confusing at first, but these examples represent the range of ways in which you can use such references effectively. Using `this` by itself as a parameter to an object's event handler refers only to that single object — a text object, in Listing 36-3. If you want to pass along a broader scope of objects that contain the current object, use the `this` keyword along with the outer object layer that you want. In Listing 36-2, we sent along a reference to the entire form by specifying `this.form` — meaning the form that contains "this" object, which is being defined in the line of HTML code.

At the other end of the scale, you can use similar-looking syntax to specify a particular property of the `this` object. Thus, in the last example, we zeroed in on just the `value` property of the current object being defined — `this.value`. Although the formats of `this.form` and `this.value` appear the same, the fact that one is a reference to an object and the other to just a value, can influence the way your functions work. When you pass a reference to an object, the function can read and modify properties of that object (as well as invoke its functions); but when the parameter passed to a function is just a property value, you cannot modify that value without building a complete reference to the object and its value.

Related Items: `form.defaultValue` property

Methods

`blur()`

Returns: Nothing

Compatibility: WinIE3+, MacIE3+, NN2+, Moz+, Safari+, Opera+, Chrome+

Just as a camera lens blurs when it goes out of focus, a text object blurs when it loses focus — when someone clicks or tabs out of the field. Under script control, `blur()` deselects whatever may be selected in the field, and the text insertion pointer leaves the field. The pointer does not proceed to the next field in tabbing order, as it does if you perform a blur by tabbing out of the field manually.

The following statement invokes the `blur()` method on a text box named `vanishText`:

```
document.forms[0].vanishText.blur();
```

Related Items: `focus()` method; `onblur` event handler

`focus()`

Returns: Nothing

Compatibility: WinIE3+, MacIE3+, NN2+, Moz+, Safari+, Opera+, Chrome+

For a text object, having focus means that the text insertion pointer is flashing in that text object's field (having focus means something different for buttons in a Windows environment). Giving a field focus is like opening it up for human editing.

Setting the focus of a field containing text does not let you place the cursor at any specified location in the field. The cursor usually appears at the beginning of the text (although in WinIE4+, you can use the `TextRange` object to position the cursor wherever you want in the field, as shown in Chapter 33, "Body Text Objects"). To prepare a field for the user to remove the existing text, use both the `focus()` and `select()` methods.

See Listing 36-4 for an example of an application of the `focus()` method in concert with the `select()` method.

Related Items: `select()` method; `onfocus` event handler

`select()`

Returns: Nothing

Compatibility: WinIE3+, MacIE3+, NN2+, Moz+, Safari+, Opera+, Chrome+

Selecting a field under script control means selecting all text within the text object. A typical application is one in which an entry validation script detects a mistake on the part of the user. After alerting the user to the mistake (via a `window.alert()` dialog box), the script finishes its task by selecting the text of the field in question. Not only does this action draw the user's eye to the field needing attention (especially important if the validation code is checking multiple fields), but it also keeps the old text there for the user to examine for potential problems. With the text selected, the next key the user presses erases the former entry.

Trying to select a text object's contents with the click of a button is problematic. One problem is that clicking a button brings the document's focus to the button, which disrupts the selection process. For more reliable selection, the script should invoke both the `focus()` and the `select()` methods for the field, in that order. No penalty exists for issuing both methods, and the extra insurance of the second method provides a more consistent user experience with the page.

Some versions of WinIE are known to exhibit anomalous (meaning buggy) behavior when using the technique of focusing and selecting a text field after the appearance of an alert dialog box. The fix is not elegant, but it works: inserting an artificial delay via the `setTimeout()` method before invoking a separate function that focuses and selects the field. Better-behaved browsers accept the workaround with no penalty.

***textObject*.select()**

Selecting a text object via script does *not* trigger the same `onselect` event handler for that object as the one that triggers if a user manually selects text in the field. Therefore, no event handler script is executed when a user invokes the `select()` method.

A click of the Verify button in Listing 36-4 performs a validation on the contents of the text box, making sure the entry consists only of numbers. All work is controlled by the `checkNumeric()` function, which receives a reference to the field needing inspection as a parameter. Because of the way the delayed call to the `doSelection()` function has to be configured, various parts of what will become a valid reference to the form are extracted from the field's and form's properties. If the validation (performed in the `isNumber()` function) fails, the `setSelection()` method is invoked after an artificial delay of zero milliseconds. As goofy as this sounds, this method is all that IE needs to recover from the display and closure of the alert dialog box. Because the first parameter of the `setTimeout()` method must be a string, the example assembles a string invocation of the `setSelection()` function via string versions of the form and field names. All that the `setSelection()` function does is focus and select the field whose reference is passed as a parameter. This function is now generalizable for multiple text boxes in a more complex form.

LISTING 36-4

Selecting a Field

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Text Object Select/Focus</title>
      <script type="text/javascript">
         // general purpose function to see if a suspected numeric input is a number
         function isNumber(inputStr)
         {
            for (var i = 0; i < inputStr.length; i++)
            {
               var oneChar = inputStr.charAt(i);
                  if (oneChar < "0" || oneChar > "9")
                  {
                     alert("Please make sure entries are numbers only.");
                     return false;
                  }
            }
            return true;
         }

         function checkNumeric(fld)
         {
            var inputStr = fld.value;
            var fldName = fld.name;
            var formName = fld.form.name;
               if (isNumber(inputStr))
```

```
            {
               alert("Thank you for a positive number experience.");
            }
            else
            {
               setTimeout("doSelection(document." + formName + "." +
                  fldName + ")", 0);
            }
         }

         function doSelection(fld)
         {
            fld.focus();
            fld.select();
         }
      </script>
   </head>
   <body>
      <form name="entryForm" onsubmit="return false">
         Enter any positive integer: <input type="text" name="numeric" />
         <p><input type="button" value="Verify"
                  onclick="checkNumeric(this.form.numeric)" />
         </p>
      </form>
   </body>
</html>
```

Related Items: `focus()` method; `onselect` event handler

Event handlers

onafterupdate
onbeforeupdate
onerrorupdate

Compatibility: WinIE4+, MacIE-, NN-, Moz-, Safari-, Opera-, Chrome-

If you are using WinIE data binding on a text element, the element is subject to three possible events in the course of retrieving updated data. The `onbeforeupdate` and `onafterupdate` events fire immediately before and after (respectively) the update takes place. If an error occurs in the retrieval of data from the database, the `onerrorupdate` event fires.

All three events may be used for advisory purposes. For example, an `onafterupdate` event handler may temporarily change the font characteristics of the element to signify the arrival of fresh data. Or, an `onerrorupdate` event handler may fill the field with hyphens because no valid data exists for the field. These events apply only to input elements of type `text` (meaning not `password` or `hidden` types).

Related Items: `dataFld`, `dataSrc` properties (Chapter 26, "Generic HTML Element Objects")

onblur
onfocus
onselect

Compatibility: WinIE3+, MacIE3+, NN2+, Moz+, Safari+, Opera+, Chrome+

All three of these event handlers should be used only after you have a firm understanding of the interrelationships between the events that reach text objects. You must use extreme care and conduct lots of user testing before including more than one of these three event handlers in a text object. Because some events cannot occur without triggering others, either immediately before or after (for example, an `onfocus` occurs immediately before an `onselect` if the field did not have focus before), whatever actions you script for these events should be as distinct as possible to avoid interference or overlap.

In particular, be careful about displaying modal dialog boxes (for example, `window.alert()` dialog boxes) in response to the `onfocus` event handler. Because the text field loses focus when the alert displays and then regains focus after the alert is closed, you can get yourself into a loop that is difficult to break out of. If you get trapped in this manner, try the keyboard shortcut for reloading the page (Ctrl+R or ~CM-R) repeatedly as you keep closing the dialog box window.

A question often arises about whether data-entry validation should be triggered by the `onblur` event handler or the `onchange` event handler. An `onblur` validation cannot be fooled, whereas an `onchange` validation can (the user simply doesn't change the bad entry as he or she tabs out of the field). What we don't like about the `onblur` way is that it can cause a frustrating experience for a user who wants to tab through a field now, and come back to it later (assuming your validation requires data to be entered into the field before submission). As in Chapter 46's discussion (on the CD-ROM) about form data validation, we recommend using `onchange` event handlers to trigger immediate data checking, and then using another last-minute check in a function called by the form's `onsubmit` event handler.

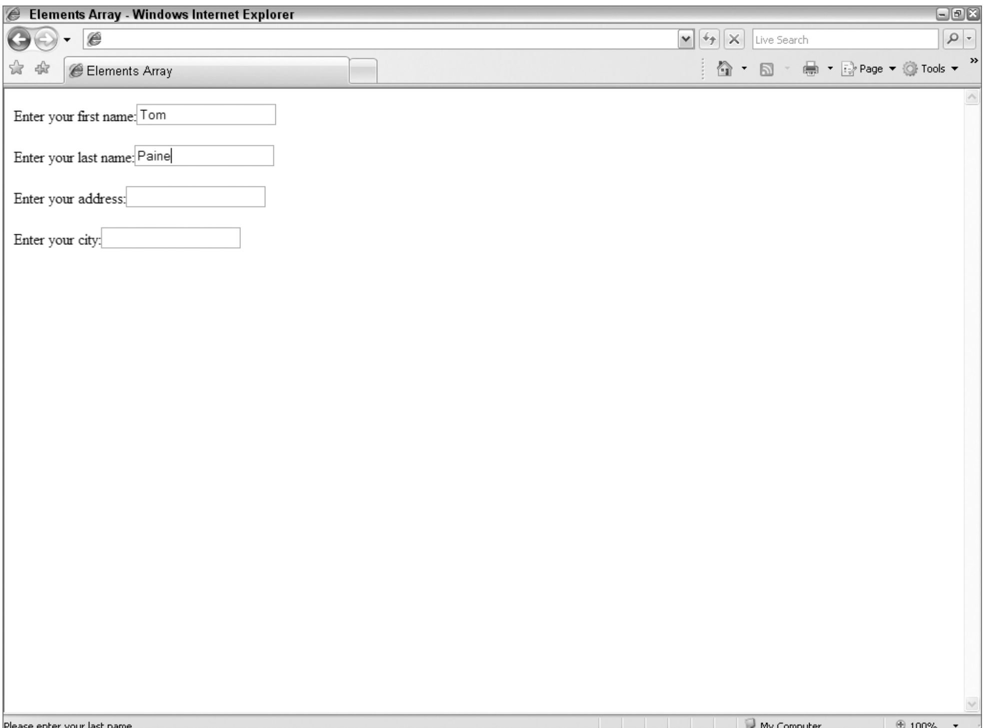
To demonstrate one of these event handlers, Listing 36-5 shows how you may use the window's status bar as a prompt message area after a user activates any field of a form. When the user tabs in to or clicks on a field, the prompt message associated with that field appears in the status bar. In Figure 36-1, the user has tabbed to the second text box, which caused the status bar message to display a prompt for the field.

Note

Some people frown upon the idea of using the browser's status bar to convey information via JavaScript, with the logic being that you shouldn't tamper with its built-in purpose of displaying status messages directly from the browser itself. There are a few browsers out there that don't have a status bar. In still another approach, Mozilla browsers by default prevent you from altering the status text. You can easily change this setting by navigating to the `about:config` URL in your Mozilla-based browser, and then changing the `dom.disable_window_status_change` preference to `false`; just double-click a preference to change its value. ■

FIGURE 36-1

An onfocus event handler triggers a status bar display.



LISTING 36-5

The onfocus Event Handler

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Elements Array</title>
      <script type="text/javascript">
         function prompt(msg)
         {
            window.status = "Please enter your " + msg + ".";
         }
      </script>
   </head>
   <body>
```

continued

LISTING 36-5 *(continued)*

```
      <form>
         Enter your first name:<input type="text" name="firstName"
                                    onfocus="prompt('first name')" />
         <p>Enter your last name:<input type="text" name="lastName"
                                      onfocus="prompt('last name')" /></p>
         <p>Enter your address:<input type="text" name="address"
                                    onfocus="prompt('address')" /></p>
         <p>Enter your city:<input type="text" name="city"
                                    onfocus="prompt('city')" /></p>
      </form>
   </body>
</html>
```

onchange

Compatibility: WinIE3+, MacIE3+, NN2+, Moz+, Safari+, Opera+, Chrome+

Of all the event handlers for a text object, the `onchange` handler is the one you will probably use most in your forms (see Listing 36-6). This event is the one we prefer for triggering the validation of whatever entry the user just typed in the field. The potential hazard of trying to do only a batch-mode data validation of all entries before submitting an entire form is that the user's mental focus is away from the entry of a given field as well. When you immediately validate an entry, the user is already thinking about the information category in question. See Chapter 46, "Data-Entry Validation" (on the CD-ROM) for more about data-entry validation, including a technique that validates user input in real time as a user types each character.

LISTING 36-6

Data Validation via an onchange Event Handler

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Text Object Select/Focus</title>
      <script type="text/javascript">
         // general purpose function to see if a suspected numeric input is a number
         function isNumber(inputStr)
         {
            for (var i = 0; i < inputStr.length; i++)
            {
               var oneChar = inputStr.substring(i, i + 1);
                  if (oneChar < "0" || oneChar > "9")
                  {
                     alert("Please make sure entries are numbers only.");
                     return false;
                  }
            }
```

```
         return true;
      }

      function checkIt(form)
      {
         inputStr = form.numeric.value;
         if (isNumber(inputStr))
         {
            alert("Thank you for a positive number experience.");
         }
         else
         {
            form.numeric.focus();
            form.numeric.select();
         }
      }
   </script>
</head>
<body>
   <form name="entryForm" onsubmit="checkIt(this); return false">
      Enter any positive integer: <input type="text" name="numeric"
                                  onchange="checkIt(this.form)" />
   </form>
</body>
</html>
```

password Input Object

Properties	Methods	Event Handlers
See the "Text Input Object" section earlier in this chapter.		

Syntax

See the "Text Input Object" section earlier in this chapter.

Compatibility: WinIE3+, MacIE3+, NN2+, Moz+, Safari+, Opera+, Chrome+

About this object

A password-style field looks like a text object, but when the user types something into the field, only asterisks or bullets (depending on your operating system) appear in the field. For the sake of security, any password exchanges should be handled by a server-side program (CGI, Java servlet, and so on).

Scripts can treat a `password` object exactly like a `text` input object. This may lead a scripter to capture a user's web site password for storage in the `document.cookie` of the client machine. A

`password` object value property is returned in plain language, so that such a captured password would be stored in the cookie file the same way. Because a client machine's cookie file can be examined on the local computer (perhaps by a snoop during lunch hour), plain-language storage of passwords is a potential security risk. Instead, develop a scripted encryption algorithm for your page for reading and writing the password in the cookie. Most password-protected sites, however, usually have a server program (CGI, for example) encrypt the password prior to sending it back to the cookie.

See the `text` object discussion for the behavior of the `password` object's properties, methods, and event handlers. The `type` property for this object returns `password`.

hidden Input Object

Properties	Methods	Event Handlers
See the "Text Input Object" section earlier in the chapter.		

Syntax

See "Text Input Object" section earlier in the chapter.

Compatibility: WinIE4+, MacIE4+, NN3+, Moz+, Safari+, Opera+, Chrome+

About this object

A `hidden` object is a simple string holder within a form object whose contents are not visible to the user of your web page. Despite the long list of properties, methods, and event handlers that this input element type inherits by virtue of being an input element, you will be doing little with a `hidden` element beyond reading and writing its `value` property.

The `hidden` object plays a vital role in applications that rely on CGI programs on the server. Very often, the server has data that it needs to convey to itself the next time the client makes a submission (for example, a user ID captured at the application's login page). A CGI program can generate an HTML page with the necessary data hidden from the user, but located in a field transmitted to the server at submit time.

Along the same lines, a page for a server application may present a user-friendly interface that makes data-entry easy for the user. But on the server end, the database, or other application, requires that the data be in a more esoteric format. A script located in the page generated for the user can use the `onsubmit` event handler to perform last-minute assembly of user-friendly data into database-friendly data in a hidden field. When the CGI program receives the request from the client, it passes along the hidden field value to the database.

We are not fans of the `hidden` object for use on client-side-only JavaScript applications. If we want to deliver some default data collections or values with our JavaScript-enabled pages, we do so in JavaScript variables and arrays as part of the script.

Because scripted changes to the contents of a hidden field are fragile (for example, a soft reload erases the changes), the only place you should consider making such changes is in the same script that submits a form to a CGI program, or in a function triggered by an `onsubmit` event handler. In effect, you're just using the hidden fields as holding pens for the scripted data to be submitted. For more persistent storage, use the `document.cookie` property or genuine text fields in hidden frames, even if just for the duration of the visit to the page.

For information about the properties of the `hidden` object, consult the earlier listing for the `text` input object. The `type` property for this object returns `hidden`.

textarea Element Object

For HTML element properties, methods, and event handlers, see Chapter 26, "Generic HTML Element Objects."

Properties	Methods	Event Handlers
`cols`	`createTextRange()`	`onafterupdate`†
`form`†	`select()`†	`onbeforeupdate`†
`name`†		`onchange`†
`readOnly`†		`onerrorupdate`†
`rows`		
`type`†		
`wrap`		

†See the ''Text Input Object'' section earlier in the chapter

Syntax

Accessing `textarea` element object properties or methods:

```
(All)       [window.]document.formName.textareaName.property |
            method([parameters])
(All)       [window.]document.formName.elements[index].property |
            method([parameters])
(All)       [window.]document.forms[index].textareaName.property |
            method([parameters])
(All)       [window.]document.forms["formName"].textareaName.property |
            method([parameters])
(All)       [window.]document.forms["formName"].elements[index].property |
            method([parameters])
(IE4+)      [window.]document.all.elemID.property | method([parameters])
(IE5+/W3C)  [window.]document.getElementById("elemID").property |
            method([parameters])
```

Compatibility: WinIE3+, MacIE3+, NN2+, Moz+, Safari+, Opera+, Chrome+

About this object

Although not in the same HTML syntax family as other `<input>` elements of a form, a `textarea` object is indeed a form input element, providing multiple-line text input facilities. Although some browsers let you put a `textarea` element anywhere in a document, it really should be contained by a `form` element.

A `textarea` object closely resembles a text object, except for attributes that define its physical appearance on the page. Because the intended use of a `textarea` object is for multiple-line text input, the attributes include specifications for height (number of rows) and width (number of columns in the monospaced font). No matter what size you specify, the browser displays a textarea with horizontal and vertical scrollbars in older browsers; more recent browsers tend to be smarter about displaying scrollbars only when needed (although there are exceptions). Text entered in the textarea wraps by default in all modern browsers, although the wrapping can be adjusted via the `wrap` attribute in IE. The `wrap` attribute can be set to `soft` (the default), `hard`, or `off`. The `soft` and `hard` settings result in word wrapping, whereas the `off` setting causes text to scroll for a significant distance horizontally (the horizontal scrollbar appears automatically). This field is, indeed, a primitive text field by GUI computing standards, in that font specifications made possible in newer browsers by way of style sheets apply to all text in the box.

Use The Evaluator Sr. (Chapter 4, "JavaScript Essentials") to play with the `cols` and `rows` property settings for the Results textarea on that page. Shrink the width of the textarea by entering the following statement into the top text box:

```
document.forms[0].output.cols = 30;
```

And make the textarea one row deeper:

```
document.forms[0].output.rows++;
```

All properties, methods, and event handlers of text objects apply to the `textarea` object. They all behave exactly the same way (except, of course, for the `type` property, which is `textarea`). Therefore, refer to the previous listings for the text object for scripting details for those items. Some additional properties that are unique to the `textarea` object are discussed next.

Carriage returns inside textareas

The three classes of operating systems supported by modern browsers — Windows, Macintosh, and UNIX — do not agree about what constitutes a carriage return character in a text string. This discrepancy carries over to the `textarea` object and its contents on these platforms.

After a user enters text and uses Enter/Return on the keyboard, one or more unseen characters are inserted into the string. In the parlance of JavaScript's literal string characters, the carriage

return consists of some combination of the newline (`\n`) and return (`\r`) character. The following table shows the characters inserted into the string for each operating system category.

Operating System	Character String
Windows	`\r\n`
Macintosh	`\r`
UNIX	`\n`

This tidbit is valuable if you need to remove carriage returns from a textarea for processing in a CGI or local script. The problem is that you obviously need to perform platform-specific operations on each. For the situation in which you must preserve the carriage return locations, but your server-side database cannot accept the carriage return values, we suggest you use the `string.escape()` method to URL-encode the string. The return character is converted to `%0D`, and the newline character is converted to `%0A`. Of course these characters occupy extra character spaces in your database, so these additions must be accounted for in your database design.

As far as writing carriage returns into textareas, the situation is a bit easier. From NN3 and IE4 onward, if you specify any one of the combinations in the preceding table, all platforms know how to automatically convert the data to the form native to the operating system. Therefore, you can set the value of a `textarea` object to `1\r\n2\r\n3` in all platforms, and a columnar list of the numbers 1, 2, and 3 will appear in those fields. Or, if you URL-encoded the text for saving to a database, you can unescape that character string before setting the textarea value, and no matter what platform the visitor has, the carriage returns are rendered correctly. Upon reading those values again by script, you can see that the carriage returns are in the form of the platform (shown in the previous table).

Properties

`cols`
`rows`

Value: Integer Read/Write

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

The displayed size of a `textarea` element is defined by its `cols` and `rows` attributes, which are represented in the object model by the `cols` and `rows` properties, respectively. Values for these properties are integers. For `cols`, the number represents the number of characters that can be displayed without horizontal scrolling of the textarea; for `rows`, the number is the number of lines of text that can be displayed without vertical scrolling.

Related Items: `wrap` property

wrap

Value: String Read/Write

Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-, Opera-, Chrome-

The `wrap` property represents the `wrap` attribute, which, surprisingly, is not a W3C-sanctioned attribute as of HTML 4.01. In any case, IE4+ lets you adjust the property by scripting. Allowable string values are `soft`, `hard`, and `off`. The browser adds soft returns (the default in IE) to word-wrap the content, but no carriage return characters are actually inserted into the text. A setting for hard returns means that carriage return characters are added to the text (and would be submitted with the value to a server application). With wrap set to `off`, text continues to extend beyond the right edge of the textarea until the user manually presses the Enter/Return key.

Related Items: `cols` property

Methods

createTextRange()

Returns: `TextRange` object

Compatibility: WinIE4+, MacIE-, NN-, Moz-, Safari-, Opera-, Chrome-

The `createTextRange()` method for a `textarea` operates just as the `document.createTextRange()` method, except that the range consists of text inside the `textarea` element, apart from the regular body content. This version of the `TextRange` object comes in handy when you want a script to control the location of the text insertion pointer inside a `textarea` element for the user.

See the example for the `TextRange.move()` method in Chapter 33, "Body Text Objects," to see how to control the text insertion pointer inside a `textarea` element.

Related Items: `TextRange` object (Chapter 33)

CHAPTER

37

Select, Option, and FileUpload Objects

IN THIS CHAPTER

Triggering action based on a user's selection in a pop-up or select list

Modifying the contents of select objects

Using the fileUpload object

Selection lists — whether in the form of pop-up menus or scrolling lists — are space-saving form elements in HTML pages. They enable designers to present a lot of information in a comparatively small space. At the same time, users are familiar with the interface elements from working in their own operating systems' preference dialog boxes and application windows.

However, selection lists are more difficult to script, especially in older browsers, because the objects themselves are complicated entities. Scripts find all the real data associated with the form control in `option` elements that are nested inside `select` elements. As you can see throughout this chapter, backward-compatible references necessary to extract information from a `select` element object and its `option` objects can get pretty long. On the upside, the most painful backward compatibility efforts are all but unnecessary given the proliferation of modern browsers.

The other object covered in this chapter, the `fileUpload` input object, is frequently misunderstood as being more powerful than it actually is. It is, alas, not the great file transfer elixir desired by many page authors.

select Element Object

For HTML element properties, methods, and event handlers, see Chapter 26, "Generic HTML Element Objects."

selectObject

Properties	Methods	Event Handlers
`form`†	`add()`	`onchange`
`length`	`options[i].add()`	
`multiple`	`item()`	
`name`†	`namedItem()`	
`options[]`	`remove()`	
`selectedIndex`	`options[i].remove()`	
`size`		
`type`		
`value`		

†See ''Text Input Object'' (Chapter 36, ''Text-Related Form Objects'').

Syntax

Accessing `select` element object properties:

```
(All)       [window.]document.formName.selectName.property |
            method([parameters])
(All)       [window.]document.formName.elements[index].property |
            method([parameters])
(All)       [window.]document.forms[index].selectName.property |
            method([parameters])
(All)       [window.]document.forms["formName"].selectName.property |
            method([parameters])
(All)       [window.]document.forms["formName"].elements[index].property |
            method([parameters])
(IE4+)      [window.]document.all.elemID.property | method([parameters])
(IE5+/W3C)  [window.]document.getElementById("elemID").property |
            method([parameters])
```

Compatibility: WinIE3+, MacIE3+, NN2+, Moz+, Safari+, Opera+, Chrome+

About this object

`select` element objects are perhaps the most visually interesting user interface elements among the standard built-in objects. In one format, they appear on the page as pop-up lists; in another format, they appear as scrolling list boxes. Pop-up lists, in particular, offer efficient use of page real estate for presenting a list of choices for the user. Moreover, only the choice selected by the user shows on the page, minimizing the clutter of unneeded verbiage.

Compared with other JavaScript objects, `select` objects are difficult to script — mostly because of the complexity of data that goes into a list of items. What the user sees as a `select` element

on the page consists of both that element and option elements that contain the actual choices from which the user makes a selection. Some properties that are of value to scripters belong to the select object, whereas others belong to the nested option objects. For example, you can extract the number (index) of the currently selected option in the list — a property of the entire select object. To get the displayed text of the selected option, however, you must zero in further to extract the text property of a single option among all options defined for the object.

When you define a select object within a form, the construction of the <select>...</select> tag pair is easy to inadvertently mess up. First, most attributes that define the entire object — such as name, size, and event handlers — are attributes of the opening <select> tag. Between the end of the opening tag and the closing </select> tag are additional tags for each option to be displayed in the list. The following object definition creates a selection pop-up list containing three color choices:

```
<form>
   <select name="RGBColors" onchange="changeColor(this)">
      <option selected="selected">Red</option>
      <option>Green</option>
      <option>Blue</option>
   </select>
</form>
```

The indented formatting of the tags in the HTML document is not critical. We indent the lines of options merely for the sake of readability, which is still a worthy cause even if it isn't technically necessary.

By default, a select element is rendered as a pop-up list. To make it appear as a scrolled list, assign an integer value greater than 1 to the size attribute to specify how many options should be visible in the list without scrolling — how tall the list's box should be, measured in lines. Because scrollbars in GUI environments tend to require a fair amount of space to display a minimum set of clickable areas (including sliding "thumbs"), you should set list-box style sizes to no less than 4. If that makes the list box too tall for your page design, consider using a pop-up menu instead.

Significant differences exist in the way each GUI platform presents pop-up menus. Because each browser sometimes relies on the operating system to display its native pop-up menu style (and sometimes the browser designers go their own way), considerable differences exist among the OS and browser platforms in the size of a given pop-up menu. What fits nicely within a standard window width of one OS may not fit in the window of another OS in a different browser. In other words, you cannot rely on any select object having a precise dimension on a page (in case you're trying to align a select object with an image).

In list-box form, you can set a select object to accept multiple, noncontiguous selections. Users typically accomplish such selections by holding down a modifier key (the Shift, Ctrl, or ⌘ key, depending on the operating system) while clicking additional options. To switch on this capability for a select object, include the multiple attribute constant in the definition.

For each entry in a list, your <select> tag definition must include an <option> tag plus the text as you want it to appear in the list. If you want a pop-up list to show a default selection

when the page loads, you must attach a `selected` attribute to that item's `<option>` tag. Without this attribute, the default item may be empty or the first item, depending on the browser. (We go more in depth about this in the `option` object discussion later in this chapter.) You can also assign a string to each `option`'s `value` attribute. As with radio buttons, this value can be text other than the wording displayed in the list. In essence, your script can act on that "hidden" value rather than on the displayed text, such as letting a plain-language select listing actually refer to a complex URL. This string value is also the value sent to a CGI program (as part of the name-value pair) when the user submits the `select` object's form.

One behavioral aspect of the `select` object may influence your page design. The `onchange` event handler triggers immediately when a user makes a new selection in a pop-up list. If you prefer to delay any action until the user makes other settings in the form, omit an `onchange` event handler in the `select` object — but be sure to create a button that enables users to initiate an action governed by those user settings.

Modifying select options (NN3+, IE4+)

Script control gives you considerable flexibility in modifying the contents and selection of a `select` object. There are several techniques at your disposal when it comes to working with the `select` object, including some whose support dates back to NN3 and IE4. We quickly show you how to approach the `select` object from this backward-compatible perspective, and then move on to revealing a W3C standard approach a bit later in the chapter. Some aspects of manipulating the `select` object are rather straightforward, such as changing the `selectObj.options[i].text` property to alter the display of a single-option entry. The situation gets tricky, though, when the number of options in the `select` object changes. Your choices include:

- Removing an individual option (and thus collapsing the list)
- Reducing an existing list to a fewer number of options
- Removing all options
- Adding new options to a `select` object

To remove an option from the list, set the specific option to `null`. For example, if a list contains five items and you want to eliminate the third item altogether (reducing the list to four items), the syntax (from the `select` object reference) for doing that task is this:

```
selectObj.options[2] = null;
```

After this statement, `selectObj.options.length` equals `4`.

In another scenario, suppose that a `select` object has five options in it and you want to replace it with one having only three options. You first must hard-code the `length` property to `3`:

```
selectObj.options.length = 3;
```

Then, set individual `text` and `value` properties for index values `0` through `2`.

Perhaps you want to start building a new list of contents by completely deleting the original list (without harming the `select` object). To accomplish this, set the `length` to `0`:

```
selectObj.options.length = 0;
```

From here, you have to create new options (as you do when you want to expand a list from, say, three to seven options). The mechanism for creating a new option involves an object constructor: `new Option()`. This constructor accepts up to four parameters, which enable you to specify the equivalent of an `<option>` tag's attributes:

- Text to be displayed in the option
- Contents of the option's `value` property
- Whether the item is the `defaultSelected` option (Boolean)
- Whether the item is selected (Boolean)

You can set any (or none) of these items as part of the constructor and return to other statements to set their properties. We suggest setting the first two parameters (leave the others blank) and then setting the `selected` property separately. The following is an example of a statement that creates a new, fifth entry, in a `select` object and sets both its displayed `text` and `value` properties:

```
selectObj.options[4] = new Option("Yahoo","http://www.yahoo.com");
```

To demonstrate all of these techniques, Listing 37-1 enables you to change the text of a `select` object — first by adjusting the text properties in the same number of options and then by creating an entirely new set of options. Radio button `onclick` event handlers trigger functions for making these changes — rare examples of when radio buttons can logically initiate visible action.

LISTING 37-1

Modifying select Options

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Changing Options On The Fly</title>
      <script type="text/javascript" language="JavaScript">
         // flag to reload page for older NNs
         var isPreNN6 = (navigator.appName == "Netscape"
                         && parseInt(navigator.appVersion) <= 4);

         // initialize color list arrays
         plainList = new Array(6);
         hardList = new Array(6);
         plainList[0] = "cyan";
         hardList[0] = "#00FFFF";
```

continued

LISTING 37-1 *(continued)*

```
plainList[1] = "magenta";
hardList[1] = "#FF00FF";
plainList[2] = "yellow";
hardList[2] = "#FFFF00";
plainList[3] = "lightgoldenrodyellow";
hardList[3] = "#FAFAD2";
plainList[4] = "salmon";
hardList[4] = "#FA8072";
plainList[5] = "dodgerblue";
hardList[5] = "#1E90FF";

// change color language set
function setLang(which)
{
   var listObj = document.forms[0].colors;
   // filter out old browsers
      if (listObj.type)
      {
         // find out if it's 3 or 6 entries
         var listLength = listObj.length;
         // save selected index
         var currSelected = listObj.selectedIndex;
         // replace individual existing entries
         for (var i = 0; i < listLength; i++)
         {
            if (which == "plain")
            {
               listObj.options[i].text = plainList[i];
            }
            else
            {
               listObj.options[i].text = hardList[i];
            }
         }
         if (isPreNN6)
         {
            history.go(0);
         }
         else
         {
            listObj.selectedIndex = currSelected;
         }
      }
}

// create entirely new options list
function setCount(choice)
{
   var listObj = document.forms[0].colors;
   // filter out old browsers
      if (listObj.type)
```

```
            {
               // get language setting
               var lang = (document.forms[0].geekLevel[0].checked) ?
                           "plain" : "hard";
               // empty options from list
               listObj.length = 0;
               // create new option object for each entry
               for (var i = 0; i < choice.value; i++)
               {
                  if (lang == "plain")
                  {
                     listObj.options[i] = new Option(plainList[i]);
                  }
                  else
                  {
                     listObj.options[i] = new Option(hardList[i]);
                  }
               }
               listObj.options[0].selected = true;
               if (isPreNN6)
               {
                  history.go(0);
               }
            }
         }
      </script>
   </head>
   <body>
      <h1>Flying Select Options</h1>
      <form>
         Choose a palette size:
         <input type="radio" name="paletteSize"
                value="3" onclick="setCount(this)"
                checked="checked" />Three
         <input type="radio" name="paletteSize" value="6"
                onclick="setCount(this)" />Six
         <p>Choose geek level:
            <input type="radio" name="geekLevel" value=""
                   onclick="setLang('plain')" checked="checked" />Plain-language
            <input type="radio" name="geekLevel" value=""
                   onclick="setLang('hard')" />Gimme hex-triplets!
         </p>
         <p>Select a color:
            <select name="colors">
               <option selected="selected">cyan</option>
               <option>magenta</option>
               <option>yellow</option>
            </select>
         </p>
      </form>
   </body>
</html>
```

Note

The property assignment event handling technique used in this example and throughout the chapter is a deliberate simplification to make the code more readable. It is generally better to use the more modern approach of binding events using the `addEventListener()` (NN6+/Moz/W3C) or `attachEvent()` (IE5+) methods. A modern cross-browser event handling technique is explained in detail in Chapter 32, "Event Objects." Several other chapters use the modern technique extensively. ■

In an effort to make this code easily maintainable, the color choice lists (one in plain language, the other in hexadecimal triplet color specifications) are established as two separate arrays. Repeat loops in both large functions can work with these arrays no matter how big they get.

The first two radio buttons (see Figure 37-1) trigger the `setLang()` function. This function's first task is to extract a reference to the `select` object to make additional references shorter (just `listObj`). Then by way of the `length` property, you find out how many items are currently displayed in the list because you just want to replace as many items as are already there. In the repeat loop, you set the `text` property of the existing `select` options to corresponding entries in either of the two array listings.

FIGURE 37-1

Radio button choices alter the contents of the select object on-the-fly.

In the second pair of radio buttons, each button stores a value indicating how many items should be displayed when the user clicks the button. This number is picked up by the `setCount()` function and is used in the repeat loop as a maximum counting point. In the meantime, the function finds the selected language radio button and zeros out the `select` object entirely. Options are rebuilt from scratch using the `new Option()` constructor for each option. The parameters are the corresponding display text entries from the arrays. Because none of these new options have other properties set (such as which one should be selected by default), the function sets that property of the first item in the list.

Notice that both functions call `history.go(0)` for legacy browsers after setting up their `select` objects. The purpose of this call is to give these earlier Navigator versions an opportunity to resize the `select` object to accommodate the contents of the list. The difference in size here is especially noticeable when you switch from the six-color, plain-language list to any other list. Without resizing, some long items are not readable. IE4+ and NN6+/Moz, on the other hand, automatically redraw the page to the newly sized form element.

Modifying select options (IE4+)

Microsoft offers another way to modify `select` element options for IE4+, but the technique involves two proprietary methods of the `options` array property of the `select` object. Because we cover all other ways of modifying the `select` element in this section, we cover the IE way of doing things here as well. Interestingly, some of the other modern browsers (Safari, Opera, and Chrome) now support this approach.

The two `options` array methods are `add()` and `remove()`. The `add()` method takes one required parameter and one optional parameter. The required parameter is a reference to an `option` element object that your script creates in another statement (using the `document.createElement()` method). If you omit the second parameter to `add()`, the new `option` element is appended to the current collection of items. But you can also specify an index value as the second parameter. The index points to the position in the `options` array where the new item is to be inserted.

Listing 37-2 shows how to modify the two main functions from Listing 37-1 using the IE approach exclusively (changes and additions appear in bold). The script assumes that only IE browsers ever load the page (in other words, there is no filtering for browser brand here, which in the "real world" is mandatory). When replacing one set of options with another, there are two approaches demonstrated. In the first (the `setLang()` function), the replacements have the same number of items, so the length of existing options provides a counter and index value for the `remove()` and `add()` methods. But when the number of items may change (as in the `setCount()` function), a tight loop removes all items before they are added back via the `add()` method without a second parameter (items are appended to the list). The approach shown in Listing 37-2 has no specific benefit over that of Listing 37-1.

LISTING 37-2

Modifying select Options (IE4+)

```
// change color language set
function setLang(which)
{
   var listObj = document.forms[0].colors;
   var newOpt;
   // filter out old IE browsers
      if (listObj.type)
      {
         // find out if it's 3 or 6 entries
         var listLength = listObj.length;
         // save selected index
         var currSelected = listObj.selectedIndex;
         // replace individual existing entries
         for (var i = 0; i < listLength; i++)
         {
            newOpt = document.createElement("option");
            newOpt.text = (which == "plain") ? plainList[i] : hardList[i];
            listObj.options.remove(i);
            listObj.options.add(newOpt, i);
         }
         listObj.selectedIndex = currSelected;
      }
}

// create entirely new options list
function setCount(choice)
{
   var listObj = document.forms[0].colors;
   var newOpt;
   // filter out old browsers
      if (listObj.type)
      {
         // get language setting
         var lang = (document.forms[0].geekLevel[0].checked) ? "plain" : "hard";
         // empty options from list
         while (listObj.options.length)
         {
            listObj.options.remove(0);
         }
         // create new option object for each entry
         for (var i = 0; i < choice.value; i++)
         {
            newOpt = document.createElement("option");
            newOpt.text = (lang == "plain") ? plainList[i] : hardList[i];
            listObj.options.add(newOpt);
         }
         listObj.options[0].selected = true;
      }
}
```

Modifying select options (W3C DOM)

Yet another approach is possible in browsers that closely adhere to the W3C DOM Level 2 standard. In NN6+, Mozilla, and Safari, for example, you can use the `add()` and `remove()` methods of the `select` element object. They work very much like the same-named methods for the `options` array in IE4+, but these are methods of the `select` element object itself. The other main difference between the two syntaxes is that the `add()` method does not use the index value as the second parameter but rather a reference to the `option` element object before which the new option is inserted. The second parameter is required, so to simply append the new item at the end of the current list, supply `null` as the parameter. Listing 37-3 shows the W3C-compatible version of the `select` element modification scripts shown in Listings 37-1 and 37-2. We highlight source code lines in bold that exhibit differences between the IE4+ and W3C DOM versions.

LISTING 37-3

Modifying select Options (W3C)

```
// change color language set
function setLang(which)
{
   var listObj = document.forms[0].colors;
   var newOpt;
   // filter out old IE browsers
      if (listObj.type)
      {
         // find out if it's 3 or 6 entries
         var listLength = listObj.length;
         // save selected index
         var currSelected = listObj.selectedIndex;
         // replace individual existing entries
         for (var i = 0; i < listLength; i++)
         {
            newOpt = document.createElement("option");
            newOpt.text = (which == "plain") ? plainList[i] : hardList[i];
            listObj.remove(i);
            listObj.add(newOpt, listObj.options[i]);
         }
         listObj.selectedIndex = currSelected;
      }
}

// create entirely new options list
function setCount(choice)
{
   var listObj = document.forms[0].colors;
   var newOpt;
   // filter out old browsers
      if (listObj.type)
      {
```

continued

LISTING 37-3 *(continued)*

```
      // get language setting
      var lang = (document.forms[0].geekLevel[0].checked) ? "plain" : "hard";
      // empty options from list
      while (listObj.options.length)
      {
         listObj.remove(0);
      }
      // create new option object for each entry
      for (var i = 0; i < choice.value; i++)
      {
         newOpt = document.createElement("option");
         newOpt.text = (lang == "plain") ? plainList[i] : hardList[i];
         listObj.add(newOpt, null);
      }
      listObj.options[0].selected = true;
   }
}
```

As with the IE4 version, the W3C version offers no specific benefit over the original, backward-compatible approach. Choose the most modern one that fits the types of browsers you need to support with your page.

Properties

`length`

Value: Integer Read/Write (see text)

Compatibility: WinIE3+, MacIE3+, NN2+, Moz+, Safari+, Opera+, Chrome+

Like all JavaScript arrays, the `options` array has a `length` property of its own. But rather than having to reference the `options` array to determine its length, the `select` object has its own `length` property that you use to find out how many items are in the list. This value is the number of options in the object. A `select` object with three choices in it has a `length` property value of `3`.

In browsers dating back to NN3 and IE4 you can adjust this value downward after the document loads. This is one way to decrease the number of options in a list. Setting the value to `0` causes the `select` object to empty but not disappear.

See Listing 37-1 for an illustration of the way you use the `length` property to help determine how often to cycle through the repeat loop in search of selected items. Because the loop counter, `i`, must start at `0`, the counting continues until the loop counter is one less than the actual length value (which starts its count with `1`).

Related Item: `options` property

multiple

Value: Boolean Read/Write

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

The `multiple` property represents the `multiple` attribute setting for a `select` element object. If the value is `true`, the element accepts multiple selections by the user (for example, Ctrl+clicking in Windows). If you want to convert a pop-up list into a multiple `select` pick list, you must also adjust the `size` property to direct the browser to render a set number of visible choices in the list.

The following statement toggles between single and multiple selections on a `select` element object whose `size` attribute is set to a value greater than `1`:

```
document.forms[0].mySelect.multiple = !document.forms[0].mySelect.multiple;
```

Related Item: `size` property

options[*index*]

Value: Array of `option` element objects Read-Only

Compatibility: WinIE3+, MacIE3+, NN2+, Moz+, Safari+, Opera+, Chrome+

You typically don't summon this property by itself. Rather, it is part of a reference to a specific option's properties (or methods in later browsers) within the entire `select` object. In other words, the `options` property is a kind of gateway to more specific properties, such as the value assigned to a single option within the list.

In modern browsers (IE4+ and W3C), you can reference individual options as separate HTML element objects. These references do not require the reference to the containing `form` or `select` element objects. If backward compatibility is a priority, however, we recommend you stick with the long references through the `select` objects.

We list the next several properties here in the `select` object discussion because they are backward-compatible with all browsers, including browsers that don't treat the `option` element as a distinct object. Be aware that all properties shown here that include `options[index]` as part of their references are also properties of the `option` element object in IE4+ and W3C browsers.

See Listings 37-1 through 37-3 for examples of how the `options` array references information about the options inside a `select` element.

Related Items: All `options[index].property` items

options[*index*].defaultSelected

Value: Boolean Read-Only

Compatibility: WinIE3+, MacIE3+, NN2+, Moz+, Safari+, Opera+, Chrome+

If your `select` object definition includes one option that features the `selected` attribute, that option's `defaultSelected` property is set to `true`. The `defaultSelected` property for all other options is `false`. If you define a `select` object that allows multiple selections (and whose `size` attribute is greater than 1), however, you can define the `selected` attribute for more than one option definition. When the page loads, all items with that attribute are preselected for the user (even in noncontiguous groups).

The following statement preserves a Boolean value if the first option of the select list is the default selected item:

```
var zeroIsDefault = document.forms[0].listName.options[0].defaultSelected;
```

Related Item: `options[index].selected` property

options[*index*].index

Value: Integer Read-Only

Compatibility: WinIE3+, MacIE3+, NN6+, Moz+, Safari+, Opera+, Chrome+

The `index` value of any single option in a `select` object is likely a redundant value in your scripting. Because you cannot access the option without knowing the index anyway (in brackets as part of the `options[index]` array reference), you have little need to extract the `index` value. The value is a property of the item just the same.

The following statement assigns the index integer of the first option of a `select` element named `listName` to a variable named `itemIndex`.

```
var itemIndex = document.forms[0].listName.options[0].index;
```

Related Item: `options` property

options[*index*].selected

Value: Boolean Read/Write

Compatibility: WinIE3+, MacIE3+, NN2+, Moz+, Safari+, Opera+, Chrome+

As mentioned earlier in the discussion of this object, better ways exist for determining which option a user selects from a list than looping through all options and examining the `selected` property. An exception to that "rule" occurs when you set up a list box to enable multiple selections. In this situation, the `selectedIndex` property returns an integer of only the topmost item selected. Therefore, your script needs to look at the `true` or `false` values of the `selected` property for each option in the list and determine what to do with the text or value data.

To accumulate a list of all items selected by the user, the `seeList()` function in Listing 37-4 systematically examines the `options[index].selected` property of each item in the list. The text of each item whose `selected` property is `true` is appended to the list. We add the "`\n`" inline carriage returns and spaces to make the list in the alert dialog box look nice and indented. If you assign other values to the `value` attributes of each option, the script can extract the `options[index].value` property to collect those values instead.

LISTING 37-4

Cycling through a Multiple-Selection List

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Accessories List</title>
      <script type="text/javascript">
         function seeList(form)
         {
            var result = "";
            for (var i = 0; i < form.accList.length; i++)
            {
               if (form.accList.options[i].selected)
               {
                  result += "\n  " + form.accList.options[i].text;
               }
            }
            alert("You have selected:" + result);
         }
      </script>
   </head>
   <body>
      <form>
         <p>Control/Command-click on all accessories you use:
            <select name="accList" size="9" multiple="multiple">
               <option selected="selected">Color Monitor</option>
               <option>Modem</option>
               <option>Scanner</option>
               <option>Laser Printer</option>
               <option>Tape Backup</option>
               <option>MO Drive</option>
               <option>Video Camera</option>
            </select>
         </p>
         <p><input type="button" value="View Summary..."
                  onclick="seeList(this.form)" />
         </p>
      </form>
   </body>
</html>
```

Related Items: `options[index].text`, `options[index].value`, `selectedIndex` properties

`options[`*`index`*`].text`

Value: String Read/Write

Compatibility: WinIE3+, MacIE3+, NN2+, Moz+, Safari+, Opera+, Chrome+

selectObject.options[*index*].text

The `text` property of an option is the text of the item as it appears in the list. If you can pass that wording along with your script to perform appropriate tasks, this property is the one you want to extract for further processing. But if your processing requires other strings associated with each option, assign a `value` attribute in the definition and extract the `options[index].value` property (see Listing 37-6).

To demonstrate the `text` property of an option, Listing 37-5 applies the text from a selected option to the `document.bgColor` property of a document in the current window. The color names are part of the collection built into all scriptable browsers; fortunately, the values are case-insensitive so that you can capitalize the color names displayed and assign them to the property.

LISTING 37-5

Using the options[index].text Property

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Color Changer 1</title>
      <script type="text/javascript">
         function seeColor(form)
         {
            var newColor =
                (form.colorsList.options[form.colorsList.selectedIndex].text);
            document.bgColor = newColor;
         }
      </script>
   </head>
   <body>
      <form>
         <p>Choose a background color:
            <select name="colorsList">
               <option selected="selected">Gray</option>
               <option>Lime</option>
               <option>Ivory</option>
               <option>Red</option>
            </select>
         </p>
         <p><input type="button" value="Change It"
                   onclick="seeColor(this.form)" />
         </p>
      </form>
   </body>
</html>
```

Related Item: `options[index].value` property

options[*index*].value

Value: String — Read/Write

Compatibility: WinIE4+, MacIE4+, NN4+, Moz+, Safari+, Opera+, Chrome+

In many instances, the words in the options list appear in a form that is convenient for the document's users but inconvenient for the scripts behind the page. Rather than set up an elaborate lookup routine to match the `selectedIndex` or `options[index].text` values with the values your script needs, you can easily store those values in the `value` attribute of each `<option>` definition of the `select` object. You can then extract those values as needed.

You can store any string expression in the `value` attributes. That includes URLs, object properties, or even entire page descriptions that you want to send to a `parent.frames[index].document.write()` method.

Starting with IE4 and W3C browsers, the `select` element object itself has a `value` property that returns the `value` property of the selected option. But for backward compatibility, use the longer approach shown in the example in Listing 37-6.

Listing 37-6 requires the option text that the user sees to be in familiar, multiple-word form. But to set the color using the browser's built-in color palette, you must use the one-word form. Those one-word values are stored in the `value` attributes of each `<option>` definition. The function then reads the `value` property, assigning it to the `bgColor` of the current document. If you prefer to use the hexadecimal triplet form of color specifications, those values are assigned to the `value` attributes (`<option value="#e9967a">Dark Salmon</option>`).

LISTING 37-6

Using the options[index].value Property

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Color Changer 2</title>
      <script type="text/javascript">
         function seeColor(form)
         {
            var newColor =
                (form.colorsList.options[form.colorsList.selectedIndex].value);
            document.bgColor = newColor;
         }
      </script>
   </head>
```

continued

LISTING 37-6 *(continued)*

```
   <body>
      <form>
         <p>Choose a background color:
            <select name="colorsList">
               <option selected="selected" value="cornflowerblue">
                       Cornflower Blue</option>
               <option value="darksalmon">Dark Salmon</option>
               <option value="lightgoldenrodyellow">
                       Light Goldenrod Yellow</option>
               <option value="seagreen">Sea Green</option>
            </select>
         </p>
         <p><input type="button" value="Change It"
                   onclick="seeColor(this.form)" />
         </p>
      </form>
   </body>
</html>
```

Related Item: `options[index].text` property

selectedIndex

Value: Integer Read/Write

Compatibility: WinIE3+, MacIE3+, NN2+, Moz+, Safari+, Opera+, Chrome+

When a user clicks a choice in a selection list, the `selectedIndex` property changes to a zero-based number corresponding to that item in the list. The first item has a value of `0`. This information is valuable to a script that needs to extract the value or text of a selected item for further processing.

You can use this information as a shortcut to getting at a selected option's properties. To examine a `select` object's `selected` property, rather than cycling through every option in a repeat loop, use the object's `selectedIndex` property to fill in the index value for the reference to the selected item. The wording gets kind of long; but from an execution standpoint, this methodology is much more efficient. Note, however, that when the `select` object is multiple-style, the `selectedIndex` property value reflects the index of only the topmost item selected in the list.

To script the selection of a particular item, assign an integer value to the `select` element object's `selectedIndex` property, as shown in Listings 37-1 through 37-3.

In the `inspect()` function of Listing 37-7, notice that the value inside the `options` property index brackets is a reference to the object's `selectedIndex` property. Because this property always returns an integer value, it fulfills the needs of the index value for the `options` property. Therefore, if you select Green in the pop-up menu, `form.colorsList.selectedIndex` returns a value of `1`; that reduces the rest of the reference to `form.colorsList.options[1].text`, which equals "Green."

LISTING 37-7

Using the selectedIndex Property

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Select Inspector</title>
      <script type="text/javascript">
         function inspect(form)
         {
            alert(form.colorsList.options[form.colorsList.selectedIndex].text);
         }
      </script>
   </head>
   <body>
      <form>
         <p>
            <select name="colorsList">
               <option selected="selected">Red</option>
               <option value="Plants">Green</option>
               <option>Blue</option>
            </select>
         </p>
         <p><input type="button" value="Show Selection"
                  onclick="inspect(this.form)" />
         </p>
      </form>
   </body>
</html>
```

Related Item: `options` property

size

Value: Integer Read/Write

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

The `size` property represents the `size` attribute setting for a `select` element object. You can modify the integer value of this property to change the number of options that are visible in a pick list without having to scroll.

The following statement uses the `size` property to set the number of visible items to 5:

```
document.forms[0].mySelect.size = 5;
```

Related Item: `multiple` property

type

Value: String Read-Only

Compatibility: WinIE4+, MacIE4+, NN3+, Moz+, Safari+, Opera+, Chrome+

Use the `type` property to help you identify a `select` object from an unknown group of form elements. The precise string returned for this property depends on whether the `select` object is defined as a single- (`select-one`) or multiple- (`select-multiple`) type.

Related Item: `form.elements` property

value

Value: String Read/Write (see text)

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

The more recent browsers (and the W3C DOM) provide a `value` property for the `select` element object. This property returns the string assigned to the `value` attribute (or `value` property) of the currently selected `option` element. If you do not assign a string to the attribute or property, the `value` property returns an empty string. For these browser generations, you can use this shortcut reference to the `select` element object's `value` property instead of the longer version that requires a reference to the `selectedIndex` property and the `options` array of the element object.

The `seeColor()` function in Listing 37-6 that accesses the chosen value the long way can be simplified for newer browsers only with the following construction:

```
function seeColor(form)
{
   document.bgColor = form.colorsList.value;
}
```

Related Item: `options[index].value` property

Methods

add(*newOptionElementRef[, index]*)
add(*newOptionElementRef, optionElementRef*)
remove(*index*)

Returns: Nothing

Compatibility: WinIE5+, MacIE5+, NN6+, Moz+, Safari+, Opera+, Chrome+

These methods represent the W3C approach to adding and removing option elements from a selection. The first parameter to each of the `add()` methods is the new `option` element object to be added to the selection. The second parameters differ due to variances in IE and other W3C browsers. The first version of `add()` is the IE version, which allows you to specify an optional index position for the new option; the option is placed just before the index position,

or it is appended to the end of the selection list if no index is provided. The W3C approach is represented by the second `add()` method, which requires an `option` object reference as the second parameter. This reference is to an option already in the selection list; the new option is added just before the option, or it is appended to the end of the selection list if `null` is passed as the second parameter.

The `remove()` method requires the `index` of the option to be removed, and simply removes the option from the selection list.

options.add(*elementRef*[, *index*])
options.remove()

Returns: Nothing

Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari+, Opera+, Chrome+

These two IE-specific methods are now supported by some of the other browsers. They belong to the `options` array property of a `select` element object. See the opening of the discussion of the `select` element object earlier in this chapter to see how to use these methods and their counterparts in other browser versions and object models.

item(*index*)
namedItem("*optionID*")

Returns: `option` element reference

Compatibility: WinIE5+, MacIE5+, NN-, Moz5+, Safari-, Opera9+, Chrome-

The `item()` and `namedItem()` methods are IE-specific convenience methods that access `option` element objects nested inside a `select` object. In a sense, they provide shortcuts to referencing nested options without having to use the `options` array property and the indexing within that array.

The parameter for the `item()` method is an index integer value. For example, the following two statements refer to the same `option` element object:

```
document.forms[0].mySelect.options[2]
document.forms[0].mySelect.item(2)
```

If your script knows the ID of an `option` element, it can use the `namedItem()` method, supplying the string version of the ID as the parameter, to return a reference to that option element.

The following statement assigns an option element reference to a variable:

```
var oneOption = document.forms[0].mySelect.namedItem("option3_2");
```

Related Item: `options` property

Event handlers

onchange

Compatibility: WinIE3+, MacIE3+, NN2+, Moz+, Safari+, Opera+, Chrome+

As a user clicks a new choice in a `select` object, the object receives a `change` event that the `onchange` event handler can capture. In examples earlier in this section (Listings 37-6 and 37-7, for example), the action is handed over to a separate button. This design may make sense in some circumstances, especially when you use multiple `select` lists or any list box. (Typically, clicking a list box item does not trigger any action that the user sees.) There are also accessibility concerns for users who do not have JavaScript enabled. Restricting action to scripted events (without a corresponding Go or similar explicit button adjacent to the `select` element) may mean that choosing an item in the list would have no effect. Therefore, consider your users carefully before implementing actions exclusively via the `change` event.

To bring a pop-up menu to life, bind an `onchange` event handler to the `<select>` definition. If the user makes the same choice as previously selected, the `onchange` event handler is not triggered. In this case, you can still trigger an action via the `onclick` event handler; but this event works for the `select` object only in modern browsers.

Listing 37-8 is a version of Listing 37-6 that invokes all action as the result of a user making a selection from the pop-up menu. The `onchange` event handler for the `<select>` tag replaces the action button. For this application — when you desire a direct response to user input — it's appropriate to have the action triggered from the pop-up menu rather than by a separate action button.

Notice two other important changes. First, the `select` element now contains a blank first option. When a user visits the page, nothing is selected yet, so you should present a blank option to encourage the user to make a selection. (The function also makes sure that the user selects one of the color-valued items before it attempts to change the background color.)

Second, the `onload` event handler invokes the `seeColor()` method, passing as a parameter a reference to the `select` element. This forces any color selection to be carried out when the page is initially loaded. As an example, this might take place if you navigate to another page and then use the browser's Back button to return to the color page. Thus, if the `select` element choice persists, the background color is adjusted accordingly after the page loads.

LISTING 37-8

Triggering a Color Change from a Pop-Up Menu

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Color Changer 2</title>
```

```
      <script type="text/javascript">
         function seeColor(list)
         {
            var newColor = (list.options[list.selectedIndex].value);
               if (newColor)
               {
                  document.bgColor = newColor;
               }
         }
      </script>
   </head>
   <body onload="seeColor(document.getElementById('colorsList'))">
      <form>
         <p>Choose a background color:
            <select name="colorsList" id="colorsList" onchange="seeColor(this)">
               <option selected="selected" value=""></option>
               <option value="cornflowerblue">Cornflower Blue</option>
               <option value="darksalmon">Dark Salmon</option>
               <option value="lightgoldenrodyellow">Light Goldenrod Yellow</option>
               <option value="seagreen">Sea Green</option>
            </select>
         </p>
      </form>
   </body>
</html>
```

option Element Object

For HTML element properties, methods, and event handlers, see Chapter 26, "Generic HTML Element Objects."

Properties	Methods	Event Handlers
`defaultSelected`††		
`form`†		
`label`		
`selected`††		
`text`††		
`value`††		

†See "Text Input Object" (Chapter 36, "Text-Related Form Objects").

††See "Select Element Object."

Syntax

Accessing `option` object properties:

```
(All)        [window.]document.formName.selectName.options[index].property |
             method([parameters])
(All)        [window.]document.formName.elements[index].options
             [index].property  | method([parameters])
(All)        [window.]document.forms[index].selectName.options[index].property
             | method([parameters])
(All)        [window.]document.forms["formName"].selectName.options[index].
             property | method([parameters])
(All)        [window.]document.forms["formName"].elements[index].options
             [index].property | method([parameters])
(IE4+)       [window.]document.all.elemID.property | method([parameters])
(IE5+/W3C)   [window.]document.getElementById("elemID").property |
             method([parameters])
(W3C)        [window.]document.forms[index].selectName.item(index).property |
             method([parameters])
(W3C)        [window.]document.forms["formName"].selectName.namedItem(elemID).
             property | method([parameters])
```

Compatibility: WinIE3+, MacIE3+, NN2+, Moz+, Safari+, Opera+, Safari+

About this object

`option` elements are nested inside `select` elements. Each option represents an item in the list of choices presented by the `select` element. Properties of the `option` element object let scripts inspect whether a particular option is currently selected or is the default selection. Other properties enable you to get or set the hidden value associated with the option as well as the visible text. For more details about the interaction between the `select` and `option` element objects, see the discussion about the `select` object earlier in this chapter as well as the discussion of the properties and methods associated with the `options` array returned by the `select` object's `options` property.

We discuss all backward-compatible `option` object properties (`defaultSelected`, `selected`, `text`, and `value`) among the `options` property descriptions in the `select` object section. The only items listed in this section are those that are unique to the `option` element object defined in newer browsers.

In all browsers dating back to NN3 and IE4, there is a provision for creating a new `option` object via an `Option` object constructor function. The syntax is as follows:

```
var newOption = new Option("text","value");
```

Here, *text* is the string that is displayed for the item in the list, and *value* is the string assigned to the `value` property of the new option. This new `option` object is not added to a `select` object until you assign it to a slot in the `options` array of the `select` object. You can see an example of this approach to modifying options in Listing 37-1.

Properties

label

Value: String Read/Write

Compatibility: WinIE6+, MacIE5+, NN6+, Moz+, Safari+, Opera+, Chrome+

The `label` property corresponds to the HTML 4.01 `label` attribute of an `option` element. This attribute (and property) enables you to assign alternate text for an option. In MacIE5+, any string assigned to the `label` attribute or corresponding property overrides the display of text found between the start and end tags of the `option` element. Therefore, you can assign content to both the attribute and tag, but only browsers adhering to the HTML 4.01 standard for this element display the value assigned to the label. Although the `label` property is implemented in NN6, the browser does not modify the option item's text to reflect the property's setting. This problem is resolved in Moz+ browsers.

The following statement modifies the text that appears as the selected text in a pop-up list:

```
document.forms[0].mySelect.options[3].label = "Widget 9000";
```

If this option is the currently selected one, the text on the pop-up list at rest changes to the new label.

Related Item: `text` property

optgroup Element Object

For HTML element properties, methods, and event handlers, see Chapter 26, "Generic HTML Element Objects."

Properties	Methods	Event Handlers
form†		
label		

†See ''Text Input Object'' (Chapter 36, ''Text-Related Form Objects'').

Syntax

Accessing `optgroup` object properties:

```
(IE)        [window.]document.all.elemID".property | method([parameters])
(W3C)       [window.]document.getElementById("elemID").property |
            method([parameters])
```

Compatibility: WinIE6+, MacIE5+, NN6+, Moz+, Safari+, Opera+, Chrome+

About this object

An optgroup element in the HTML 4.01 specification enables authors to group options into subgroups within a select list. The label assigned to the optgroup element is rendered in the list as a non-selectable item, usually differentiated from the selectable items by some alternate display. In W3C browsers, optgroup items by default are shown in bold italic, whereas all option elements nested within an optgroup are indented but with normal font characteristics.

Browsers not recognizing this element ignore it. All options are presented as if the optgroup elements are not there.

Properties

label

Value: String Read/Write

Compatibility: WinIE6+, MacIE5+, NN6+, Moz+, Safari+, Opera+, Chrome+

The label property corresponds to the HTML 4.01 label attribute of an optgroup element. This attribute (and property) enables you to assign text to the label that encompasses a group of nested option elements in the pop-up list display.

Note

MacIE5+ exhibits a bug that prevents scripts from assigning values to the last optgroup element inside a select element. ■

Listing 37-9 demonstrates how a script can alter the text of option group labels. This page is an enhanced version of the background color setters used in other examples of this chapter. Be aware that through Opera 9 and several versions of IE prior to IE7, the browser does not alter the last optgroup element's label, and NN6+ achieves only a partial change to the text displayed in the select element. Newer Mozilla-based browsers such as Firefox 1.5+ have no problems with the task.

LISTING 37-9

Modifying optgroup Element Labels

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Color Changer 3</title>
      <script type="text/javascript">
         var regularLabels = ["Reds","Greens","Blues"];
         var naturalLabels = ["Apples","Leaves","Sea"];
         function setRegularLabels(list)
```

```
      {
         var optGrps = list.getElementsByTagName("optgroup");
         for (var i = 0; i < optGrps.length; i++)
         {
            optGrps[i].label = regularLabels[i];
         }
      }
      function setNaturalLabels(list)
      {
         var optGrps = list.getElementsByTagName("optgroup");
         for (var i = 0; i < optGrps.length; i++)
         {
            optGrps[i].label = naturalLabels[i];
         }
      }
      function seeColor(list)
      {
         var newColor = (list.options[list.selectedIndex].value);
            if (newColor)
            {
               document.bgColor = newColor;
            }
      }
   </script>
</head>
<body onload="seeColor(document.getElementById('colorsList'))">
   <form>
      <p>Choose a background color:
         <select name="colorsList" id="colorsList" onchange="seeColor(this)">
            <optgroup id="optGrp1" label="Reds">
               <option value="#ff9999">Light Red</option>
               <option value="#ff3366">Medium Red</option>
               <option value="#ff0000">Bright Red</option>
               <option value="#660000">Dark Red</option>
            </optgroup>
            <optgroup id="optGrp2" label="Greens">
               <option value="#ccff66">Light Green</option>
               <option value="#99ff33">Medium Green</option>
               <option value="#00ff00">Bright Green</option>
               <option value="#006600">Dark Green</option>
            </optgroup>
            <optgroup id="optGrp3" label="Blues">
               <option value="#ccffff">Light Blue</option>
               <option value="#66ccff">Medium Blue</option>
               <option value="#0000ff">Bright Blue</option>
               <option value="#000066">Dark Blue</option>
            </optgroup>
         </select>
      </p>
      <p><input type="radio" name="labels" checked="checked"
               onclick="setRegularLabels(this.form.colorsList)" />Regular
                  Label Names
```

continued

LISTING 37-9 *(continued)*

```
            <input type="radio" name="labels"
                   onclick="setNaturalLabels(this.form.colorsList)" />Label Names
                      from Nature
         </p>
      </form>
   </body>
</html>
```

Related Item: `option.label` property

file Input Element Object

For HTML element properties, methods, and event handlers, see Chapter 26, "Generic HTML Element Objects."

Properties	Methods	Event Handlers
`defaultValue`†	`select()`†	`onchange`†
`form`†		
`name`†		
`readOnly`†		
`size`†		
`type`†		
`value`†		

†See "Text Input Object" (Chapter 36, "Text-Related Form Objects").

Syntax

Accessing `file` input element object properties:

```
(NN3+/IE4+)    [window.]document.formName.inputName.property |
               method([parameters])
(NN3+/IE4+)    [window.]document.formName.elements[index].property |
               method([parameters])
(NN3+/IE4+)    [window.]document.forms[index].inputName.property |
               method([parameters])
(NN3+/IE4+)    [window.]document.forms["formName"].inputName.property |
               method([parameters])
```

```
(NN3+/IE4+)    [window.]document.forms["formName"].elements[index].property |
               method([parameters])
(IE4+)         [window.]document.all.elemID.property | method([parameters])
(IE5+/W3C)     [window.]document.getElementById("elemID").property |
               method([parameters])
```

Compatibility: WinIE4+, MacIE4+, NN3+, Moz+, Safari+, Opera+, Chrome+

About this object

Some web sites enable you to upload files from the client to the server, typically by using a form-style submission to a CGI program on the server. The `input` element whose type is set to `"file"` (also known as a `fileUpload` object) is merely a user interface that enables users to specify which file on their PC they want to upload. Without a server process capable of receiving the file, the `file` input element does nothing. Moreover, you must also set two `form` element attributes as follows:

```
method="post"
enctype="multipart/form-data"
```

This element displays a field and a Browse button. The Browse button leads to an Open File dialog box (in the local operating system's interface vernacular) where a user can select a file. After you make a selection, the filename (or pathname, depending on the operating system) appears in the `file` input element's field. The `value` property of the object returns the filename.

You do not have to script much for this object on the client side. The `value` property, for example, is read-only in earlier browsers; in addition, a form cannot surreptitiously upload a file to the server without the user's knowledge or consent.

Listing 37-10 helps you see what the `file` input element looks like in an example page.

LISTING 37-10

file Input Element

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>FileUpload Object</title>
   </head>
   <body>
      <form method="post" action="yourCGIURL" enctype="multipart/form-data">
         File to be uploaded: <input type="file" size="40" name="fileToGo" />
         <p><input type="button" value="View Value"
                  onclick="alert(this.form.fileToGo.value)" />
```

continued

LISTING 37-10 *(continued)*

```
         </p>
      </form>
   </body>
</html>
```

In a true production environment, a Submit button and a URL to your CGI process are specified for the `action` attribute of the `<form>` tag. You would also likely use the more modern (but more code intensive) event binding approach to handle the `onclick` event, as you've seen in other examples in this chapter — here it was more concise to stick with the simple attribute assignment technique.

CHAPTER

38

Style Sheet and Style Objects

IN THIS CHAPTER

Managing style sheets by script

Changing element styles on-the-fly

Distinguishing among style, styleSheet, and style objects

Style sheets promote a concept that makes excellent sense in the fast-paced, high-volume content creation environment that is today's World Wide Web: separating content from its rendering. Textual content may come from any number of electronic sources, but it may need to be dropped into different contexts — just like an online news feed that becomes amalgamated into dozens of web portal sites, each with its own look and feel. All the content author cares about is the text and its meaning; the web page designer then decides how that content should be rendered on the page.

The style sheet concept has other advantages. Consider the large corporate web site that wants to promote its identity through a distinct style. A family of style sheets can dictate the font face, font size, the look of emphasized text, and the margin width of all body text. To apply these styles on an element-by-element basis is not only a tedious page-authoring task; it is fraught with peril. If the style is omitted from the tags of one page, the uniformity of the look is destroyed. Worse yet, if the corporate design changes to use a different font face, the task of changing every style in every tag — even with a highly powered search-and-replace operation — is risky. But if a single external style sheet file dictates the styles, then the designer need make the change only in that one file to cause the new look to ripple through the entire web site.

Learning how to create and apply style sheets is beyond the scope of this book, and this chapter assumes you are already familiar with style sheet terminology, such as a style sheet rule and a selector. If these terms are not in your vocabulary, you can find numerous tutorials on the subject both online and in books. Although IE, Firefox, Camino, Safari, and other recent browsers adhere fairly closely to W3C standards for style sheets (called Cascading Style Sheets, or CSS for short), your first learning experience should come from sources that focus on standards, rather than browser-specific features. Microsoft includes some extras in the style sheet

vocabulary that work only on IE4+ for Windows; Firefox and other Mozilla-based browsers have specially named, preliminary properties that offer future CSS capabilities in advance of the final standards. Unless you develop for a single target browser brand and client operating system, learning the common denominator of style sheet features is the right way to go. Details in this chapter cover all versions, so pay close attention to compatibility listings for each item.

Making Sense of the Object Names

The first task in this chapter is to clarify the seemingly overlapping terminology for the style sheet-related objects that you will be scripting. Some objects are more abstract than others, but they are all important. The objects in question are

- `style` element object
- `styleSheet` object (a member of the `styleSheets` array)
- `rule` or `cssRule` object (a member of the `rules` or `cssRules` array)
- `style` object

A `style` element object is the object that represents the `<style>` tag in your document. Most of its properties are inherited from the basic HTML element objects you see detailed in Chapter 26, “Generic HTML Element Objects.” By and large, you won’t be reading or writing style sheet properties via the `style` element object.

A style sheet can be embedded in a document via the `<style>` tag, or it may be linked in via a `<link>` tag. One property of the `document` object, the `styleSheets` property, returns an array (collection) of all `styleSheet` objects that are currently “visible” to the document, whether or not they are disabled. Included in the collection are style sheets defined by a `<style>` tag or linked in via a `<link>` tag. Even though the `<style>` tag, for example, contains lines of code that make up the rules for a style sheet, the `style` element object is not the path to reach the individual rules. The `styleSheet` object is. It is through the `styleSheet` object that you can enable or disable an entire sheet, access individual rules (via the `rules` or `cssRules` property array), and add or delete rules for that style sheet.

The meat of any style sheet is the set of rules that define how elements are to be rendered. At this object level, the terminology forks in IE4 and NN6/Moz. The IE4+ object model calls each style sheet rule a rule object; the W3C DOM Level 2 model (in NN6+/Moz), calls each rule a `cssRule` object. MacIE5 and Safari support both references to the same object. Despite the incompatible object names, the two objects share key property names. Assembling a reference to a rule requires array references. For example, the reference to the first rule of the first `styleSheet` object in the document is as follows for various browsers:

```
var oneRule = document.styleSheets[0].rules[0];      // IE4+, MacIE5, Safari
var oneRule = document.styleSheets[0].cssRules[0];   // NN6+/Moz, MacIE5, Safari
```

The last object in this quartet of style-related objects is the `style` object. This object is the mother lode, where actual style definitions take place. In earlier chapters, you have seen countless examples of modifying one or more `style` properties of an element. Most typically, this

modification is accomplished through the `style` property of the HTML element. For example, you would set the font color of a `span` element whose ID is "hot" as follows:

```
document.getElementById("hot").style.color = "red";
```

The `style` object is also a property of a `rule`/`cssRule` object. Thus, if you need to modify the style of elements affected by an existing style sheet rule, you approach the `style` object through a different reference path, but the `style` object is treated just as it is for elements:

```
document.styleSheets[0].rules[0].style.color = "red";    // IE4+, MacIE5, Safari
document.styleSheets[0].cssRules[0].style.color = "red"; // NN6+/Moz, MacIE5
                                                          // Safari
```

Many scripters concern themselves solely with the `style` object, and at that, a `style` object associated with a particular element object. Rare are instances that require manipulation of `styleSheet` objects beyond enabling and disabling them under script control. Therefore, if you are learning about these objects for the first time, pay closest attention to the `style` object details rather than to the other related objects.

Imported Style Sheets

Style sheets embedded in a document via the `style` element can import additional style sheets via the `@import` selector:

```
<style type="text/css">
   @import url(externalStyle.css);
   p {font-size:16pt}
</style>
```

In this example scenario, the document sees just one `styleSheet` object. But that object has a style sheet nested inside — the style sheet defined by the external file. IE4+ calls one of these imported style sheets an `import` object. An `import` object has all the properties of any `styleSheet` object, but its `parentStyle` property is a reference to the `styleSheet` that "owns" the `@import` rule. In fact, the `@import` statement does not even appear among the rules collection of the IE `styleSheet` object. Therefore, to access the first rule of the imported style sheet, the reference is as follows:

```
document.styleSheets[0].imports[0].rules[0]
```

The W3C DOM and NN6+/Moz treat `import` rule objects differently from the IE model. To the W3C DOM, even an at-rule is considered one of the `cssRules` collections of a `styleSheet` object. One of the properties of a `cssRule` object is `type`, which conveys an integer code value revealing whether the rule is a plain CSS rule or one of several other types, including an `import` rule. Of course, an imported rule object then has as one of its properties the `styleSheet` object that, in turn, contains the rules defined in the external style sheet file. The parent-child relationship exists here, as well, whereby the style sheet that contains the `@import` rule is referenced by the imported `styleSheet` object's `parentStyle` property (just as in IE4+).

Reading Style Properties

Both the IE4+ and W3C object models exhibit a behavior that at first glance may seem disconcerting. On the one hand, the W3C, and good HTML practice, encourage defining styles remotely (that is, embedded via `<style>` or `<link>` tags) rather than as values assigned to the `style` attribute of individual element tags throughout the document. This more closely adheres to the notion of separating style from content.

On the other hand, object models can be very literal beasts. Strictly speaking, if an element object presents a scriptable property that reflects an attribute for that element's tag, the first time a script tries to read that property, a value will be associated with that property *only* if the attribute is explicitly assigned in the HTML code. But if you assign style sheet settings via remote style sheets, the values are not explicitly set in the tag. Therefore, the `style` property of such an element comes up empty, even though the element is under the stylistic control of the remote style sheet. If all you want to do is assign a new value to a style property, that's not a problem, because your assignment to the element object's `style` property overrides whatever style is assigned to that property in the remote style sheet (and then that new value is subsequently readable from the `style` property). But if you want to see what the current setting is, the initial value won't be in the element's `style` object.

Microsoft (in IE5+) and the W3C DOM provide competing (and incompatible) solutions to this problem.

IE5+ provides an extra read-only property — `currentStyle` — that reveals the style sheet values that are currently being applied to the element, regardless of where the style sheet definitions are. The `currentStyle` property returns an object that is in the same format and has most of the same properties as the regular `style` property. If your audience runs browsers no earlier than IE5, you should make a habit of reading styles for an element via its `currentStyle` property. If you want a change to a `style` object's property to apply to only one element, use the element's `style` property to set that value; but if the change is to apply to all elements covered by the same remote style sheet rule, modify the `style` property of the rule object.

The W3C DOM solution is the `getComputedStyle()` method. Although the W3C DOM doesn't (yet) talk about a `window` object, it does describe an object (called the `defaultView`) which Mozilla-based browsers channel through the `window` object. The truly W3C standard way to access this method (supported by Mozilla-based and WebKit-based browsers) is via the `document.defaultView` object. To read the value of a particular style property being applied to an element, you first retrieve a *computed style* value for the element, and then read the desired CSS style property. For example, to read the `font-family` currently applied to an element whose ID is `myP`, use the following sequence:

```
var elem = document.getElementById("myP");
var computedStyle = document.defaultView.getComputedStyle(elem, "");
var fontFam = computedStyle.getPropertyValue("font-family");
```

Note that you must use the CSS property name (for example, `font-family`) and not the scripted equivalent of that property (for example, `fontFamily`).

style Element Object

See Chapter 26, "Generic HTML Element Objects," for items shared by all HTML elements.

Properties	Methods	Event Handlers
media		
type		

Syntax

Accessing `style` element object properties and methods:

```
(IE4+)        document.all.objectID.property | method([parameters])
(IE5+/W3C)    document.getElementById(objectID).property | method([parameters])
```

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

About this object

The `style` element is part of the classification of HTML directive elements (see Chapter 40, "HTML Directive Objects") in that it goes in the `head` portion of a document and does not have any of its own content rendered in the page. But the style element's contents obviously have a great amount of control over the rendering of other elements. Most of the properties, methods, and event handlers that the `style` element inherits from all HTML elements are irrelevant.

One exception is the Boolean `disabled` property. Although there are additional ways to disable a style sheet (the `disabled` property of the `styleSheet` object, for example), it may be easier to disable or enable a style sheet by way of the `style` element object. Because you can assign an ID to this element and reference it explicitly, doing so may be more convenient than trying to identify which `styleSheet` object among the document's `styleSheets` collection you intend to enable or disable.

Properties

media

Value: String — Read/Write

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

The `media` property represents the `media` attribute of a `style` element. This attribute can define what kind of output device is governed by the style sheet. The HTML 4.0 specification has lofty goals for this attribute, but most browsers are limited to the following values: `screen`, `print`, and `all`. Thus, you can design one set of styles to apply when the page is viewed on the computer screen and a different set for when it's printed.

Other types that may eventually enter the picture with the `media` property include `braille`, `embossed`, `handheld`, `projection`, `speech`, `tty`, and `tv`. Web development tools have already begun offering support for these property values. For example, Dreamweaver 8 has a special toolbar that allows you to access all of the previously listed `media` types except for `aural`, `braille`, and `embossed`. Opera is currently the only browser to support `media` property settings beyond `screen`, `print`, and `all`. In Opera, you can also specify `projection`, `handheld`, `speech`, and `tv`.

type

Value: String Read/Write

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

The `type` property represents the `type` attribute of the `style` element. For Cascading Style Sheets, this property is always set to `text/css`. If your scripts assign some other value to this property and the browser does not support that style sheet type, the style sheet no longer functions as a Cascading Style Sheet, and any styles it controls revert to their default styles.

styleSheet Object

Properties	Methods	Event Handlers
`cssRules`	`addImport()`	
`cssText`	`addRule()`	
`disabled`	`deleteRule()`	
`href`	`insertRule()`	
`id`	`removeRule()`	
`imports`		
`media`		
`ownerNode`		
`ownerRule`		
`owningElement`		
`pages`		
`parentStyleSheet`		
`readOnly`		
`rules`		
`title`		
`type`		

Syntax

Accessing `styleSheet` object properties and methods:

```
(IE4+/W3C)    document.styleSheets[index].property | method([parameters])
```

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

About this object

If the `style` element object is the concrete incarnation of a style sheet, then the `styleSheet` object is its abstract equivalent. A `styleSheet` object exists by virtue of a style sheet definition being embedded in the current document, either by way of the `<style>` tag or linked in from an external file via the `<link>` tag. Each element that introduces a style sheet into a document creates a separate `styleSheet` object. Access to a `styleSheet` object is via the `document.styleSheets` array. If the document contains no style sheet definitions, then the array has a length of zero. Styles that are introduced into a document by way of an element's `style` attribute are not considered `styleSheet` objects.

Although both IE4+ and W3C DOM browsers present `styleSheet` objects — and the object represents the same "thing" in both browser families — the set of properties and methods diverges widely between browsers. In many cases, the object provides the same information but through differently named properties in the two families. Interestingly, on some important properties, such as the ones that return the array of style rules and a reference to the HTML element that is responsible for the style sheet's being in the document, MacIE5 and Safari provide both the Microsoft and W3C terminology. Methods for this object focus on adding rules to and deleting rules from the style sheet. For the most part, however, your use of the `styleSheet` object will be as a reference gateway to individual rules (via the `rules` or `cssRules` array).

Properties

cssRules

Value: Array of rule objects Read-Only

Compatibility: WinIE-, MacIE5, NN6+, Moz+, Safari+, Opera9+, Chrome+

The `cssRules` property returns an array of style sheet rule objects. Strictly speaking, the objects are called `cssRule` objects in the W3C DOM terminology. This property is implemented in MacIE5, but not in the Windows version as of IE7. The list of rule objects is in source code order. The corresponding WinIE4+ property is `rules`.

Example

Use The Evaluator (Chapter 4, "JavaScript Essentials") to look at the `cssRules` property in W3C/Moz browsers or MacIE5. First, view how many rules are in the first `styleSheet` object of the page by entering the following statement into the top text box:

```
document.styleSheets[0].cssRules.length
```

Now use the array with an index value to access one of the rule objects and view the rule object's properties list. Enter the following statement into the bottom text box:

```
document.styleSheets[0].cssRules[1]
```

You use this syntax to modify the style details of an individual rule belonging to the `styleSheet` object.

Keep in mind that you can access the same information in WinIE by changing the example code to use the `rules` property instead of `cssRules`.

Related Items: `rules` property; `cssRule`, `rule` objects

cssText

Value: String Read/Write

Compatibility: WinIE5+, MacIE5, NN-, Moz-, Safari-, Opera-, Chrome-

The `cssText` property contains a string of all style sheet rules included in the `styleSheet` object. Parsing this text in search of particular strings is not wise because the text returned by this property can have carriage returns and other formatting that is not obvious from the text that is assigned to the rules in the style sheet. But you can use this property as a way to completely rewrite the rules of a style sheet in a rather brute-force manner: Assemble a string consisting of all the new rules and assign that string to the `cssText` property. The more formal way of modifying rules (by adding and removing them) is perhaps better form, but there is no penalty for using the `cssText` property if your audience is strictly IE5+.

Example

Use The Evaluator (Chapter 4, "JavaScript Essentials") in WinIE to replace the style rules in one blast via the `cssText` property. Begin by examining the value returned from the property for the initially disabled style sheet by entering the following statement into the top text box:

```
document.styleSheets[0].cssText
```

Next, enable the style sheet so that its rules are applied to the document:

```
document.styleSheets[0].disabled = false
```

Finally, enter the following statement into the top text box to overwrite the style sheet with entirely new rules:

```
document.styleSheets[0].cssText = "p {color:red}"
```

Reload the page after you are finished to restore the original state.

Related Items: `addRule()`, `deleteRule()`, `insertRule()`, `removeRule()` methods

disabled

Value: Boolean Read/Write

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

While the `disabled` property of the `style` element object works with that element only, the `styleSheet` object's `disabled` property works with a `styleSheet` object that comes into the document by a `link` element as well.

Enabling and disabling style sheets is one way to swap different appearance styles for a page, allowing the user to select the preferred style. The page can contain multiple style sheets that control the same selectors, but your script can enable one and disable another to change the overall style. You can even perform this action via the `onload` event handler. For example, if you have separate style sheets for Windows and Mac browsers, you can put both of them in the document, both initially disabled. An `onload` event handler determines the operating system and enables the style sheet tailored for that OS. Unless your style sheets are very extensive, there is little download performance penalty for having both style sheets in the document.

Example

Use The Evaluator (Chapter 4, "JavaScript Essentials") to toggle between the enabled and disabled state of the first `styleSheet` object on the page. Enter the following statement into the top text box:

```
document.styleSheets[0].disabled = (!document.styleSheets[0].disabled)
```

The inclusion of the NOT operator (`!`) forces the state to change from `true` to `false` or `false` to `true` with each click of the Evaluate button.

Related Item: `disabled` property of the `style` element object

`href`

Value: String　　　　Read/Write (See Text)

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera9+, Chrome+

When a style sheet is linked into a document via a `link` element, the `href` property of the `styleSheet` object contains a string with the URL to that file. Essentially, the `href` property of the `link` element is passed along to the `styleSheet` object that loads as a result. In WinIE4+ only, this property is read/write, allowing you to dynamically link in an external style sheet file after the page has loaded. In MacIE and NN6+/Moz, this property is read-only.

Related Item: `link` element object

`id`

Value: String　　　　Read-Only

Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-, Opera-, Chrome-

The `id` property of a `styleSheet` object inherits the `id` property of its containing element (the `style` or `link` element). This can get confusing, because it may appear as though two objects in the document have the same ID. The `id` string, however, can be used as an index to the `document.styleSheets` array in IE4+ (for example, `document.styleSheets["myStyle"]`). NN/Moz does not provide a comparable identifier associated with a `styleSheet` object.

Related Item: `id` property of all element objects

imports

Value: Array of `styleSheet` objects Read-Only

Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-, Opera-, Chrome-

A style sheet can contain one or more `@import` rules to import an external style sheet file into the document. Each imported `styleSheet` object is treated as an `import` object. The `imports` property is a collection of all imported `styleSheet` objects that belong to the current `styleSheet` object. Imported style sheets are not added to the `document.styleSheets` collection, so that references to an imported `styleSheet` object must be through the `document.styleSheets[i].imports[i]` array.

An `import` object is, itself, a `styleSheet` object. All properties and methods applicable to a `styleSheet` object also apply to an `import` object. Therefore, if you want to load a different external style sheet into the page, you can assign the new URL to the imported `styleSheet` object's `href` property:

```
document.styleSheets[0].imports[0].href = "alternate.css";
```

Modifications of this nature work in IE for Windows, but not in MacIE.

Related Item: `styleSheet` object

media

Value: See text Read/Write

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

Cascading Style Sheets can be defined to apply to specific output media, such as the video display screen, printer, and, in the future, devices such as speech synthesizers or Braille generators. A style sheet gets this direction from the `media` attribute of a `style` or `link` element. That value is represented in the `media` property of the `styleSheet` object.

In IE4+, the `media` property value is a string with one of three possible values: `screen`, `printer`, or `all`. The W3C DOM and NN6+ take this one step further by allowing for potentially multiple values being assigned to the `media` attribute. The W3C value is an array of string media names (returned in an object called a `mediaList`).

Related Item: `type` property of the `style` element object

ownerNode

Value: Node reference Read-Only

Compatibility: WinIE-, MacIE-, NN6+, Moz+, Safari+, Opera+, Chrome+

The `ownerNode` property is a reference to the document node in which the `styleSheet` object is defined. For `styleSheet` objects defined inside `style` and `link` elements, the `ownerNode`

property is a reference to that element. The corresponding property in IE4+ is owningElement. Oddly, MacIE5 has an additional, misnamed property called owningNode, whose value equals that of the owningElement property.

Example

Use The Evaluator (Chapter 4, "JavaScript Essentials") with NN6+/Moz, and Gecko-, WebKit-, and Presto-based browsers to inspect the ownerNode of the first styleSheet object in the document. Enter the following statement into the top text box:

```
document.styleSheets[0].ownerNode.tagName
```

The returned value is the style element tag name.

Related Items: ownerRule, owningElement properties

ownerRule

Value: cssRule object Read-Only

Compatibility: WinIE-, MacIE-, NN6+, Moz+, Safari+, Opera+, Chrome+

The ownerRule property applies to a styleSheet object that has been imported into a document via the @import rule. The property returns a reference to the @import rule responsible for loading the external style sheet. There is an interaction between the ownerRule and ownerNode properties in that an imported rule has an ownerRule, but its ownerNode property is null; conversely, a regular styleSheet has an ownerNode, but its ownerRule property is null.

Related Item: ownerNode property

owningElement

Value: Element reference Read-Only

Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-, Opera-, Chrome-

The owningElement property is a reference to the element object in which the styleSheet object is defined. For styleSheet objects defined inside style and link elements, the owningElement property is a reference to that element. The corresponding property in W3C is ownerNode. Oddly, MacIE5 has an additional, misnamed property called owningNode, whose value equals that of the owningElement property.

Example

Use The Evaluator (Chapter 4, "JavaScript Essentials") with IE4+ to inspect the owningElement of the first styleSheet object in the document. Enter the following statement into the top text box:

```
document.styleSheets[0].owningElement.tagName
```

The returned value is the style element tag name.

Related Item: `ownerNode` property

pages

Value: Array of `@page` rules Read-Only

Compatibility: WinIE5.5+, MacIE-, NN-, Moz-, Safari-, Opera-, Chrome-

An `@page` style rule defines the dimensions and margins for printed versions of a web page. The `pages` property returns a collection of `@page` rules contained by the current `styleSheet` object. If no `@page` rules are defined in the style sheet, the array has a length of zero.

While an `@page` rule has the same properties as any rule object, it has one more read-only property, the `pseudoClass` property, which returns any pseudo-class definitions in the rule. For example, the following `@page` rules define different rectangle specifications for the left and right printed pages:

```
@page :left {margin-left:4cm; margin-right:3cm;}
@page :right {margin-left:3cm; margin-right:4cm;}
```

Values for the `pseudoClass` property of these two page rules are `:left` and `:right`, respectively.

To the W3C DOM, an `@page` rule is just another rule object, but one whose `type` property returns `page`.

For more information about the paged media specification, see `http://www.w3.org/TR/CSS2/page.html`.

Related Items: None

parentStyleSheet

Value: `styleSheet` object Read-Only

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera-, Chrome-

An imported style sheet is present thanks to the hosting of a `styleSheet` object created by a `style` or `link` element. That host `styleSheet` object is referenced by the `parentStyleSheet` property. For most `styleSheet` objects (that is, those not imported via the `@import` rule), the `parentStyleSheet` property is `null`. Take note of the distinction between the `parentStyleSheet` property, which points to a `styleSheet` object, and the various properties that refer to the HTML element that "owns" the `styleSheet` object.

Related Items: None

readOnly

Value: Boolean Read-Only

Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-, Opera-, Chrome-

The `readOnly` property's name is a bit misleading. Its Boolean value lets your script know whether the current style sheet was embedded in the document by way of the `style` element or

brought in from an external file via the `link` element or `@import` rule. When embedded by a `style` element, the `readOnly` property is `false`; for style sheets defined outside the page, the property is `true`. But a value of `true` doesn't mean that your scripts cannot modify the style properties. Style properties can still be modified on-the-fly, but of course the changes will not be reflected in the external file from which the initial settings came.

Related Item: `owningElement` property

rules

Value: Array of `rule` objects Read-Only

Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari+, Opera-, Chrome+

The `rules` property returns an array of all rule objects (other than @ rules) defined in the current style sheet. The order of rule objects in the array is based on the source code order of the rules defined in the `style` element or in the external file.

Use the `rules` array as the primary way to reference an individual rule inside a style sheet. If you use a `for` loop to iterate through all rules in search of a particular rule, you will most likely be looking for a match of the rule object's `selectorText` property. This assumes, of course, that each selector is unique within the style sheet. Using unique selectors is good practice, but no restrictions prevent you from reusing a selector name in a style sheet to apply additional style information to the same selector elements.

The corresponding property name for W3C is `cssRules`. MacIE5 and WebKit-based browsers respond to both the `rules` and `cssRules` properties.

Example

Use The Evaluator (Chapter 4, "JavaScript Essentials") with IE4+ to examine the `rules` property of the first `styleSheet` object in the page. First, find out how many rules are in the `styleSheet` object by entering the following statement into the top text box:

```
document.styleSheets[0].rules.length
```

Next, examine the properties of one of the rules by entering the following statement into the bottom text box:

```
document.styleSheets[0].rules[1]
```

You now see all the properties that IE4+ exposes for a rule object.

Related Items: `rule` object; `cssRules` property

title

Value: String Read/Write

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

If you assign a value to the `title` attribute of a `style` element, or to a `link` element that loads a style sheet, that string value filters down to the `title` property of the `styleSheet` object.

You can use the string value as a kind of identifier, but it is not usable as a true identifier as an index to the `styleSheets` array. In visible HTML elements, the `title` attribute usually sets the text that displays with the tooltip over the element. But for the unseen `style` and `link` elements, the attribute has no impact on the rendered display of the page. Therefore, you can use this attribute and corresponding property to convey any string value you want.

Related Item: `title` property of all HTML elements

type

Value: String Read/Write

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

The `type` property of a `styleSheet` object picks up the `type` attribute of the `style` or `link` element that embeds a style sheet into the page. Unless you are experimenting with some new types of style sheet language (assuming it is supported in the browser), the value of the type property is `text/css`.

Related Items: None

Methods

addImport("*URL*"[, *index*])

Returns: Integer

Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-, Opera-, Chrome-

The `addImport()` method lets you add an `@import` rule to a `styleSheet` object. A required first parameter is the URL of the external `.css` file that contains one or more style sheet rules. If you omit the second parameter, the `@import` rule is appended to the end of the rules in the `styleSheet` object. Or, you can specify an integer as the index of the position within the rules collection where the rule should be inserted. The order of rules in a `styleSheet` object can influence the cascading order of overlapping style sheet rules (that is, multiple rules that apply to the same elements).

The value returned by the method is an integer representing the index position of the new rule within the rules collection of the `styleSheet`. If you need subsequent access to the new rule, you can preserve the value returned by the `addImport()` method and use it as the index to the `rules` collection.

Related Item: `addRule()` method

addRule("*selector*", "*styleSpec*"[, *index*])
removeRule(*index*)

Returns: Integer (for `addRule()`)

Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari+, Opera-, Chrome+

The addRule() method appends or inserts a style sheet rule into the current styleSheet object. The first two parameters are strings for the two components of every rule: the selector and the style specification. Any valid selector, including multiple, space-delimited selectors, is permitted. For the style specification, the string should contain the semicolon-delimited list of style attribute:value pairs, but without the curly braces that surround the specification in a regular style sheet rule.

If you omit the last parameter, the rule is appended to the end of the rules collection for the style sheet. Or, you can specify an integer index value signifying the position within the rules collection where the new rule should go. The order of rules in a styleSheet object can influence the cascading order of overlapping style sheet rules (meaning multiple rules that apply to the same elements).

The return value conveys no meaningful information.

To remove a rule from a styleSheet object's rules collection, invoke the removeRule() method. Exercise some care here, because you must have the correct index value for the rule that you want to remove. Your script can use a for loop to iterate through the rules collection, looking for a match of the selectorText property (assuming that you have unique selectors). The index for the matching rule can then be used as the parameter to removeRule(). This method returns no value.

For W3C, the corresponding methods are called insertRule() and deleteRule().

Example

Use The Evaluator (Chapter 4, "JavaScript Essentials") with IE4+ to add a style sheet rule to the first styleSheet object of the page. First, make sure the style sheet is enabled by entering the following statement into the top text box:

```
document.styleSheets[0].disabled = false
```

Next, append a style that sets the color of the textarea element:

```
document.styleSheets[0].addRule("textarea", "color:red")
```

Enter any valid object (such as document.body) into the bottom text box to see how the style has been applied to the textarea element on the page.

Now remove the style, using the index of the last item of the rules collection as the index:

```
document.styleSheets[0].removeRule(document.styleSheets[0].rules.length - 1)
```

The text in the textarea returns to its default color.

Related Items: deleteRule(), insertRule() methods

deleteRule(*index*)
insertRule("*rule*", *index*)

Returns: Integer (for `insertRule()`)

Compatibility: WinIE-, MacIE5, NN6+, Moz+, Safari+, Opera+, Chrome+

The `insertRule()` method appends or inserts a style sheet rule into the current `styleSheet` object. The first parameter is a string containing the style rule as it would normally appear in a style sheet, including the selector and curly braces surrounding the semicolon-delimited list of style `attribute:value` pairs.

You must supply an index location within the `cssRules` array where the new rule is to be inserted. If you want to append the rule to the end of the list, use the `length` property of the `cssRules` collection for the parameter. The order of rules in a `styleSheet` object can influence the cascading order of overlapping style sheet rules (meaning multiple rules that apply to the same elements).

The return value is an index for the position of the inserted rule.

To remove a rule from a `styleSheet` object's `cssRules` collection, invoke the `deleteRule()` method. Exercise some care here, because you must have the correct index value for the rule that you want to remove. Your script could use a `for` loop to iterate through the `cssRules` collection, looking for a match of the `selectorText` property (assuming that you have unique selectors). The index for the matching rule can then be used as the parameter to `deleteRule()`. This method returns no value.

For IE4+, the corresponding methods are called `addRule()` and `removeRule()`.

Example

Use The Evaluator (Chapter 4, "JavaScript Essentials") with NN6+/Moz to add a style sheet rule to the first `styleSheet` object of the page. First, make sure the style sheet is enabled by entering the following statement into the top text box:

```
document.styleSheets[0].disabled = false
```

Next, append a style that sets the color of the `textarea` element:

```
document.styleSheets[0].insertRule("textarea {color:red}",
document.styleSheets[0].cssRules.length)
```

Enter any valid object (such as `document.body`) into the bottom text box to see how the style has been applied to the `textarea` element on the page.

Now remove the style, using the index of the last item of the rules collection as the index:

```
document.styleSheets[0].deleteRule(document.styleSheets[0].cssRules.length - 1)
```

Related Items: `addRule()`, `removeRule()` methods

cssRule and rule Objects

Properties	Methods	Event Handlers
`cssText`		
`parentStyleSheet`		
`readOnly`		
`selectorText`		
`style`		
`type`		

Syntax

Accessing `rule` or `cssRule` object properties:

```
(IE4+)          document.styleSheets[index].rules[index].property
(MacIE5/W3C)    document.styleSheets[index].cssRules[index].property
```

About these objects

The `rule` and `cssRule` objects are different object model names for the same objects. For IE4+, the object is known as a *rule* (and a collection of them, the `rules` collection); for NN6+/Moz-/Safari (and MacIE5), the object follows the W3C DOM recommendation, calling the object a *cssRule* (and a collection of them the `cssRules` collection). For the remainder of this section, they will be referred to generically as the `rule` object.

A rule object has two major components. The first is the selector text, which governs which element(s) are to be influenced by the style rule. The second component is the style definition, with its set of semicolon-delimited `attribute:value` pairs. In both the IE4+ and NN6+/Moz/W3C object models, the style definition is treated as an object: the `style` object, which has tons of properties representing the style attributes available in the browser. The `style` object that belongs to a rule object is precisely the same `style` object that is associated with every HTML element object. Accessing `style` properties of a style sheet rule requires a fairly long reference, as in

```
document.styleSheets[0].rules[0].style.color = "red";
```

but the format follows the logic of JavaScript's dot-syntax to the letter.

Properties

`cssText`

Value: String — Read/Write

Compatibility: WinIE-, MacIE5, NN6+, Moz+, Safari+, Opera+, Chrome+

The `cssText` property returns the full text of the current `cssRule` object. While the text returned from this property can be parsed to locate particular strings, it is easier and more reliable to access individual style properties and their values via the `style` property of a `cssRule` object.

Related Item: `style` property

parentStyleSheet

Value: `styleSheet` object Read-Only

Compatibility: WinIE-, MacIE5, NN6+, Moz+, Safari+, Opera+, Chrome+

The `parentStyleSheet` property is a reference to the `styleSheet` object that contains the current `cssRule` object. The return value is a reference to a `styleSheet` object, from which scripts can read and write properties related to the entire style sheet.

Related Item: `parentRule` property

readOnly

Value: Boolean Read-Only

Compatibility: WinIE5+, MacIE5, NN-, Moz-, Safari-, Opera-, Chrome-

The `readOnly` property's name is a bit misleading. Its Boolean value lets your script know whether the current rule's `styleSheet` was embedded in the document by way of the `style` element or brought in from an external file via the `link` element or `@import` rule. When embedded by a `style` element, the `readOnly` property is false; for style sheets defined outside the page, the property is `true`. But a value of `true` doesn't mean that your scripts cannot modify the style properties. Style properties can still be modified on-the-fly, but of course the changes are not reflected in the external file from which the initial settings came.

Related Item: `styleSheet.readOnly` property

selectorText

Value: String Read-Only

Compatibility: WinIE5+, MacIE5, NN6+, Moz+, Safari+, Opera+, Chrome+

The `selectorText` property returns only the selector portion of a style sheet rule. The value is a string, and if the selector contains multiple, space-delimited items, the `selectorText` value returns the same space-delimited string. For selectors that are applied to classes (preceded by a period) or IDs (preceded by a crosshatch), those leading characters are returned as part of the string as well.

If you want to change the selector for a rule, removing the original rule and adding a new one in its place is better. You can always preserve the `style` property of the original rule and assign the style to the new rule.

Example

Use The Evaluator (Chapter 4, "JavaScript Essentials") to examine the `selectorText` property of rules in the first `styleSheet` object of the page. Enter each of the following statements in the top text box:

```
document.styleSheets[0].rules[0].selectorText
document.styleSheets[0].rules[1].selectorText
```

Compare these values against the source code view for the `style` element in the page.

Related Item: `style` property

style

Value: `style` object Read/Write

Compatibility: WinIE5+, MacIE5, NN6+, Moz+, Safari+, Opera+, Chrome+

The `style` property of a rule (or `cssRule`) is, itself, an object whose properties consist of the CSS style properties supported by the browser. Modifying a property of the `style` object requires a fairly long reference, as in

```
document.styleSheets[0].rules[0].style.color = "red";
```

Any change you make to the rule's `style` properties is reflected in the rendered style of whatever elements are denoted by the rule's selector. If you want to change the style of just one element, access the `style` property of just that element. Style values applied directly to an element override whatever style sheet style values are associated with the element.

Example

Use The Evaluator (Chapter 4, "JavaScript Essentials") to modify a `style` property of one of the `styleSheet` rules in the page. The syntax shown here is for IE4+, but you can substitute the `cssRules` reference for the `rules` collection reference in NN6+/Moz, MacIE5, and W3C browsers if you like.

Begin by reloading the page and making sure the style sheet is enabled. Enter the following statement into the top text box:

```
document.styleSheets[0].disabled = false
```

The first rule is for the `myP` element on the page. Change the rule's `font-size` style:

```
document.styleSheets[0].rules[0].style.fontSize = "20pt"
```

Look over the `style` object properties in the discussion of the `style` object later in this chapter, and have fun experimenting with different style properties. After you are finished, reload the page to restore the styles to their default states.

Related Item: `style` object

type

Value: Integer Read-Only

Compatibility: WinIE-, MacIE-, NN6+, Moz+, Safari+, Opera+, Chrome+

The W3C DOM defines several classes of style sheet rules. To make it easier for a script to identify the kind of `cssRule` it is working with, the `type` property returns an integer whose value is associated with one of the known `cssRule` types. While not all of these rule types may be implemented in current browsers, the complete W3C DOM list is as follows:

Type	Description
0	Unknown type
1	Regular style rule
2	@charset rule
3	@import rule
4	@media rule
5	@font-face rule
6	@page rule

Most of the style sheet rules you work with are type 1. To learn more about these rule types, consult the W3C specification for CSS at `http://www.w3.org/TR/CSS2`.

Related Items: None

currentStyle, runtimeStyle, and style Objects

Properties	Methods	Event Handlers
See the following text.		

Syntax

Accessing `currentStyle`, `runtimeStyle`, or `style` object properties:

```
(IE4+/W3C)       elementReference.style.property
(IE4+/W3C)       document.styleSheets[index].style.property
(IE5+)           elementReference.currentStyle.property
(IE5.5+)         elementReference.runtimeStyle.property
```

About these objects

All three of these objects — `currentStyle`, `runtimeStyle`, and `style` — return an object that contains dozens of properties related to style sheet specifications associated either with a `styleSheet` object (for the `style` object only) or any rendered HTML element object. With the browser page reflow facilities of modern browsers, changes made to the properties of the `style` and IE-specific `runtimeStyle` objects are reflected immediately by the rendered content on the page.

The primary object, the `style` object, is accessed as a property of an HTML element object. It is vital to remember that style properties of an HTML element are reflected by the `style` object only if the specifications are made via the `style` attribute inside the element's tag. If your coding style requires that style sheets be applied via `style` or `link` tags, and if your scripts need to access the `style` property values as set by those style sheets, then you must read the properties of the effective style sheet through the read-only `currentStyle` property (available in IE5+) or the W3C DOM `window.getComputedStyle()` method (NN6+/Moz, and Gecko-, WebKit-, and Presto-based browsers).

IE's `currentStyle` object does not have precisely the same properties as its `style` object. Missing from the `currentStyle` object are the properties that contain combination values, such as `border` or `borderBottom`. On the other hand, `currentStyle` provides separate properties for each of the sides of a clipping rectangle (`clipTop`, `clipRight`, `clipBottom`, and `clipLeft`), which the `clip` property does not provide.

Microsoft introduced one more flavor of style object — the `runtimeStyle` object — in IE5.5. This object lets scripts override any style property that is set in a style sheet or via the `style` attribute. In other words, the `runtimeStyle` object is like a read/write version of `currentStyle`, except that assigning a new value to one of its properties does not modify the style sheet definition or the value assigned in a `style` attribute. By and large, however, your scripts will modify the `style` property of an element to make changes, unless you modify styles by enabling and disabling style sheets (or changing the `className` property of an element so that it is under the control of a different selector).

Style properties

If you add up all the `style` object properties available in browsers starting with IE4 and NN6/Moz, you have a list approximately 180 properties long. A sizable percentage are in common among all browsers and are scriptable versions of W3C Cascading Style Sheet properties. The actual CSS property names are frequently script-unfriendly in that multiple-word properties have hyphens in them, such as `font-size`. JavaScript identifiers do not allow hyphens, so multiple-word properties are converted to camelCase versions, such as `fontSize`.

Not all style properties are supported by all browsers that have the `style` object in their object models. Microsoft, in particular, has added many properties that are sometimes unique to IE, and sometimes unique to just IE for Windows. On the Mozilla side, you find some properties that appear to be supported by the `style` object, but the browser doesn't genuinely support the attributes. For example, the CSS specification defines several attributes that enhance the

delivery of content rendered through a speech synthesizer. Although Firefox doesn't qualify, the Gecko browser engine at its core could be adapted to such a browser. Therefore, if you see a property in the following listings that doesn't make sense to you, test it out in the compatible browsers to verify that it works as you need it. You will also find some properties that are proprietary to Mozilla-based browsers — properties that begin with `moz`. These properties are preliminary implementations of as yet unreleased CSS Level 3 properties. The `moz` prefix lets you use these properties today without conflicting with future, sanctioned implementations of the properties (which won't have the `moz` prefix). When specifying these properties in CSS syntax for your style sheets, the properties begin with the special prefix `-moz-`, as in `-moz-opacity` (and the scripted equivalent, `mozOpacity`).

Some browsers also expose advanced `style` object properties to scripters, when, in fact, they are not genuinely supported in the browser. For example, an inspection of the `style` object for MacIE5 and NN6+/Moz shows a `quotes` property, which matches the `quotes` style property in the W3C CSS2 specification. But in truth, the `quotes` style property cannot be set by script in these browsers. When you see that a property is supported by MacIE5 and NN6+/Moz but not others, testing out the `style` property (and the style sheet attribute as well) in The Evaluator is a good idea before attempting to employ the property in your application.

With so many properties associated with an object, it may be difficult to locate the specific property you need for a particular style effect. To help you locate properties, the listings that follow are divided into functional categories, ordered by popularity:

Category	Description
Text & Fonts	Font specifications, text rendering, text alignment
Inline Display & Layout	Element flow, alignment, and display
Positioning	Explicit positioning of ''layers''
Background	Background images and colors
Borders & Edges	Borders, padding, and margins around elements
Lists	Details for `ul` and `ol` elements
Scroll bars	Scroll bar colors (WinIE5.5+ only)
Tables	Details for `table` elements and components
Printing	Page breaks and alignment for printed pages
Miscellaneous	Odds and ends
Aural	For rendering via speech-synthesis

Property values

All `style` object property values are strings. Moreover, many groups of style properties share the same format for their values. Knowing the formats for the frequently used values is helpful.

The purpose of this chapter is not to teach you about style sheets but to show you how to script them. Therefore, if you see unfamiliar terminology here, consult online or print instructional material about Cascading Style Sheets.

Length

Values for length cover a wide range, but they all define an amount of physical space in the document. Because content can be displayed on a video monitor or printed on a sheet of paper, any kind of length value should include a unit of measure as well as a quantity. One group of units (`px`, `em`, `ex`) are considered *relative* units, because the precise size depends on factors beyond the control of the style sheet (for example, the pixel density of the display) or units set by elements with more global scope (for example, a `p` element's margin `em` length is dependent upon the `body` element's font-size setting). *Absolute* units (`in`, `cm`, `mm`, `pi`, `pt`) are more appropriate for printed output. Length units are referenced in script according to the following table:

Unit	Script Version	Example
Pixel	`px`	`14px`
Em	`em`	`1.5em`
Ex	`ex`	`1.5ex`
Inch	`in`	`3.0in`
Centimeter	`cm`	`4.0cm`
Millimeter	`mm`	`40mm`
Pica	`pi`	`72pi`
Point	`pt`	`14pt`

A length value can also be represented by a percentage as a string. For example, the `lineHeight` style for a paragraph would be set to `120%` of the font size established for the paragraph by the following statement:

```
document.getElementById("myP").style.lineHeight = "120%";
```

Style inheritance — an important CSS concept — often has significant impact on style properties whose values are lengths.

Color

Values for colors can be one of three types:

- RGB values (in a few different formats)
- plain-language versions of the color names
- plain-language names of system user interface items

RGB values can be expressed as hexadecimal values. The most common way is with a crosshatch character followed by six hex numbers, as in `#ff00ff` (letters can be uppercase or lowercase). A special shortcut is also available to let you specify three numbers with the assumption that they will be expanded to pairs of numbers. For example, a color of `#f0f` is expanded internally to be `#ff00ff`.

An alternative RGB expression is with the `rgb()` prefix and three numbers (from 0 to 255) or percentages corresponding to the red, green, and blue components of the color. Here are a couple of examples:

```
document.styleSheets[0].rules[0].style.color = "rgb(0, 255, 0)";
document.styleSheets[0].rules[0].style.color = "rgb(0%, 100%, 0%)";
```

Browsers also respond to a long list of plain-language color names originally adopted from the X Window System palette by Netscape, and now known as X11 colors. You can see the list with sample colors at `http://en.wikipedia.org/wiki/X11_color_names`. Not all of those colors are necessarily part of what are known as "web safe" colors. For a demonstration of web safe colors, visit `http://www.lynda.com/hexh.html`. Of course, it's worth noting that "web safe" colors only enter the picture when a user is limited to an 8-bit (256) color display, which is rare these days.

The last category of color values references user interface pieces, many of which are determined by the user's control panel for video display. The string values correspond to recognizable UI components (also called system colors), as follows:

```
activeborder
activecaption
appworkspace
background
buttonface
buttonhighlight
buttonshadow
buttontext
captiontext
graytext
highlight
highlighttext
inactiveborder
inactivecaption
inactivecaptiontext
infobackground
infotext
menu
menutext
scrollbar
threeddarkshadow
threedface
threedhighlight
```

```
threedlightshadow
threedshadow
window
windowframe
windowtext
```

Using these color settings may be risky for public sites, because you are at the mercy of the color settings the user has chosen. For a corporate environment where system installations and preferences are strictly controlled, these values could help define a safe color scheme for your pages.

Rectangle sides

Many style properties control the look of sides of rectangles (for example, thickness of a border around a block element). In most cases, the style values can be applied to individual sides or combinations of sides, depending on the number of values supplied to the property. The number of values affects the four sides of the rectangle according to the following matrix:

Number of Values	Impact
1	All four sides set to the one value
2	Top and bottom sides set to first value; left and right sides set to second value
3	Top side set to first value; left and right sides set to second value; bottom side set to third value
4	Top, right, bottom, and left sides set to individual values in that order

For example, to set the border color of an element so that all sides are red, the syntax is

```
elementRef.style.borderColor = "red";
```

To set the top and bottom to red but the left and right to green, the syntax is

```
elementRef.style.borderColor = "red green";
```

Properties that accept these multiple values cover a wide range of styles. Values may be colors, lengths, or selections from a fixed list of possible values.

Combination values

Another category of style values includes properties that act as shortcuts for several related properties. For example, the `border` property encompasses the `borderWidth`, `borderStyle`, and `borderColor` properties. This is possible because very different classes of values represent the three component properties: `borderWidth` is a length; `borderStyle` is based on a fixed list of values; and `borderColor` is a color value. Therefore, you can specify one or more of these property values (in any order), and the browser knows how to apply the values to the detailed

subproperty. Only one value is permitted for any one of these subproperties, which means that if the property is one of the four-sided styles described in the previous section, the value is applied to all four sides equally.

For example, setting the `border` property to a single value, as in

```
elementRef.style.border = "blue";
```

is the same as setting the border color:

```
elementRef.style.borderColor = "blue";
```

But if you set multiple items, as in

```
elementRef.style.border = "groove blue 3px";
```

then you have set the equivalent of the following three statements:

```
elementRef.style.borderStyle = "groove";
elementRef.style.borderColor = "blue";
elementRef.style.borderWidth = "3px";
```

In the property descriptions that follow, these combination values are denoted by their scripted property names and the OR (||) operator, as in

```
border = "borderStyle || borderColor || borderWidth";
```

URLs

Unlike other property values containing URLs, a `style` property requires a slightly different format. This format includes the `url()` prefix, with the actual URL (relative or absolute) located inside the parentheses. The URL itself is not quoted, but the entire property value is, as in

```
elementRef.style.backgroundImage = "url(chainlink.jpg)";
```

URLs should not have any spaces in them, but if they do, use the URL-encoded version for the file specification: convert spaces to `%20`. This format distinguishes a URL value from some other string value for shortcut properties.

Text and font properties

`color`

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

Controls: Foreground color of an element, primarily used to assign color to text. May also affect edges and highlights of other elements in some browsers.

Value: Color specification

Example: `elementRef.style.color = "rgb(#22FF00)";`

font

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

Controls: Up to six font-related style properties.

Value: Combination values: `fontStyle` || `fontVariant` || `fontWeight` || `fontSize` || `lineHeight` || `fontFamily`. See individual properties for their value formats

Example: `elementRef.style.font = "bold sans-serif 16px";`

fontFamily

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

Controls: Font family to be applied to an element in order of priority.

Value: Comma-delimited list of font families to be applied to element, starting with the most preferred font family name. You can also use one of several generic family names that rely on the browser to choose the optimal font to match the class: `serif` | `sans-serif` | `cursive` | `fantasy` | `monospace`. Not all browsers support all constants, but `serif`, `sans-serif`, and `monospace` are commonly implemented.

Example: `elementRef.style.fontFamily = "Bauhaus 93, Arial, monospace";`

fontSize

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

Controls: Size of the characters of the current font family.

Value: Lengths (generally px or pt values); relative size constants: `larger` | `smaller`; absolute size constants: `xx-small` | `x-small` | `small` | `medium` | `large` | `x-large` | `xx-large`

Examples: `elementRef.style.fontSize = "16px"; elementRef.style.fontSize = "small";`

fontSizeAdjust

Compatibility: WinIE-, MacIE5, NN6+, Moz+, Safari-, Opera-, Chrome-

Controls: Aspect value of a secondary font family so that it maintains a similar character height as the primary font family.

Value: Number (including floating-point value) or `none`

Example: `elementRef.style.fontSizeAdjust = "1.05";`

fontStretch

Compatibility: WinIE-, MacIE5, NN6+, Moz+, Safari-, Opera-, Chrome-

Controls: Rendered width of a font's characters.

Value: Constant: `ultra-condensed` | `extra-condensed` | `condensed` | `semi-condensed` | `semi-expanded` | `expanded` | `extra-expanded` | `ultra-expanded`; or: `wider` | `narrower` | `inherit` | `normal`

Example: `elementRef.style.fontStretch = "expanded";`

fontStyle

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

Controls: Italic style of characters.

Value: Constant: `normal` | `italic` | `oblique` | `inherit`

Example: `elementRef.style.fontStyle = "italic";`

fontVariant

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari1.3+, Opera+, Chrome+

Controls: Rendering characters as small caps.

Value: Constant: `normal` | `small-caps` | `inherit`

Example: `elementRef.style.fontVariant = "small-caps";`

fontWeight

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

Controls: Rendering characters in bold or light weights. Fonts that support numbered gradations can be controlled by those numbers. Normal = `400`; bold = `700`.

Value: Constant: `bold` | `bolder` | `lighter` | `normal` | `100` | `200` | `300` | `400` | `500` | `600` | `700` | `800` | `inherit`

Example: `elementRef.style.fontWeight = "bold";`

letterSpacing

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

Controls: Spacing between characters. Used to override a font family's own characteristics.

Value: Length (usually em units, relative to current font size); Constant: `normal` | `inherit`

Example: `elementRef.style.letterSpacing = "1.2em";`

lineBreak

Compatibility: WinIE5+, MacIE-, NN-, Moz-, Safari-, Opera-, Chrome-

Controls: Line-break rules for Japanese text content.

Value: Constant: `normal` | `strict`

Example: `elementRef.style.lineBreak = "strict";`

lineHeight

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

Controls: Height of the rectangular space that holds a line of text characters.

Value: Length (usually em units, relative to current font size); number (a multiplier on the inherited line height); percentage (relative to inherited line height); constant: `normal | inherit`

Example: `elementRef.style.lineHeight = "1.1";`

quotes

Compatibility: WinIE-, MacIE5, NN6+, Moz+, Safari-, Opera+, Chrome-

Controls: Characters to be used for quotation marks.

Value: Space-delimited pairs of open and close quotation symbols; Constant: `none | inherit`

Example: `elementRef.style.quotes = "« »"`

rubyAlign

Compatibility: WinIE5+, MacIE5, NN-, Moz-, Safari-, Opera-, Chrome-

Controls: Alignment of ruby text within a `ruby` element.

Value: Constant: `auto | left | center | right | distribute-letter | distribute-space | line-edge`

Example: `RUBYelementRef.style.rubyAlign = "distribute=letter";`

rubyOverhang

Compatibility: WinIE5+, MacIE5, NN-, Moz-, Safari-, Opera-, Chrome-

Controls: Overhang of ruby text within a `ruby` element.

Value: Constant: `auto | whitespace | none`

Example: `RUBYelementRef.style.rubyOverhang = "whitespace";`

rubyPosition

Compatibility: WinIE5+, MacIE5, NN-, Moz-, Safari-, Opera-, Chrome-

Controls: Placement of ruby text with respect to the `ruby` element's base text.

Value: Constant: `above | inline`

Example: `RUBYelementRef.style.rubyPosition = "inline";`

textAlign

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

Controls: Horizontal alignment of text with respect to its containing element.

Value: Constant: `center | justify | left | right`

Example: `elementRef.style.textAlign = "center";`

textAlignLast

Compatibility: WinIE5.5+, MacIE-, NN-, Moz-, Safari-, Opera-, Chrome-

Controls: Horizontal alignment of last line of text in a paragraph.

Value: Constant: `auto | center | justify | left | right`

Example: `elementRef.style.textAlignLast = "justify";`

textAutospace

Compatibility: WinIE5+, MacIE-, NN-, Moz-, Safari-, Opera-, Chrome-

Controls: Extra spacing between ideographic and non-ideographic text.

Value: Constant: `none | ideograph-alpha | ideograph-numeric | ideograph-parenthesis | ideograph-space`

Example: `elementRef.style.textAutospace = "ideograph=alpha";`

textDecoration

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

Controls: Display of underline, overline, or line-through with text.

Value: Constant: `none | blink | line-through | overline | underline`

Example: `elementRef.style.textDecoration = "underline";`

textDecorationBlink
textDecorationLineThrough
textDecorationNone
textDecorationOverline
textDecorationUnderline

Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-, Opera-, Chrome-

Controls: Individual text decoration characteristics for text, allowing for multiple decorations to be applied to the same text.

Value: Boolean (not strings): `true | false`

Example: `elementRef.style.textDecorationUnderline = true;`

textIndent

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

Controls: Amount of indentation for the first line of a block text element (for example, `p`).

Value: Length (negative values for outdenting); percentage (relative to inherited value)

Example: `elementRef.style.textIndent = "2.5em";`

textJustify

Compatibility: WinIE5+, MacIE5, NN-, Moz-, Safari-, Opera-, Chrome-

Controls: Additional detailed specifications for an element whose `textAlign` property is set to `justify`.

Value: Constant: `auto` | `distribute` | `distribute-all-lines` | `distribute-center-last` | `inter-cluster` | `inter-ideograph` | `inter-word` | `kashida` | `newspaper`

Example: `elementRef.style.textJustify = "distribute";`

textJustifyTrim

Compatibility: WinIE5+, MacIE5, NN-, Moz-, Safari-, Opera-, Chrome-

Reserved for future use.

textKashidaSpace

Compatibility: WinIE5.5+, MacIE-, NN-, Moz-, Safari-, Opera-, Chrome-

Controls: Ratio of kashida expansion to white space expansion for Arabic writing systems.

Value: Percentage

Example: `elementRef.style.textKashidaSpace = "90%";`

textOverflow

Compatibility: WinIE6+, MacIE-, NN-, Moz-, Safari1.3+, Opera-, Chrome+

Controls: Whether an ellipsis (...) is displayed at the end of a line of overflowed text to indicate that more text is available.

Value: Constant: `clip` | `ellipsis`

textShadow

Compatibility: WinIE-, MacIE5, NN6+, Moz+, Safari1.2+, Opera+, Chrome+

Controls: Shadow rendering around text characters. Note: The `style` attribute for this property is not implemented in MacIE5 or NN6+/Moz, but the property is listed as valid for the `style` object.

Value: Each shadow specification consists of an optional color and three space-delimited length values (horizontal shadow offset, vertical shadow offset, blur radius length). Multiple shadow specifications are comma-delimited.

textTransform

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

Controls: Case rendering of the text (without altering the case of the original text).

Value: Constant: `none` | `capitalize` | `lowercase` | `uppercase`

Example: `elementRef.style.textTransform = "uppercase";`

textUnderlinePosition

Compatibility: WinIE5.5+, MacIE-, NN-, Moz-, Safari-, Opera-, Chrome-

Controls: Whether an underline text decoration is displayed above or below the text. Seems redundant with `textDecorationUnderline` and `textDecorationOverline`.

Value: Constant: `above | below`

Example: `elementRef.style.textUnderlinePosition = "above";`

unicodeBidi

Compatibility: WinIE5+, MacIE5, NN6+, Moz+, Safari+, Opera+, Chrome+

Controls: Within bidirectional text (for example, English and Arabic), to what extent an alternate direction text block is embedded within the outer element.

Value: Constant: `normal | embed | bidi-override`

Example: `elementRef.style.unicodeBidi = "embed";`

whiteSpace

Compatibility: WinIE5.5+, MacIE5, NN6+, Moz+, Safari+, Opera+, Chrome+

Controls: Treatment of white space characters within an element's source code.

Value: Constant: `normal | nowrap | pre`

Example: `elementRef.style.whiteSpace = "nowrap";`

wordBreak

Compatibility: WinIE5+, MacIE-, NN-, Moz-, Safari+, Opera-, Chrome+

Controls: Word breaking characteristics, primarily for Asian-language text or text containing a mixture of Asian and Latin characters.

Value: Constant: `normal | break-all | keep-all`

Example: `elementRef.style.wordBreak = "break-all";`

wordSpacing

Compatibility: WinIE6+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

Controls: Spacing between words.

Value: Length (usually in em units); Constant: `normal`

Example: `elementRef.style.wordSpacing = "1em";`

wordWrap

Compatibility: WinIE5.5+, MacIE-, NN-, Moz-, Safari1.3+, Opera-, Chrome+

Controls: Word wrapping characteristics of text in a block element, explicitly-sized inline element, or positioned element.

Value: Constant: `normal` | `break-word`

Example: `elementRef.style.wordWrap = "break-word";`

writingMode

Compatibility: WinIE5.5+, MacIE-, NN-, Moz-, Safari-, Opera-, Chrome-

Controls: Direction of content flow (left-to-right/top-to-bottom or top-to-bottom/right-to-left, as in some Asian languages).

Value: Constant: `lr-tb` | `tb-rl`

Example: `elementRef.style.writingMode = "tb-rl";`

Inline display and layout properties

clear

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

Controls: Layout orientation of an element with respect to a neighboring floating element.

Value: Constant: `both` | `left` | `none` | `right`

Example: `elementRef.style.clear = "right";`

clip

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

Controls: The clipping rectangle of an element (that is, the position of the rectangle through which the user sees an element's content).

Value: `rect(topLength, rightLength, bottomLength, leftLength)` | `auto`

Example: `elementRef.style.clip = "rect(10px, 300px, 200px, 0px)";`

clipBottom
clipLeft
clipRight
clipTop

Compatibility: WinIE5+, MacIE-, NN-, Moz-, Safari-, Opera-, Chrome-

Controls: Individual edges of the clipping rectangle of an element. These properties are read-only properties of the `currentStyle` object.

Value: Length | `auto`

Example: var leftEdge = *elementRef*.currentStyle.clipLeft;

content

Compatibility: WinIE-, MacIE5, NN6+, Moz+, Safari1.3+, Opera+, Chrome+

Controls: The content rendered in addition to the element, usually to be applied with a `:before` or `:after` pseudo-class. This feature will become more useful when CSS counters are implemented in browsers. They'll provide automatic section or paragraph numbering. While the CSS equivalent is implemented in NN7/Moz and Gecko-, WebKit-, and Presto-based browsers, changes to the scripted property are not rendered.

Value: See `http://www.w3.org/TR/CSS2/generate.html#propdef-content`

counterIncrement

Compatibility: WinIE-, MacIE5, NN6+, Moz1.8+, Safari+, Opera+, Chrome+

Controls: The jumps in counter values to be displayed via the `content` style property.

Value: One or more pairs of counter identifier and integers

counterReset

Compatibility: WinIE-, MacIE5, NN6+, Moz1.8+, Safari+, Opera9+, Chrome+

Controls: Resets a named counter for content to be displayed via the `content` style property.

Value: One or more pairs of counter identifiers and integers.

cssFloat

Compatibility: WinIE-, MacIE5, NN6+, Moz+, Safari+, Opera+, Chrome+

Controls: Horizontal alignment of an element that allows other content to wrap around the element (usually text wrapping around an image). Corresponds to the CSS `float` style attribute. See also the `styleFloat` property, later in this chapter. Floating (non-positioned) elements follow a long sequence of rules for their behavior, detailed at `http://www.w3.org/TR/CSS2/visuren.html#propdef-float`.

Value: Constant: `left | right | none`

Example: `elementRef.style.cssFloat = "right";`

cursor

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari1.3+, Opera+, Chrome+

Controls: The icon used for the cursor on the screen from a library of system-generated cursors. The CSS2 specification defines syntax for downloadable cursors, but this feature is not implemented in NN6+/Moz. You can change this style property only if a `:hover` pseudo-class is initially defined for the element.

Value: Constant: `auto | crosshair | default | e-resize | help | move | n-resize | ne-resize | nw-resize | pointer | s-resize | se-resize | sw-resize | text | w-resize | wait`. New values for IE6 are: `all-scroll | col-resize | no-drop | not-allowed | progress | row-resize | url | vertical-text`. Mozilla-based browsers include: `alias | cell | context-menu | copy | count-down | count-up | count-up-down | grab | grabbing | spinning`.

Example: `elementRef.style.cursor = "hand";`

`direction`

Compatibility: WinIE5+, MacIE5, NN6+, Moz+, Safari+, Opera+, Chrome+

Controls: Layout direction (left-to-right or right-to-left) of inline text (same as `dir` attribute of an element).

Value: Constant: `ltr | rtl`

Example: `elementRef.style.direction = "rtl";`

`display`

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

Controls: Whether an element is displayed on the page and in which display mode. Content surrounding an undisplayed element cinches up to occupy the undisplayed element's space — as if the element didn't exist for rendering purposes (see the `visibility` property for a different approach). Commonly used to hide or show segments of a graphical tree structure. Also used to direct the browser to display an element as an inline or block-level element. Some special-purpose values are associated with specific element types (for example, lists, table cells, and so on).

Value: Constant: `block | compact | inline | inline-table | list-item | none | run-in | table | table-caption | table-cell | table-column-group | table-footer-group | table-header-group | table-row | table-row-group`

Example: `elementRef.style.display = "none"; // removes element from page`

`filter`

Compatibility: WinIE4+, MacIE-, NN-, Moz-, Safari-, Opera-, Chrome-

Controls: Rendering effects on static content and on transitions between hiding and showing elements. Microsoft made a massive overhaul of the `filter` stylesheet syntax in WinIE5.5 (using the `DXImageTransform` ActiveX control). Scripting transitions require several steps to load the transition and actions before playing the transition. Use `style.filter` to read or write the entire filter specification string; use the `elem.styles[i]` object to access individual filter properties. See the discussion of the `filter` object later in this chapter.

Value: Filter specification as string

Example: `var filterSpec = elementRef.style.filter = "alpha(opacity=50) flipH()";`

`layoutGrid`

Compatibility: WinIE5+, MacIE-, NN-, Moz-, Safari-, Opera-, Chrome-

Controls: Page grid properties (primarily for Asian-language pages).

Value: Combination values: `layoutGridMode || layoutGridType || layoutGridLine || layoutGridChar`. See individual properties for their value formats.

Example: `elementRef.style.layoutGrid = "2em fixed";`

layoutGridChar

Compatibility: WinIE5+, MacIE-, NN-, Moz-, Safari-, Opera-, Chrome-

Controls: Size of the character grid (Asian languages).

Value: Length; Percentage; Constant: `none` | `auto`

Example: `elementRef.style.layoutGridChar = "2em";`

layoutGridLine

Compatibility: WinIE5+, MacIE-, NN-, Moz-, Safari-, Opera-, Chrome-

Controls: Line height of the grid (Asian languages).

Value: Length; Percentage; Constant: `none` | `auto`

Example: `elementRef.style.layoutGridLine = "110%";`

layoutGridMode

Compatibility: WinIE5+, MacIE-, NN-, Moz-, Safari-, Opera-, Chrome-

Controls: One- or two-dimensional grid (Asian languages).

Value: Constant: `both` | `none` | `line` | `char`

Example: `elementRef.style.layoutGridMode = "both";`

layoutGridType

Compatibility: WinIE5+, MacIE-, NN-, Moz-, Safari-, Opera-, Chrome-

Controls: Type of grid for text content (Asian languages).

Value: Constant: `loose` | `strict` | `fixed`

Example: `elementRef.style.layoutGridType = "strict";`

markerOffset

Compatibility: WinIE-, MacIE-, NN6+, Moz+, Safari-, Opera-, Chrome-

Controls: Distance between the edges of a marker box (content whose display is of a marker type) and a block-level element's box. Note: The CSS property is not implemented in MacIE5 or NN6+/Moz, but the property is listed as valid for a `style` object.

Value: Length; Constant: `auto`

Example: `elementRef.style.markerOffset = "2em";`

marks

Compatibility: WinIE-, MacIE5, NN6+, Moz+, Safari-, Opera-, Chrome-

Controls: Rendering of crop marks and the like on the printed page. Note: The CSS property is not implemented in MacIE5 or NN6+/Moz, but the property is listed as valid for a `style` object.

Value: Constant: `crop` || `cross` | `none`

Example: `elementRef.style.marks = "crop";`

maxHeight
maxWidth
minHeight
minWidth

Compatibility: WinIE (see text), MacIE-, NN6+, Moz+, Safari1+, Opera+, Chrome+

Controls: Maximum or minimum height or width of an element. Microsoft supports `maxHeight` only starting with IE7.

Value: Length; Percentage; Constant (for max properties only): `none`

Example: `elementRef.style.maxWidth = "300px";`

mozOpacity

Compatibility: WinIE-, MacIE-, NN7+, Moz+, Safari-, Opera-, Chrome-

Controls: The level of opacity (transparency) of the element as a percentage; the lower the value, the more transparent the element becomes (`0%` or `0.0` is completely transparent, while `100%` or `1.0` is completely opaque).

Value: Percentage, or numeric value between `0.0` and `1.0`

Example: `elementRef.style.mozOpacity = "75%";`

opacity

Compatibility: WinIE-, MacIE-, NN-, Moz1.7.2+, Safari1.2+, Opera+, Chrome+

Controls: The level of opacity (transparency) of the element as a percentage; the lower the value, the more transparent the element becomes (`0%` or `0.0` is completely transparent, while `100%` or `1.0` is completely opaque). Unlike `mozOpacity`, which is unique to Mozilla, `opacity` is the official W3C standard for opacity.

Value: Percentage, or numeric value between `0.0` and `1.0`

Example: `elementRef.style.opacity = "25%";`

overflow

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

Controls: The rendering of a block-level element's content when its native rectangle exceeds that of its next outermost rectangular space. A `hidden` overflow clips the block-level content; a `scrolled` overflow forces the outermost rectangle to display scroll bars so that users can scroll around the block-level element's content; a `visible` overflow causes the block-level element to extend beyond the outermost container's rectangle (indeed, "overflowing" the container).

Value: Constant: `auto` | `hidden` | `scroll` | `visible`

Example: `elementRef.style.overflow = "scroll";`

overflowX
overflowY

Compatibility: WinIE5+, MacIE-, NN-, Moz1.8+, Safari1.2, Opera+, Chrome+

Controls: The rendering of a block-level element's content when its native rectangle exceeds the width (`overflowX`) or height (`overflowY`) of its next outermost rectangular space. A hidden overflow clips the block-level content; a scrolled overflow forces the outermost rectangle to display scroll bars so that users can scroll around the block-level element's content; a visible overflow causes the block-level element to extend beyond the outermost container's rectangle (indeed, "overflowing" the container).

Value: Constant: `auto` | `hidden` | `scroll` | `visible`

Example: `elementRef.style.overflowX = "scroll";`

styleFloat

Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-, Opera+, Chrome-

Controls: Horizontal alignment of an element that allows other content to wrap around the element (usually text wrapping around an image). Corresponds to the CSS `float` style attribute. See also the `cssFloat` property, earlier in the chapter. Floating (non-positioned) elements follow a long sequence of rules for their behavior, detailed at `http://www.w3.org/TR/CSS2/visuren.html#propdef-float`.

Value: Constant: `left` | `right` | `none`

Example: `elementRef.style.styleFloat = "right";`

verticalAlign

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari1.2+, Opera+, Chrome+

Controls: How inline and table cell content aligns vertically with surrounding content. Not all constant values are supported by all browsers.

Value: Constant: `baseline` | `bottom` | `middle` | `sub` | `super` | `text-bottom` | `text-top` | `top`; Length; Percentage

Example: `elementRef.style.verticalAlign = "baseline";`

visibility

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

Controls: Whether an element is displayed on the page. The element's spot is preserved as empty space when the element is hidden. To cinch up surrounding content, see the `display` property. This property is used frequently for hiding and showing positioned elements under script control.

Value: Constant: `collapse` | `hidden` | `visible`

Example: `elementRef.style.visibility = "hidden";`

width

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

Controls: Horizontal dimension of a block-level element. Earlier browsers exhibit unexpected behavior when nesting elements that have their `width` style properties set.

Value: Length; Percentage; Constant: `auto`

Example: `elementRef.style.width = "200px";`

zoom

Compatibility: WinIE5.5+, MacIE-, NN-, Moz-, Safari-, Opera-, Chrome-

Controls: Magnification factor of a rendered element.

Value: Constant: `normal`; Percentage (where `100%` is normal); floating-point number (scale multiplier, where 1.0 is normal)

Example: `elementRef.style.zoom = ".9";`

Positioning properties

See Chapter 43, "Positioned Objects", on the CD-ROM for coding examples of positioned elements and their `style` properties.

bottom
right

Compatibility: WinIE5+, MacIE5, NN6+, Moz+, Safari+, Opera+, Chrome+

Controls: The offset measure of a positioned element from its containing rectangle's bottom and right edges, respectively. In practice, you should adjust the size of a positioned element via the style's `height` and `width` properties.

Value: Length; Percentage; Constant: `auto`

Example: `elementRef.style.bottom = "20px";`

left
top

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

Controls: The offset measure of a positioned element from its containing rectangle's left and top edges, respectively. In practice, use these properties to position an element under script control. To position an absolute-positioned element atop an inline element, calculate the position of the inline element via the `offsetTop` and `offsetLeft` properties with some browser-specific adjustments, as shown in Chapter 43, "Positioned Objects."

Value: Length; Percentage; Constant: `auto`

Example: `elementRef.style.top = "250px";`

***elementRefObject*.style.height**

height
width

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

Controls: Height and width of a block-level element's box. Used most commonly to adjust the dimensions of a positioned element (Chapter 43, "Positioned Objects").

Value: Length; Percentage; Constant: `auto`

Example: `elementRef.style.height = "300px";`

pixelBottom
pixelHeight
pixelLeft
pixelRight
pixelTop
pixelWidth

Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari+, Opera+, Chrome+

Controls: Integer pixel values for (primarily positioned) elements. Because the non-pixel versions of these properties return strings that also contain the unit measure (for example, `30px`), these properties let you work exclusively in integers for pixel units. The same can be done cross-platform by using `parseInt()` on the non-pixel versions of these properties. The `pixelBottom` and `pixelRight` properties are not supported by MacIE4.

Value: Integer

Example: `elementRef.style.pixelTop = elementRef.style.pixelTop + 20;`

posBottom
posHeight
posLeft
posRight
posTop
posWidth

Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari+, Opera+, Chrome+

Controls: Numeric values for (primarily positioned) elements in whatever unit was specified by the corresponding `style` attribute. Because the non-pos versions of these properties return strings that also contain the unit measure (for example, `1.2em`), these properties let you work exclusively in numeric values of the same units in which the style was originally defined. The same can be done cross-platform by using `parseFloat()` on the non-pixel versions of these properties.

Value: Integer

Example: `elementRef.style.posTop = elementRef.style.posTop + 0.5;`

position

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

Controls: The type of positioning to be applied to the element. An element that is not explicitly positioned is said to be *static*. A relative-positioned element appears in its normal page flow location but can be explicitly positioned relative to that location. An absolute-positioned element must have its `top` and `left` style attributes assigned to give the element a set of coordinates for its location. MacIE5 and NN6+/Moz, and Gecko-, WebKit-, and Presto-based browsers also allow for a fixed positioned element, which remains at its designated position in the browser window, even if the page scrolls (for example, for a watermark effect). You cannot use scripts to change between positioned and non-positioned style settings. See Chapter 43 for more information on positioned elements.

Value: Constant: `absolute` | `fixed` | `relative` | `static`

Example: `elementRef.style.position = "absolute";`

zIndex

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

Controls: Front-to-back layering of positioned elements. Multiple items with the same `zIndex` value are layered in source code order (earliest item at the bottom). The higher the value, the closer to the user's eye the element is.

Value: Integer number; Constant: `auto`

Example: `elementRef.style.zIndex = "3";`

Background properties

background

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

Controls: Up to five background style properties for an element.

Value: Combination values: `backgroundAttachment` || `backgroundColor` || `backgroundImage` || `backgroundPosition` || `backgroundRepeat`

Example: `elementRef.style.background = "scroll url(bricks.jpg) repeat-x";`

backgroundAttachment

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari1.2+, Opera+, Chrome+

Controls: Whether the background image remains fixed or scrolls with the content. Default is `scroll`.

Value: Constant: `fixed` | `scroll`

Example: `elementRef.style.backgroundAttachment = "fixed";`

backgroundColor

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

Controls: Solid opaque color for the background, or completely transparent. If you assign a background image, the color is layered behind the image so that any transparent spots of the image show the background color.

Value: Color value; Constant: `transparent`

Example: `elementRef.style.backgroundColor = "salmon";`

backgroundImage

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

Controls: The URL (if any) of an image to be used for the background for the element.

Value: URL value; Constant: `none`

Example: `elementRef.style.backgroundImage = "url(bricks.jpg)";`

backgroundPosition

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

Controls: The left-top location of the background image. Any offset from the left-top corner (default value `"0% 0%"`) allows background color to show through along the left and top edges of the element.

Value: Length values; Percentages; Constant: `left | center | right || top | center | bottom`. While single values are accepted, their behavior may not be as expected. Providing space-delimited pairs of values is more reliable

Example: `elementRef.style.backgroundPosition = "left top";`

backgroundPositionX
backgroundPositionY

Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari1.3+, Opera-, Chrome+

Controls: The left (`backgroundPositionX`) and top (`backgroundPositionY`) locations of the background image. Any offset from the left-top corner (default value "0%") allows background color to show through along the left and top edges of the element.

Value: Length value; Percentage; Constant: `left | center | right` (for `backgroundPositionX`); Constant: `top | center | bottom` (for `backgroundPositionY`)

Example: `elementRef.style.backgroundPositionX = "5px";`

backgroundRepeat

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

Controls: Image repetition characteristics of a background image. You can force the image to repeat along a single axis, if you want.

Value: Constant: `repeat` | `repeat-x` | `repeat-y` | `no-repeat`

Example: `elementRef.style.backgroundRepeat = "repeat-y";`

Border and edge properties

border

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

Controls: Up to three border characteristics (color, style, and width) for all four edges of an element.

Value: Combination values: `borderColor` || `borderStyle` || `borderWidth`

Example: `elementRef.style.border = "green groove 2px";`

borderBottom
borderLeft
borderRight
borderTop

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

Controls: Up to three border characteristics (color, style, and width) for a single edge of an element.

Value: Combination values: (for `borderBottom`) `borderBottomColor` || `borderBottomStyle` || `borderBottomWidth` (for `borderLeft`) `borderLeftColor` || `borderLeftStyle` || `border-LeftWidth` (for `borderRight`) `borderRightColor` || `borderRightStyle` || `border-RightWidth` (for `borderTop`) `borderTopColor` || `borderTopStyle` || `borderTopWidth`

Example: `elementRef.style.borderLeft = "#3300ff solid 2px";`

borderBottomColor
borderLeftColor
borderRightColor
borderTopColor

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

Controls: Color for a single border edge of an element.

Value: Color values; Constant: `transparent`

Example: `elementRef.style.borderTopColor = "rgb(30%, 50%, 0%)";`

borderBottomStyle
borderLeftStyle
borderRightStyle
borderTopStyle

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

Controls: Rendered style for a border edge of an element.

Value: Constant: `none` | `hidden` | `dotted` | `dashed` | `solid` | `double` | `groove` | `ridge` | `inset` | `outset`. WinIE versions prior to IE5.5 do not respond to the `dotted` or `dashed` types; MacIE does not respond to the `hidden` type

Example: `elementRef.style.borderRightStyle = "double";`

borderBottomWidth
borderLeftWidth
borderRightWidth
borderTopWidth

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

Controls: Thickness of a border edge of an element.

Value: Length value; Constant: `thin` | `medium` | `thick` (precise measure is at browser's discretion)

Example: `elementRef.style.borderBottomWidth = "5px";`

borderColor

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

Controls: Rendered color for one to four sides of an element.

Value: Color values for one to four rectangle sides

Example: `elementRef.style.borderColor = "green black";`

borderStyle

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

Controls: Rendered style for one to four sides of an element.

Value: One to four rectangle side constants: `none` | `hidden` | `dotted` | `dashed` | `solid` | `double` | `groove` | `ridge` | `inset` | `outset`. WinIE versions prior to IE5.5 do not respond to the `dotted` or `dashed` types; MacIE does not respond to the `hidden` type

Example: `elementRef.style.borderStyle = "ridge";`

borderWidth

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

Controls: Thickness of border for one to four sides of an element.

Value: One to four rectangle side length values or constants: `thin` | `medium` | `thick` (precise dimension is at browser's discretion)

Example: `elementRef.style.borderWidth = "5px 4px 5px 3px";`

margin

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

Controls: Thickness of transparent margin space outside the element's borders for one to four edges.

Value: One to four rectangle side length values

Example: `elementRef.style.margin = "10px 5px";`

marginBottom
marginLeft
marginRight
marginTop

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

Controls: Thickness of transparent margin space outside the element's borders for a single border edge.

Value: Length value

Example: `elementRef.style.marginBottom = "50px";`

mozBorderRadius
mozBorderRadiusBottomLeft
mozBorderRadiusBottomRight
mozBorderRadiusTopLeft
mozBorderRadiusTopRight

Compatibility: WinIE-, MacIE-, NN7+, Moz+, Safari-, Opera-, Chrome-

Controls: Radius of the border around the element. You can specify each radius corner as a series of values in the `mozBorderRadius` style (one value for all four corners; two values for top-left/bottom-right and top-right/bottom-left; three values for top-left, top-right/bottom-left, and bottom-right; four values for top-left, top-right, bottom-right, and bottom-left), or set each corner radius individually with its own property.

Value: Radius length value

Example: `elementRef.style.mozBorderRadius = "20px 10px 20px 10px";`

outline

Compatibility: WinIE-, MacIE5, NN-, Moz1.8.1+, Safari1.2+, Opera+, Chrome+

Controls: Up to three characteristics of an outline surrounding an element (similar to a border, but not shifting the location of internal content).

Value: Combination values: `outlineColor` || `outlineStyle` || `outlineWidth`

Example: `elementRef.style.outline = "red groove 2px";`

outlineColor

Compatibility: WinIE-, MacIE5, NN-, Moz1.8.1+, Safari1.2+, Opera+, Chrome+

Controls: Color of all four edges of an outline.

Value: Color values; Constant: `invert`

Example: `elementRef.style.outlineColor = "cornflowerblue";`

outlineOffset

Compatibility: WinIE-, MacIE5, NN-, Moz1.8.1+, Safari1.2+, Opera+, Chrome+

Controls: The space between an outline surrounding an element and the border of the element.

Value: Length value

Example: `elementRef.style.outlineOffset = "3px";`

outlineStyle

Compatibility: WinIE-, MacIE5, NN-, Moz1.8.1+, Safari1.2+, Opera+, Chrome+

Controls: Rendered style for all four sides of an element outline.

Value: Constant: `none` | `hidden` | `dotted` | `dashed` | `solid` | `double` | `groove` | `ridge` | `inset` | `outset`

Example: `elementRef.style.outlineStyle = "ridge";`

outlineWidth

Compatibility: WinIE-, MacIE5, NN-, Moz1.8.1+, Safari1.2+, Opera+, Chrome+

Controls: Thickness of all four sides of an element outline.

Value: Length value or constant: `thin` | `medium` | `thick` (precise dimension is at browser's discretion)

Example: `elementRef.style.outlineWidth = "4px";`

padding

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

Controls: Thickness of space between an element's content and its borders for one to four edges.

Value: One to four rectangle side length values.

Example: `elementRef.style.padding = "5px";`

paddingBottom
paddingLeft
paddingRight
paddingTop

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

Controls: Thickness of space between an element's content and its borders for a single edge.

Value: Length value

Example: `elementRef.style.paddingBottom = "20px";`

List properties

listStyle

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

Controls: Up to three characteristics of a list (`ol` or `ul`) presentation. Also applies to `dd`, `dt`, and `li` elements.

Value: Combination values: `listStyleImage` || `listStylePosition` || `listStyleType`

Example: `elementRef.style.listStyle = "none inside lower-alpha";`

listStyleImage

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

Controls: URL of the image to be used as a marker for a list item.

Value: URL value; Constant: `none`

Example: `elementRef.style.listStyleImage = "url(custombullet.jpg)";`

listStylePosition

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

Controls: Whether the marker should be formatted inside the wrapped text of its content or dangle outside the wrapped text (default).

Value: Constant: `inside` | `outside`

Example: `elementRef.style.listStylePosition = "inside";`

listStyleType

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

Controls: Which of the standard marker sets should be used for items in the list. A change to this property for a single `li` element causes succeeding items to be in the same style.

Value: For `ul` elements, constant: `circle` | `disc` | `square`

For `ol` elements, constant: `decimal` | `decimal-leading-zero` | `lower-alpha` | `lower-greek` | `lower-latin` | `lower-roman` | `upper-alpha` | `upper-greek` | `upper-latin` | `upper-roman`, and non-Roman formats when supported by the operating system (as in Mozilla for MacOS X): `armenian` | `georgian` | `hebrew` | `cjk-ideographic` | `hiragana` | `hiragana-iroha` | `katakana` | `katakana-iroha`

Example: `elementRef.style.listStyleType = "upper-roman";`

Scroll bar properties

scrollbar3dLightColor
scrollbarArrowColor
scrollbarBaseColor
scrollbarDarkShadowColor
scrollbarFaceColor
scrollbarHighlightColor
scrollbarShadowColor
scrollbarTrackColor

Compatibility: WinIE5.5, Mac-, NN-, Moz-, Safari-, Opera+, Chrome-

Controls: Colors of individual components of scroll bars when they are displayed for `applet`, `body`, `div`, `embed`, `object`, or `textarea` elements. To experiment with how different colors can affect the individual components, visit `http://msdn.microsoft.com/en-us/library/ms533055%28VS.85%29.aspx`.

Value: Color values; Constant: `none`

Example: `elementRef.style.scrollbarTrackColor = "hotpink";`

Table properties

borderCollapse

Compatibility: WinIE5+, MacIE5, NN6+, Moz+, Safari1.3+, Opera+, Chrome+

Controls: Whether a `table` element adheres to the CSS2 separated borders model or the collapsed borders model. Style is not fully supported in MacIE5.

Value: Constant: `collapse | separate`

Example: `elementRef.style.borderCollapse = "separate";`

borderSpacing

Compatibility: WinIE5+, MacIE5, NN6+, Moz+, Safari+, Opera+, Chrome+

Controls: For a table following the separated borders model, the thickness of the spacing between cell rectangles (akin to the `cellspacing` attribute of `table` elements). Style is not fully supported in MacIE5.

Value: One length value (for horizontal and vertical spacing) or comma-delimited list of two length values (the first for horizontal; the second for vertical)

Example: `elementRef.style.borderSpacing = "10px";`

captionSide

Compatibility: WinIE-, MacIE5, NN6+, Moz+, Safari+, Opera+, Chrome+

Controls: Position of the `caption` element inside a `table` element. Style is not implemented in MacIE5 and is only partially implemented in Safari.

Value: Constant: `top` | `right` | `bottom` | `left`

Example: `elementRef.style.captionSide = "bottom";`

emptyCells

Compatibility: WinIE-, MacIE5, NN6+, Moz+, Safari1.3+, Opera+, Chrome+

Controls: Rendering of cells and their borders when the cells have no content. Default behavior is to not render borders around empty cells. Style is not implemented in MacIE5.

Value: Constant: `show` | `hide`

Example: `elementRef.style.emptyCells = "show";`

tableLayout

Compatibility: WinIE5+, MacIE5, NN6+, Moz+, Safari+, Opera+, Chrome+

Controls: Whether table is rendered progressively based on fixed width settings of the first row of cells, or is rendered after the widths of all row content can be determined. Modifying this property after a table loads has no effect on the table.

Value: Constant `auto` | `fixed`

Example: `elementRef.style.tableLayout = "auto";`

Page and printing properties

orphans
widows

Compatibility: WinIE-, MacIE5, NN6+, Moz+, Safari-, Opera+, Chrome-

Controls: The minimum number of lines of a paragraph to be displayed at the bottom of a page (orphans) or top of a page (widows) when a page break occurs.

Value: Integer

Example: `elementRef.style.orphans = "4";`

page

Compatibility: WinIE-, MacIE5, NN6+, Moz+, Safari-, Opera-, Chrome-

Controls: The page (defined in an `@page` rule) with which the current element should be associated for printing.

Value: Identifier assigned to an existing `@page` rule

Example: `elementRef.style.page = "landscape";`

pageBreakAfter
pageBreakBefore

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari1.3+, Opera+, Chrome+

Controls: Whether a printed page break should be before or after the current element and the page break type.

Value: Constant: `auto | always | avoid | left | right`

Example: `elementRef.style.pageBreakBefore = "always";`

pageBreakInside

Compatibility: WinIE8+, MacIE5, NN6+, Moz+, Safari+, Opera+, Chrome+

Controls: Whether a printed page break is allowed inside an element.

Value: Constant: `auto | avoid`

Example: `elementRef.style.pageBreakInside = "avoid";`

size

Compatibility: WinIE-, MacIE-, NN6+, Moz+, Safari-, Opera+, Chrome-

Controls: The size or orientation of the page box (linked to the style rule via the page property) used to determine printed pages.

Value: One (same value for width and height) or two (width and height) space-delimited length values; constant: `auto | portrait | landscape`

Example: `elementRef.style.size = "portrait";`

Miscellaneous properties

accelerator

Compatibility: WinIE5+, MacIE-, NN-, Moz-, Safari-, Opera-, Chrome-

Controls: Whether an accelerator key is defined for an element.

Value: Boolean

Example: `elementRef.style.accelerator = "true";`

behavior

Compatibility: WinIE5+, MacIE-, NN-, Moz-, Safari-, Opera-, Chrome-

Controls: The external behavior to be applied to the current element.

Value: Space-delimited list of URL values. URLs can be a file location, an `object` element ID, or one of the built-in (default) behaviors.

Example: `elementRef.style.behavior = "url(#default#anchorClick)";`

cssText

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari1.3+, Opera+, Chrome+

Controls: Actual CSS rule text (read-only). This property exists by virtue of the browser's object model and is not part of the CSS specification. There is no corresponding CSS attribute.

Value: String

Example: `var cssRuleText = elementRef.style.cssText;`

imeMode

Compatibility: WinIE5+, MacIE-, NN-, Moz-, FF+, Safari-, Opera-, Chrome-

Controls: Whether text is entered into a text `input` or `textarea` element through the Input Method Editor (for languages, such as Chinese, Japanese, or Korean).

Value: Constant: `auto` | `active` | `inactive` | `disabled`

Example: `elementRef.style.imeMode = "active";`

Aural properties

Although these properties are defined in the CSS2 specification and placeholders exist for them in Mozilla-based browsers, the styles are not implemented in any browser other than Opera, and that support is experimental. The script equivalent properties are listed here for the sake of completeness only.

azimuth
cue
cueAfter
cueBefore
elevation
pause
pauseAfter
pauseBefore
pitch
pitchRange
playDuring
richness
speak
speakHeader
speakNumeral
speakPunctuation
speechRate
stress
voiceFamily
volume

Compatibility: WinIE-, MacIE-, NN6+, Moz+, Safari-, Opera+, Chrome-

Controls: A variety of styles primarily for browsers that support speech synthesis output.

Value: Consult `http://www.w3.org/TR/CSS2/aural.html` for details on aural stylesheets

filter Object

Properties	Methods	Event Handlers
See text		

Syntax

Accessing `filter` object properties and methods:

```
(IE4+)    document.all.objectID.filters[i].property | method([parameters])
(IE5.5+)  document.all.objectID.filters[filterName].property |
          method([parameters])
```

Compatibility: WinIE4+, MacIE-, NN-, Moz-, Safari-, Opera-, Chrome-

About this object

Earlier in this chapter, the `style.filter` property was shown to allow reading and writing of the string value that is assigned to an element's `style.filter` property. Filters are available in WinIE only, even though MacIE5 returns the `style.filter` property value. The purpose of this section is not to teach you how to use filters, but rather, how to script them.

Multiple filters are merely part of the space-delimited list of filters. Some filter types have additional specifications. For example, the `glow()` filter has three properties that more clearly define how the element should be rendered with a glow effect. The style sheet rule for an element whose ID is `glower` looks like the following:

```
#glower {filter:glow(color=yellow, strength=5, enabled=true)}
```

Accessing the `currentStyle.filter` property for that element yields the string value:

```
glow(color=yellow, strength=5, enabled=true)
```

Attempting to modify a single subproperty of the `glow()` filter by way of string parsing would be cumbersome and hazardous at best. For example, imagine trying to increment the glow filter's `strength` property by 5.

Reading and writing subproperties

A cleaner way to work with individual properties of a filter is to access the filter as an object belonging to the element affected by the filter. Each type of filter object has as its properties the individual sub-properties that you set in the style sheet. Continuing with the `glow()` filter example, you could access just the `color` property of the filter as follows:

```
var currColor = document.all.glower.filters["glow"].color;
```

To modify the color, assign a new value to the `filter` object's property:

```
document.all.glower.filters["glow"].color = "green";
```

To increment a numeric value, such as increasing the `glow()` filter's `strength` property by 5, use a construction such as the following (long-winded though it may be):

```
document.all.glower.filters["glow"].strength =
document.all.glower.filters["glow"].strength + 5;
```

Table 38-1 lists the filter object names that work all the way back to IE4, and the properties associated with each filter type.

In addition to the static filter types, which are applied to content and sit there unless modified by script, the IE4+ `filter` object also provides types for blends and reveals, for transitions between visible and invisible elements. Scripting transitions that occur when a script hides or shows an element requires a few lines of code, including calls to some of the `filter` object's methods. First, Table 38-2 shows the IE4+ syntax for transition filters.

TABLE 38-1

IE4-Compatible Static Filter Types

Filter Name	Description and Properties		
`alpha()`	Transparency level		
	Properties:	`opacity`	(0 to 100)
		`finishopacity`	(0 to 100)
		`style`	(gradient shape 0 to 3)
		`startX`	(coordinate integer)
		`startY`	(coordinate integer)
		`finishX`	(coordinate integer)
		`finishY`	(coordinate integer)
`blur()`	Simulate blurred motion		
	Properties:	`add`	(1 or 0)
		`direction`	(0, 45, 90, 135, 180, 225, 270, 315)
		`strength`	(pixel count)
`chroma()`	Color transparency		
	Properties:	`color`	(color value)
`dropShadow()`	Shadow effect		
	Properties:	`color`	(color value)
		`offx`	(horizontal offset in pixels)

continued

TABLE 38-1 *(continued)*

Filter Name	Description and Properties		
		`offy`	(vertical offset in pixels)
		`positive`	(1 or 0)
`flipH()`	Horizontally mirrored image		
	Properties:	None	
`flipV()`	Vertically mirrored image		
	Properties:	None	
`glow()`	Outer edge radiance		
	Properties:	`color`	(color value)
		`strength`	(intensity 1 to 255)
`gray()`	Eliminate color		
	Properties:	None	
`invert()`	Opposite hue, saturation, brightness levels		
	Properties:	None	
`light()`	Add light source (controlled by methods)		
	Properties:	None	
`mask()`	Overlay transparent mask		
	Properties:	`color`	(color value)
`shadow()`	Render as silhouette		
	Properties:	`color`	(color value)
		`direction`	(0, 45, 90, 135, 180, 225, 270, 315)
`wave()`	Add sine-wave distortion		
	Properties:	`add`	(1 or 0)
		`freq`	(integer number of waves)
		`light`	(strength 0 to 100)
		`phase`	(percentage offset 0 to 100)
		`strength`	(intensity 0 to 255)
`xRay()`	Render edges only		
	Properties:	None	

TABLE 38-2

IE4+ Transition Filters

Filter Name	Description and Properties			
`blendTrans()`	Fade out old element, fade in new element			
	Properties:	`duration`	(floating-point number of seconds)	
	Methods:	`apply()`	(freezes current display)	
		`play()`	(plays transition)	
		`stop()`	(stops transition mid-stream)	
`revealTrans()`	Reveal element to be shown through an effect			
	Properties:	`duration`	(floating-point number of seconds)	
		`transition`	(code number for effect)	
			`0`	Box in
			`1`	Box out
			`2`	Circle in
			`3`	Circle out
			`4`	Wipe up
			`5`	Wipe down
			`6`	Wipe right
			`7`	Wipe left
			`8`	Vertical blinds
			`9`	Horizontal blinds
			`10`	Checkerboard across
			`11`	Checkerboard down
			`12`	Random dissolve
			`13`	Split vertical in

continued

TABLE 38-2 *(continued)*

Filter Name	Description and Properties			
			14	Split vertical out
			15	Split horizontal in
			16	Split horizontal out
			17	Strips left down
			18	Strips left up
			19	Strips right down
			20	Strips right up
			21	Random bars horizontally
			22	Random bars vertically
			23	Random effect
	Methods:	`apply()`	(freezes current display)	
		`play()`	(plays transition)	
		`stop()`	(stops transition mid-stream)	

To make a transition work under script control, a filter must be applied to the element that you want the transition to work on. That can be done by script or by assigning a filter style to the element. As for the scripting, you begin by invoking the `apply()` method of the desired `filter` object. Next, script the change, such as assigning a new URL to the `src` property of an `img` element. While you do this, the `apply()` method freezes the image until you invoke the `play()` method on the filter. Listing 38-1 effects a checkerboard transition between two images after you click the image.

LISTING 38-1

A Reveal Transition Between Images

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>IE Transition</title>
      <style type="text/css">
         img {filter:revealTrans(transition=10)}
```

```
      </style>
      <script type="text/javascript">
         function doReveal()
         {
            document.getElementById("myIMG").filters["revealTrans"].apply();
               if (document.getElementById("myIMG").src.indexOf("desk1") != -1)
               {
                  document.getElementById("myIMG").src = "desk3.gif";
               }
               else
               {
                  document.getElementById("myIMG").src = "desk1.gif";
               }
            document.getElementById("myIMG").filters["revealTrans"].play();
         }
      </script>
   </head>
   <body>
      <h1>IE Transition</h1>
      <hr />
      <p>Click on the image to cause a reveal transition.</p>
      <img id="myIMG" alt="image" src="desk1.gif" height="90" width="120"
           onclick="doReveal()" />
   </body>
</html>
```

Note

The property assignment event handling technique used in this example, and the next, is a deliberate simplification to make the code more readable. It is generally better to use the more modern approach of binding events using the `addEventListener()` (NN6+/Moz/W3C) or `attachEvent()` (IE5+) methods. A modern cross-browser event handling technique is explained in detail in Chapter 32, "Event Objects." ■

Building on the example in Listing 38-1, the next example in Listing 38-2 demonstrates how a script can also modify a `filter` object's property, including a transition filter. Before the transition filter has its `apply()` method invoked, the script sets the transition type based on a user choice in a `select` list.

LISTING 38-2

Choosing Reveal Transitions Between Images

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>IE Transition and Choices</title>
      <style type="text/css">
         img {filter:revealTrans(transition=10)}
```

continued

LISTING 38-2 *(continued)*

```
   </style>
   <script type="text/javascript">
      function doReveal()
      {
         document.getElementById("myIMG").filters
           ["revealTrans"].transition = document.forms[0].transChoice.value;
         document.getElementById("myIMG").filters["revealTrans"].apply();
            if (document.getElementById("myIMG").src.indexOf("desk1") != -1)
            {
               document.getElementById("myIMG").src = "desk3.gif";
            }
            else
            {
               document.getElementById("myIMG").src = "desk1.gif";
            }
         document.getElementById("myIMG").filters["revealTrans"].play();
      }
   </script>
</head>
<body>
   <h1>IE Transition and Choices</h1>
   <hr />
   <form>
      <p>Choose the desired transition type:
         <select name="transChoice">
            <option value="0">Box in</option>
            <option value="1">Box out</option>
            <option value="2">Circle in</option>
            <option value="3">Circle out</option>
            <option value="4">Wipe up</option>
            <option value="5">Wipe down</option>
            <option value="6">Wipe right</option>
            <option value="7">Wipe left</option>
            <option value="8">Vertical blinds</option>
            <option value="9">Horizontal blinds</option>
            <option value="10">Checkerboard across</option>
            <option value="11">Checkerboard down</option>
            <option value="12">Random dissolve</option>
            <option value="13">Split vertical in</option>
            <option value="14">Split vertical out</option>
            <option value="15">Split horizontal in</option>
            <option value="16">Split horizontal out</option>
            <option value="17">Strips left down</option>
            <option value="18">Strips left up</option>
            <option value="19">Strips right down</option>
            <option value="20">Strips right up</option>
            <option value="21">Random bars horizontally</option>
            <option value="22">Random bars vertically</option>
            <option value="23">Random effect</option>
```

```
            </select>
         </p>
      </form>
      <p>Click on the image to cause a reveal transition.</p>
      <img alt="image" id="myIMG" src="desk1.gif" height="90" width="120"
           onclick="doReveal()" />
   </body>
</html>
```

WinIE5.5+ filter syntax changes

While WinIE5.5+ still supports the original IE4 way of controlling filters, the browser also implements a new filter component, which Microsoft strongly encourages authors to use (as evidenced by the difficulty in finding documentation for the IE4 syntax at its developer web site). In the process of implementing this new filter component, the names of many filters change, as do their individual properties. Moreover, the way the filter component is invoked in the style sheet is also quite different from the original component.

The style sheet syntax requires a reference to the new component as well as to the filter name. Here is the old way:

```
#glower {filter:glow(color=yellow, strength=5, enabled=true)}
```

And here is the new way:

```
#glower {filter:progid:DXImageTransform.Microsoft.Glow(color=yellow,
   strength=5, enabled=true)}
```

Don't overlook the extra `progid:` pointer in the reference. This program identifier becomes part of the name that your scripts use to reference the filter:

```
document.getElementById("glower").filters[
   "DXImageTransform.Microsoft.Glow"].color = "green";
```

While some of the filter names and properties stay the same (except for the long prefix), several older properties are subsumed by new filters whose properties help identify the specific effect. The former `revealTrans()` filter is now divided among several new filters dedicated to transition effects. Table 38-3 shows the IE5.5+ syntax.

Note

Using the filter syntax introduced in IE5.5+ can cause frequent crashes of the browser (at least in early released versions), especially when it's used for transition filters. If you implement the new syntax, be sure to torture-test your pages extensively. Ideally, you should encourage users of these pages to run IE6+. ■

TABLE 38-3

IE5.5 DXImageTransform.Microsoft Filter Names

Filter Name	Description and Properties		
Alpha()	Transparency level		
	Properties:	opacity	(0 to 100)
		finishopacity	(0 to 100)
		style	(gradient shape 0 to 3)
		startX	(coordinate integer)
		startY	(coordinate integer)
		finishX	(coordinate integer)
		finishY	(coordinate integer)
Barn()	Barn-door style transition		
	Properties:	duration	(floating-point number of seconds)
		motion	(in or out)
		orientation	(horizontal or vertical)
		percent	(0 to 100)
		status	0 (stopped), 1 (applied), 2 (playing)
	Methods:	apply()	(freezes current display)
		play()	(plays transition)
		stop()	(stops transition mid-stream)
BasicImage()	Element rotation, flip, color effects, and opacity		
	Properties:	grayScale	(1 or 0)
		invert	(1 or 0)
		mask	(1 or 0)
		maskColor	(color value)
		mirror	(1 or 0)
		opacity	(0.0 to 1.0)

Filter Name	Description and Properties		
		rotation	0 (no rotation), 1 (90°), 2 (180°), 3 (270°)
		xRay	(1 or 0)
Blinds()	Action transition with Venetian blind effect		
	Properties:	direction	(up, down, right, left)
		squaresX	(integer column count)
		squaresY	(integer row count)
		status	0 (stopped), 1 (applied), 2 (playing)
	Methods:	apply()	(freezes current display)
		play()	(plays transition)
		stop()	(stops transition mid-stream)
Checkerboard()	Action transition with checkerboard effect		
	Properties:	bands	(1 to 100)
		direction	(up, down, right, left)
		duration	(floating-point number of seconds)
		percent	(0 to 100)
		slideStyle	(HIDE, PUSH, SWAP)
		status	0 (stopped), 1 (applied), 2 (playing)
	Methods:	apply()	(freezes current display)
		play()	(plays transition)
		stop()	(stops transition mid-stream)
Chroma()	Color transparency		
	Properties:	color	(color value)
DropShadow()	Shadow effect		
	Properties:	color	(color value)

continued

TABLE 38-3 *(continued)*

Filter Name	Description and Properties		
		offx	(horizontal offset pixels)
		offy	(vertical offset pixels)
		positive	(1 or 0)
Fade()	Blend transition		
	Properties:	duration	(floating-point number of seconds)
		overlap	(0.0 to 1.0 seconds)
		percent	(0 to 100)
		status	0 (stopped), 1 (applied), 2 (playing)
	Methods:	apply()	(freezes current display)
		play()	(plays transition)
		stop()	(stops transition mid-stream)
Glow()	Outer edge radiance		
	Properties:	color	(color value)
		strength	(intensity 1 to 255)
Iris()	Action transition with zoom effect		
	Properties:	duration	(floating-point number of seconds)
		irisStyle	(CIRCLE, CROSS, DIAMOND, PLUS, SQUARE, STAR)
		motion	(in or out)
		percent	(0 to 100)
		status	0 (stopped), 1 (applied), 2 (playing)
	Methods:	apply()	(freezes current display)
		play()	(plays transition)
		stop()	(stops transition mid-stream)
Light()	Add light source (controlled by methods)		
	Properties:	None	

Filter Name	Description and Properties		
	Methods:	addAmbient (red, green, blue, strength)	
		addCone (sourceLeft, sourceTop, sourceZAxis, targetLeft, targetTop, red, green, blue, strength, spreadAngle)	
		addPoint (sourceLeft, sourceTop, sourceZAxis, red, green, blue, strength)	
		changeColor (lightID, red, green, blue, absoluteColorFlag)	
		changeStrength (lightID, strength, absoluteIntensityFlag)	
		clear()	
		moveLight (lightID, sourceLeft, sourceTop, sourceZAxis, absoluteMovementFlag)	
MaskFilter()	Overlay transparent mask		
	Properties:	color	(color value)
MotionBlur()	Simulate blurred motion		
	Properties:	add	(1 or 0)
		direction	(0, 45, 90, 135, 180, 225, 270, 315)
		strength	(pixel count)
RandomDissolve()	Pixelated dissolve transition		
	Properties:	duration	(floating-point number of seconds)

continued

TABLE 38-3 *(continued)*

Filter Name	Description and Properties		
		percent	(0 to 100)
		status	0 (stopped), 1 (applied), 2 (playing)
	Methods:	apply()	(freezes current display)
		play()	(plays transition)
		stop()	(stops transition mid-stream)
RandomBars()	Bar style transition		
	Properties:	duration	(floating-point number of seconds)
		orientation	(horizontal or vertical)
		percent	(0 to 100)
		status	0 (stopped), 1 (applied), 2 (playing)
	Methods:	apply()	(freezes current display)
		play()	(plays transition)
		stop()	(stops transition mid-stream)
Shadow()	Render as silhouette		
	Properties:	color	(color value)
		direction	(0, 45, 90, 135, 180, 225, 270, 315)
Stripes()	Striped style transition		
	Properties:	duration	(floating-point number of seconds)
		motion	(in or out)
		percent	(0 to 100)
		status	0 (stopped), 1 (applied), 2 (playing)
	Methods:	apply()	(freezes current display)
		play()	(plays transition)
		stop()	(stops transition mid-stream)

Filter Name	Description and Properties		
`Wave()`	Add sine-wave distortion		
	Properties:	`add`	(1 or 0)
		`freq`	(integer number of waves)
		`light`	(strength 0 to 100)
		`phase`	(percentage offset 0 to 100)
		`strength`	(intensity 0 to 255)
`xRay()`	Render edges only		
	Properties:	None	

For more details on deploying filters in IE for Windows, visit `http://msdn.microsoft.com/library/default.asp?url=/workshop/author/filter/filters.asp`. Because most of the live examples at that page require WinIE5.5+, be sure to use that version for the best experience.

CHAPTER 39

Ajax, E4X, and XML

IN THIS CHAPTER

Treating XML elements as objects

Creating XML data islands

Accessing XML element attributes

Using the XMLHttpRequest object

Using E4X to create dynamic XML objects

XML (eXtensible Markup Language) is an undeniably hot topic in the Internet world, and has been for the past few years. Not only has the W3C organization formed multiple working groups and recommendations for XML and its offshoots, but the W3C DOM recommendation also has XML in mind when it comes to defining how elements, attributes, and data of any kind — not just the HTML vocabulary — are exposed to browsers as an object model. Most of the arcana of the W3C DOM Core specification — especially the structure based on the node — are in direct response to the XML possibilities of documents that are beginning to travel the Internet.

ECMAScript for XML (E4X) is a standardized programming language extension that provides native XML support to ECMAScript as a primitive type, at the same level as strings, numbers, and Booleans. It has been supported since FireFox 1.5, but is not supported as of Safari 4.0.4, Google Chrome 4.1.249.1045, Opera 10.51, or IE 8.

During its early explorations into XML and browsers, Microsoft devised a custom HTML element — the `<xml>` tag — that allowed authors to embed XML data into an HTML document. These tags created what were called XML *data islands*. A more practical solution came slightly later with the creation of an ActiveX control that could retrieve XML data (from either a static `.xml` file or a web service that returns XML-structured data) into a web page without disturbing the HTML portion. Scripts could then use W3C DOM methods and properties to read the node tree as needed. Mozilla, Opera, and Safari browsers emulate the behavior of this `XMLHttpRequest` control in a native object so that modern web applications can load external XML data into a page for script inspection and manipulation. In an unusual turn of events, Microsoft has also now implemented the native `XMLHttpRequest` object in IE7 to match the implementation of other browsers.

The functionality made possible by the `XMLHttpRequest` object encapsulates the much-hyped buzzword Ajax, which stands for Asynchronous JavaScript And XML. This chapter covers both WinIE XML data islands and the client-side aspects of Ajax (the `XMLHttpRequest` object). Out of necessity, this book assumes that you are already familiar with XML such that your server-based applications serve up XML data exclusively, embed XML islands into HTML documents, or convert database data into XML. The focus of this chapter, and the application examples in Chapter 55, "Application: Outline-Style Table of Contents," and Chapter 60, "Application: Transforming XML Data" (on the CD-ROM), is how to access XML data and apply that data to rendered HTML content.

Elements and Nodes

When you leave the specialized DOM vocabulary of HTML elements, the world can appear rather primitive — a highly granular world of node hierarchies, elements, element attributes, and node data. This granularity is a necessity in an environment in which the elements are far from generic, and the structure of data in a document does not have to follow a format handed down from above. One web application can describe an individual's contact information with one set of elements, whereas another application uses a completely different approach to element names, element nesting, and their sequence.

Fortunately, most, if not all, scripting you do on XML data is on data served up by your own applications. Therefore, you know what the structure of the data is — or you know enough of it to let your scripts access the data.

The discussion of the W3C DOM in Chapter 25, "Document Object Model Essentials," should serve as a good introduction to the way you need to think about elements and their content. All relevant properties and methods are listed among the items shared by all elements in Chapter 26, "Generic HTML Element Objects."

XML data, whether delivered raw or embedded in a WinIE HTML document as a *data island* is a hierarchy of nodes. Typically, the outermost nodes are elements. Some elements have attributes, each of which is a typical name/value pair. Some elements have data that goes between the start and end tags of the element (such data is a text node nested inside the element node). And some elements can have both attributes and data. When an XML data collection contains the equivalent of multiple database records, an element container whose tag name is the same as each of the other records surrounds each record. Thus, the `getElementsByTagName()` method frequently accesses a collection of like-named elements.

When you have a reference to an element node, you can reference that element's attributes as properties; however, a more formal access route is through the `getAttribute()` method of the element. If the element has text data between its start and end tags, you can access that data from the element's reference by calling the `firstChild.nodeValue` property (although you may want to verify that the element has a child node of the text type before committing to retrieving the data).

Of course, your specific approach to `xml` elements and their data varies with what you intend to script with the data. For example, you may wish to do nothing more with scripting than enable

a different style sheet for the data based on a user choice. The XSL (eXtensible Stylesheet Language) standard is a kind of (non-JavaScript) scripting language for transforming raw `xml` data into a variety of presentations. But you can still use JavaScript to connect user-interface elements that control which of several style sheets renders the data. Or, as demonstrated in Chapters 55 and 60, you may want to use JavaScript for more explicit control over the data and its rendering, taking advantage of JavaScript sorting and data manipulation facilities along the way.

Table 39-1 summarizes the W3C DOM Core objects, properties, and methods that you are most likely to use in extracting data from `xml` elements. You can find details about all these items in Chapter 26.

TABLE 39-1

Properties and Methods for XML Element Reading

Property or Method	Description
`Node.nodeValue`	Data of a text node
`Node.nodeType`	Which node type
`Node.parentNode`	Reference to parent node
`Node.childNodes`	Array of child nodes
`Node.firstChild`	First of all child nodes
`Node.lastChild`	Last of all child nodes
`Node.previousSibling`	Previous node at same level
`Node.nextSibling`	Next node at same level
`Element.parentNode`	Reference to parent node
`Element.childNodes`	Array of child nodes
`Element.firstChild`	First of all child nodes
`Element.lastChild`	Last of all child nodes
`Element.previousSibling`	Previous node at same level
`Element.nextSibling`	Next node at same level
`Element.tagName`	Tag name
`Element.getAttribute(name)`	Retrieves attribute (`Attr`) object
`Element.getElementsByTagName(name)`	Array of nested, named elements
`Attr.name`	Name part of attribute object's name/value pair
`Attr.value`	Value part of attribute object's name/value pair

xml Element Object

For HTML element properties, methods, and event handlers, see Chapter 26, "Generic HTML Element Objects."

Properties	Methods	Event Handlers
`src`		
`XMLDocument`		

Syntax

Accessing `xml` element object properties or methods:

```
(IE5+)    [window.]document.all.elementID.property | method([parameters])
```

Compatibility: WinIE5+, MacIE-, NN-, Moz-, Safari-, Opera-, Chrome-

About this object

The `xml` element object is the primary container of an `xml` data island within an HTML page. If your scripts intend to traverse the node hierarchy within the element, or simply access properties of nested elements, you should assign an identifier to the `id` attribute of the XML element. For example, if the XML data contains results from a database query for music recordings that match some user-entered criteria, each returned record might be denoted as a `recording` element as follows:

```
<xml id="results">
   <searchresults>
      <recording>
         ...elements with details...
      </recording>
      <recording>
         ...elements with details...
      </recording>
      <recording>
         ...elements with details...
      </recording>
   </searchresults>
</xml>
```

Your script can now obtain an array of references to `recording` elements as follows:

```
var recs =
   document.getElementById("results").getElementsByTagName("recording");
```

Although it is also true that there is no known HTML element with the tag name `recording` (which enables you to use `document.getElementsByTagName("recording")`), the

unpredictability of `xml` data element names is reason enough to limit the scope of the `getElementsByTagName()` method to the `xml` data island.

The W3C DOM Level 2 does not define an `xml` element object within the HTML section. However, you can embed an XML document inside an HTML document in Mozilla even though the standards clearly indicate that a document can be one or the other, but not both. Of course, the browser understandably gets confused when custom elements have tag names that already belong to the HTML DTD. Therefore, we do not recommend attempting to embed custom elements into an HTML document for NN6+/Moz unless you are very careful to use entirely unique tag names that don't clash in any way with HTML or you know how to use XML namespaces within an XHTML document.

Properties

src

Value: String Read/Write

Compatibility: WinIE5+, MacIE-, NN-, Moz-, Safari-, Opera-, Chrome-

The `src` property represents the `src` attribute of the `xml` element. The attribute points to the URL of an external `xml` document whose data is embedded within the current HTML document.

XMLDocument

Value: Object reference Read-Only

Compatibility: WinIE5+, MacIE-, NN-, Moz-, Safari-, Opera-, Chrome-

The `XMLDocument` property returns a reference to Microsoft's proprietary XML `document` object and the object model associated with it (the so-called XML DOM). A lot of this object model is patterned after the W3C DOM model, but access to these properties is through a rather roundabout way. For more details, visit `http://msdn.microsoft.com/en-us/library/ms535918.aspx`.

XMLHttpRequest Object

Properties	Methods	Event Handlers
readyState	abort()	onreadystatechange†
responseText	getAllResponseHeaders()	
responseXML	getResponseHeader()	
status	open()	
statusText	send()	
	setRequestHeader()	

†See Chapter 26, "Generic HTML Element Objects."

Syntax

Accessing `XMLHttpRequest` object properties or methods:

```
(IE5+/Moz)     XMLHttpRequestObjectRef.property | method([parameters])
```

Compatibility: WinIE5+, MacIE-, NN7+, Moz+, Safari1.2+, Opera+, Chrome+

About this object

The `XMLHttpRequest` object is an abstract object that lets your scripts retrieve XML data from, or send XML data to, any URL designed for that purpose. All the action occurs invisibly to the user, and it is the responsibility of your scripts to make the connection with the server and process the XML data, either after retrieval or prior to submission. This object was originally designed by Microsoft as part of its XML Core Services (MSXML), as first released as part of Internet Explorer 5 for Windows. Mozilla engineers implemented much of the same functionality in Mozilla browsers, with almost identical syntax. Similar functionality entered the Safari equation in Safari 1.2, not to mention the Opera browser in Opera 8. These latter implementations adhere very closely to the Mozilla `XMLHttpRequest` implementation.

Where the IE and Mozilla variations differ is how you create the object to begin with. Because the IE version is an ActiveX control, you create the object using the `ActiveXObject` constructor function. At least that's the case with versions of IE prior to version 7. In IE7, Microsoft finally got around to supporting the Mozilla object creation approach, which involves using a constructor for the `XMLHttpRequest` object. If you plan on using the `XMLHttpRequest` object in versions of IE prior to version 7, which is likely, you must equalize the creation of the two object versions in a single document and branch your code accordingly. Use object detection to handle the branching most effectively:

```
var req = null;
// branch for native XMLHttpRequest object
if (window.XMLHttpRequest)
   {
      try
      {
         req = new XMLHttpRequest();
      }
      catch(e)
      {
         req = null;
      }
   // branch for IE/Windows ActiveX version
   } else if (window.ActiveXObject)
   {
      try
      {
         req = new ActiveXObject("Msxml2.XMLHTTP");
      }
      catch(e)
```

```
        {
           try
           {
              req = new ActiveXObject("Microsoft.XMLHTTP");
           }
           catch(e)
           {
              req = null;
           }
        }
     }
```

Notice in the code how there are actually two different ActiveX objects that support the `XMLHttpRequest` functionality in IE. The first ActiveX object, `Microsoft.XMLHTTP`, represents the first incarnation of `XMLHttpRequest` as found in IE5. IE5.5 supplanted this ActiveX object with a newer one called `Msxml2.XMLHTTP`, which continued to be the IE-preferred means of accessing `XMLHttpRequest` through IE6. IE7 added support for the Mozilla-style approach of instantiating an actual `XMLHttpRequest` object without ActiveX. The example code, therefore, demonstrates how to gracefully create an `XMLHttpRequest` object while taking into consideration the various browser inconsistencies dating back to IE5/NN6/Moz1/Safari1.2/Opera8.

After the object is created, the basic syntax for opening a connection, sending the request, and retrieving the response data is the same for both WinIE and other browsers. To retrieve an XML document (node tree) from a URL source, the basic conceptual sequence is as follows:

1. Open the request object, specifying the request type and URL.
2. Bind an event handler function to the request object; this function is called when the request finishes.
3. Send the request.
4. Process the results of the request.

Let's take a look at each of these steps and the JavaScript code involved. Following is the code required to open the request object:

```
req.open("GET", "sourceURL", true);
```

This line of code opens the request object by passing along the `GET` request type, the URL of the data source, and whether the request is synchronous or asynchronous. The last argument is undoubtedly the most important because it directly controls whether the request is allowed to place in the background (asynchronously) or if the script should wait on the request (synchronously). Seeing as how the word *asynchronous* is in the acronym Ajax, it stands to reason that all Ajax requests pass `true` as the third parameter to the `open()` method. The asynchronous nature of Ajax is what gives Ajax applications such a unique feel, in that work can be carried out on the server and dynamically reflected on the client as it finishes.

The event handler binding in Step 2 of the previous list involves setting a function reference to the `onreadystatechange` property:

```
req.onreadystatechange = processRequest;
```

The function you assign here is called when the status of the request changes. You will typically only be concerned with the status changing to "complete."

An intermediate step that isn't strictly required involves setting the content type of the request header. The `XMLHttpRequest` object isn't limited to opening XML documents. Because of this, you may want to explicitly set the header type to `text/xml`, just to make sure there is no confusion when you are opening XML data; some browsers act very strict with respect to the content type of the header. Following is the code that sets the header's content type:

```
req.setRequestHeader("Content-Type", "text/xml");
```

Step 3, sending the request, is perhaps the simplest step in performing an Ajax request:

```
req.send("");
```

At this point, the request has been issued, and you can begin to check and see if it has completed. Control has returned to the browser thanks to the asynchronous nature of the request. The job of checking the status of the request and processing any results falls to the `processRequest()` handler function that was set a moment ago.

The request handler function is automatically called when a change occurs in the state of the request. It is possible for the request to cycle through any of the following states:

- Uninitialized (0)
- Loading (1)
- Loaded (2)
- Interactive (3)
- Complete (4)

The number beside each of the states corresponds to possible values for the `readyState` property of the request object. This is the property you use to find out if the request has finished and is ready for processing. There is one other property, however, that is important to look at before charging into the XML processing. We're referring to the `status` property, which really has only one value of concern to you, `200` — which means the request was successful.

Pulling this information together enables you to assemble a skeletal request event handling function:

```
function processRequest(req) {
   if (req.readyState == 4 && req.status == 200)
   {
      var xmlDoc = req.responseXML;
      // further processing of document here
   }
}
```

At this point, scripts can inspect the contents of the `xmlDoc` value by way of W3C DOM node properties and methods.

Note

The `XMLHttpRequest` object in some browsers must reference pages served from a web server, and not a local file. You can experiment successfully from a personal web server running on your PC, but not with files accessed through the `file:` protocol. If the xml file is accessed through the `file:` protocol, the `status` property will return a `0`. ■

Listing 39-1 shows a utility script that retrieves XML content from a URL (passed as a parameter to the `loadXML()` function) in a cross-browser manner. Additional error checking verifies that the retrieval is successful before moving forward. Notice that this code is more of an Ajax template than a functioning example. You have to plug in your own code inside the `processRequest()` function once the `xmlDoc` variable is set.

LISTING 39-1

Utility XML Data Reading Script

```
var req = null;

// retrieve XML document as document object
function loadXMLDoc(url)
{
   // branch for native XMLHttpRequest object
   if (window.XMLHttpRequest)
   {
      try
      {
         req = new XMLHttpRequest();
      }
      catch(e)
      {
         req = null;
      }
   // branch for IE/Windows ActiveX version
   } else if (window.ActiveXObject)
   {
      try
      {
         req = new ActiveXObject("Msxml2.XMLHTTP");
      }
      catch(e)
      {
         try
         {
            req = new ActiveXObject("Microsoft.XMLHTTP");
         }
         catch(e)
         {
            req = null;
         }
      }
   }
```

```
    if (req)
    {
       req.open("GET", url, true);
       req.onreadystatechange = processRequest;
       req.setRequestHeader("Content-Type", "text/xml");
       req.send("");
    }
}

function processRequest()
{
    if (req.readyState == 4 && req.status == 200)
    {
       var xmlDoc = req.responseXML;
          if (xmlDoc)
          {
             // get busy processing XML
          }
    }
}
```

Properties and methods described in this chapter are those that the object has in common for both WinIE and Mozilla browsers, as well as WebKit-based browsers and Opera. You can see examples of this object, and the template in Listing 39-1, within the applications of Chapters 55 and 60.

Properties

readyState

Value: Integer Read-Only

Compatibility: WinIE5+, MacIE-, NN7+, Moz+, Safari1.2+, Opera+, Chrome+

Your scripts can read the value of the `readyState` property to determine the state of the `XMLHttpRequest` object, particularly while it is operating during its initialization or data transfer. Values are the same as for other objects that offer this property. See the bulleted list earlier in this chapter, as well as Table 26-6, for integer values and their meanings. When carrying out asynchronous (Ajax) requests, you assign an `onreadystatechange` event handler to the `XMLHttpRequest` object; the event function then inspects the `readyState` property for further processing.

Related Item: `status` property

responseText

Value: String Read-Only

Compatibility: WinIE5+, MacIE-, NN7+, Moz+, Safari1.2+, Opera+, Chrome+

After the `send()` method executes, and if the server returns any data (as it will with a `GET` operation), you can access a string version of the returned data through the `responseText` property. If the returned data is an XML document, this property provides a string-only version of the entire content.

Related Item: `responseXML` property

responseXML

Value: XML document object — Read-Only

Compatibility: WinIE5+, MacIE-, NN7+, Moz+, Safari1.2+, Opera+, Chrome+

After the `send()` method executes, and if the server returns any data (as it will with a `GET` operation), you can access the returned W3C DOM–compliant document object through the `responseXML` property. The object to which this property points is a genuine document node (with a `nodeType` of 9), which gives your scripts the power to walk the node tree, and retrieve tags, attributes, and text nodes inside elements, as you would with any DOM document.

As the examples in Chapters 55 and 60 demonstrate, you can use the data from the XML document to build HTML that displays the XML content in the format of your choice (using JavaScript as a more flexible alternative to XSL). If your page is interactive to the extent that users can modify the content, you may then modify the document tree stored in your script variable, and send the revised XML back to the server by opening a new `XMLHttpRequest` connection pointing to the URL that accepts the posted data.

Related Items: `responseText` property; `open()` method

status

Value: Integer — Read-Only

Compatibility: WinIE5+, MacIE-, NN7+, Moz+, Safari1.2+, Opera+, Chrome+

After the `send()` method executes, you can read the status of the transaction through the `status` property. The value is an integer corresponding to the response issued by the server at the end of the transaction. A successful transaction value is 200 (corresponding to the OK `statusText` property value). Perhaps the other most common status value is 404, which occurs if the URL you supply to the `open()` method points to a file or source not found on the server. As shown in Listing 39-1, you can use the 200 value as the key to determining if the transaction is a success. You might consider reporting any other value to the user (although inexperienced users may not understand the meaning of the status text).

A complete list of status values and related descriptions (status text) is shown in Table 39-2. Keep in mind that the vast majority of the time you will be concerned only with whether or not the status code is 200 (OK).

Related Item: `statusText` property

TABLE 39-2

HTTP Status Codes for the status Property

Status Code	Status Text
100	Continue
101	Switching Protocols
200	OK
201	Created
202	Accepted
203	Non-Authoritative Information
204	No Content
205	Reset Content
206	Partial Content
300	Multiple Choices
301	Moved Permanently
302	Found
303	See Other
304	Not Modified
305	Use Proxy
307	Temporary Redirect
400	Bad Request
401	Unauthorized
402	Payment Required
403	Forbidden
404	Not Found
405	Method Not Allowed
406	Not Acceptable
407	Proxy Authentication Required
408	Request Timeout
409	Conflict
410	Gone
411	Length Required

Status Code	Status Text
412	Precondition Failed
413	Request Entity Too Large
414	Request-URI Too Long
415	Unsupported Media Type
416	Requested Range Not Suitable
417	Expectation Failed
500	Internal Server Error
501	Not Implemented
502	Bad Gateway
503	Service Unavailable
504	Gateway Timeout
505	HTTP Version Not Supported

`statusText`

Value: String Read-Only

Compatibility: WinIE5+, MacIE-, NN7+, Moz+, Safari1.2+, Opera+, Chrome+

After the `send()` method executes, you can read the plain-language status of the transaction through the `statusText` property. The value is a string corresponding to the response integer by the server at the end of the transaction. A successful transaction value is OK (corresponding to the 200 `status` property value). Use the `status` property for testing the results in your script, and the `statusText` property to report errors to users. Table 39-2 contains a list of the possible status text values that may be stored in the `statusText` property. For an example using the `status` property, see Listing 39-1.

Related Item: `status` property

Methods

`abort()`

Returns: Nothing

Compatibility: WinIE5+, MacIE-, NN7+, Moz+, Safari1.2+, Opera+, Chrome+

The `abort()` method stops any transaction currently in progress. This method is the scripted equivalent of clicking a browser's Stop button while it retrieves the contents of a web page.

Related Items: `readyState` property; `send()` method

getAllResponseHeaders()
getResponseHeader("*headerName*")

Returns: String

Compatibility: WinIE5+, MacIE-, NN7+, Moz+, Safari1.2+, Opera+, Chrome+

For each transaction, the server transmits a series of name/value pairs as a header to the actual data. The `getAllResponseHeaders()` method returns the complete set as received by the `XMLHttpRequest` object. Such a header set may look like the following:

```
Date: Mon, 12 Feb 2007 03:12:59 GMT
Server: Apache/1.3.27 (Darwin)
Last-Modified: Sun, 28 Jan 2007 22:13:04 GMT
Etag: "12babe-3a2-3f809770"
Accept-Ranges: bytes
Content-Length: 930
Keep-Alive: timeout=15, max=100
Connection: Keep-Alive
Content-Type: text/xml
```

If you want to retrieve the value of just one of the headers, use the `getResponseHeader()` method, and pass as a parameter a string with only the name portion of one of the headers. For example:

```
var size = req.getResponseHeader("Content-Length");
```

The parameter is not case-sensitive, but the spelling (along with any hyphen in the name) is critical.

Related Items: `readyState` property; `send()` method

open("*method*", "URL"[, *asyncFlag*[, "*userName*"[, "*password*"]]])

Returns: Nothing

Compatibility: WinIE5+, MacIE-, NN7+, Moz+, Safari1.2+, Opera+, Chrome+

Use the `open()` method to specify the transaction type and URL of the destination of the request. The *method* parameter may be either `GET` (for retrieving data from a server) or `POST` (for sending XML to a server). The *URL* may be either relative to the current page, or a complete `http:` URL.

Three additional parameters are optional. The first is a Boolean value for whether the request should be asynchronous. If `true` (the default), the `XMLHttpRequest` object does not wait for a response (after the `send()` method) before continuing with script processing. By setting this parameter to `false`, you ensure that processing continues only after the transaction has completed or timed out. Of course, this also ensures that the user can't do anything while waiting for the server to process your request, and may feel as though the browser has frozen. The preferred approach is to set the parameter to `true` and carry out all requests asynchronously. All

the `XMLHttpRequest` examples in this book (Chapters 55 and 60, primarily) utilize this latter asynchronous (Ajax) approach.

The other optional parameters are strings for a username and password, if authentication is needed to access the URL.

Note that the `open()` method merely fills various properties of the request, and that the request does not occur until the `send()` method is invoked.

Related Item: `send()` method

send(*content*)

Returns: Nothing

Compatibility: WinIE5+, MacIE-, NN7+, Moz+, Safari1.2+, Opera+, Chrome+

After setting the characteristics of the request through the `open()` method and its parameters, invoke the `send()` method to trigger the actual request over the network. For a GET operation, specify `""` or `null` as the parameter. But for a POST operation, the parameter should be a reference to a DOM document that has been assembled in script. You may also specify a string as the value being posted to the request's URL.

Related Item: `open()` method

setRequestHeader("*name*", "value")

Returns: Nothing

Compatibility: WinIE5+, MacIE-, NN7+, Moz+, Safari1.2+, Opera+, Chrome+

The `setRequestHeader()` method enables you to specify a name/value pair for the header being sent with the HTTP request. For this method to succeed, it must be called only when `readyState` is set to 1 (Loading); see Table 26-6 for more details. In practical coding terms, this equates to setting the request header after the call to `open()` but before the call to `send()`.

Related Item: `readyState` property

ECMAScript for XML (E4X)

For HTML element properties, methods, and event handlers, see Chapter 26, "Generic HTML Element Objects."

Properties	Methods	Event Handlers
	`toXMLString()`	

Prior to E4X, the only way to access XML documents with JavaScript was through the DOM at the object level. With the advent of E4X, XML is treated as a primitive type, at the same level as strings, numbers, and Booleans. What does this mean for us? The short version is that

XML access is now simpler for us as web developers, and faster for the end user. An `XML` object created with the XML primitive is not part of the DOM; neither is it a DOM representation of XML. Because it is a primitive, you use it in ways similar to how you use strings, numbers and Booleans. The sample code in this section is in listing `jsb-02.html` on the CD-ROM.

While you can use the standard MIME type with E4X

```
<script type="text/javascript">
```

you could end up with inexplicable syntax errors. This usually occurs for one of two reasons: you are using HTML comments to hide script from older browsers, or you are putting your scripts into XML CDATA sections. If either of these is the case, then you can add the E4X argument to the standard MIME type:

```
<script type="text/javascript; e4x=1">
```

With E4X, you declare an `XML` object, in much the same way that you declare a string or numeric or Boolean object, with the `XML` constructor:

```
var copyrightData1 = new XML();
```

You can then assign XML text to the object:

```
copyrightData1 = <copyright>
                    <date>2010</date>
                    <author>Danny Goodman</author>
                 </copyright>;
```

You can also pass a string to the `XML` constructor:

```
var copyrightText = "<copyright>"
                  + "<date>2010</date>"
                  + "<author>Danny Goodman</author>"
                  + "</copyright>";
var copyrightData2 = new XML(copyrightText);
```

Either way, you create an `XML` object. The XML elements within the object are properties, which means you can use the standard `object.property` notation that you've been using all along in JavaScript. At the very least, you can take your very simple XML document and write it to the current web page:

```
document.write("Copyright by " +
               copyrightData1.author);
```

E4X becomes really exciting when you want the `XML` object to dynamically interact with JavaScript expressions. You do this with curly braces:

```
copyrightDate = new Date();
copyrightData1 = <copyright>
                    <date>{copyrightDate.getFullYear()}</date>
                    <author>Danny Goodman</author>
                 </copyright>;
```

Just as in XML documents, the elements in your `XML` objects can have attributes. Those attribute values can be set dynamically with curly brace notation as well. Notice that you can extract the value of an element's attribute by using dot notation, with the @ operator in front of the attribute name:

```
copyrightData1 = <copyright date={copyrightDate.getFullYear()}>
                    <author>Danny Goodman</author>
                 </copyright>;
document.write("<br />The value of the date attribute" +
               " of the copyright element is " +
               copyrightData1.@date +
               ".");
```

Most XML documents contain repeating sets of elements. You can also create such sets in your `XML` object:

```
copyrightData1 = <copyright date={copyrightDate.getFullYear()}>
                    <author>Danny Goodman</author>
                    <withAuthor>Michael Morrison</withAuthor>
                    <withAuthor>Paul Novitski</withAuthor>
                    <withAuthor>Cynthia Gustaff Rayl</withAuthor>
                 </copyright>;
```

When you access the repeating set, you get back an `XMLList` object. It can be processed almost like an array. There are two primary differences: `length()` is a method and not a property; also, other `Array` object methods are not supported by `XMLList`.

```
document.write("<br >The number of repeating 'with' elements is " +
              copyrightData1.withAuthor.length() +
              ".");
var message = "<br />The JavaScript Bible was written by " +
              copyrightData1.author +
              " with";
for each (i in copyrightData1.withAuthor)
{
   message += "<br />" + i;
}
document.write(message);
```

Methods

toXMLString()

Returns: String

Compatibility: WinIE5-, MacIE-, NN-, FF1.5+, Safari-, Opera-, Chrome-

The `toXMLString()` method returns a string that is the concatenated text of all the text nodes of the XML object.

Related Item: `toString()` method (see Chapter 26)

CHAPTER

HTML Directive Objects

IN THIS CHAPTER

Accessing non-displayed element objects

Linking operating system–specific style sheet definitions

HTML, head, link, title, meta, base, and script elements

Thanks to the modern browser's desire to expose all HTML elements to the document object model, we can now access a variety of objects that represent HTML elements that are normally invisible to the human viewer of a page. These elements are called *directive elements* because they predominantly contain instructions for the browser — instructions that direct the browser to locate associated content on the page, link in external specifications, treat content as executable script statements, and more.

As you browse through the objects of this chapter, you may wonder why they have so many properties that normally indicate that the elements occupy space on the rendered page. After all, how can a `meta` element have dimension or position on the page when it has no renderable content? The reason is that modern browsers internally employ some form of object-oriented behavior that lets all HTML elements — rendered or not — inherit the same set of properties, methods, and event handlers that any generic element has (see Chapter 26, "Generic HTML Element Objects"). The logical flaw is that unrendered elements can have properties and methods that don't genuinely apply to them. In such cases, their property values may be zero, an empty string, or an empty array. Yet the properties and methods exist in the objects just the same. Therefore, despite the large number of objects covered in this chapter, there are relatively few properties and methods that are not shared with all HTML elements (as covered in Chapter 26).

HTML Element Object

For HTML element properties, methods, and event handlers, see Chapter 26.

Properties	Methods	Event Handlers
version		

Syntax

Accessing `html` element object properties or methods:

```
(IE4+)     [window.]document.all.elemID.property | method([parameters])
(IE4+)     [window.]document.body.parentElement.property | method([parameters])
(IE5+/W3C) [window.]document.body.parentNode.property | method([parameters])
(IE5+/W3C) [window.]document.getElementById("elemID").property |
                     method([parameters])
```

About this object

The `html` element is the big wrapper around all other elements of the page. In the object tree, the `html` element sits between the all-encompassing `document` object and the element's most common children, the `head` and `body` elements. Other than one deprecated property (`version`), the `html` element object offers nothing of importance to the scripter — with one possible exception. When your script needs to use methods on the child nodes of the `html` element, you must invoke most of those methods from the point of view of the `html` element. Therefore, you should know how to create a reference to the `html` element object (shown in the preceding "Syntax" section), just in case you need it.

Property

version

Value: String Read-Only

Compatibility: WinIE6+, MacIE5+, NN6+, Moz1+, Safari1+, Opera+, Chrome+s

The `version` property is an artifact of an "ancient" way that an HTML document specified the HTML version of its content. These days, the preferred way to declare the HTML version for a document is through a Document Type Declaration (DTD) statement that precedes the `<html>` tag. An example of a modern DTD statement that accommodates HTML 4, plus deprecated elements and attributes, as well as frameset support is

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Frameset//EN"
   "http://www.w3.org/TR/REC-html40/frameset.dtd">
```

If you're looking to create pages that adhere to a somewhat stricter standard, then the XHTML 1.0 Transitional DTD might be a better option:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
```

See `http://www.w3.org/TR/REC-html40/struct/global.html#h-7.2` for several other possibilities. A DTD statement does not affect the `version` property of an html element object.

head Element Object

For HTML element properties, methods, and event handlers, see Chapter 26.

Properties	Methods	Event Handlers
`profile`		

Syntax

Accessing `head` element object properties or methods:

```
(IE4+)     [window.]document.all.elemID.property | method([parameters])
(IE5+/W3C) [window.]document.getElementById("elemID").property |
                     method([parameters])
```

About this object

The purpose of the `head` element is primarily to act as a container for most of the other HTML directive elements. Other than as a reference point to the child elements nested within, the `head` element object rarely comes into play when scripting a document.

Property

`profile`

Value: String Read-Only

Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+, Opera+, Chrome+

The `profile` property is the script version of the optional `profile` attribute of a `head` element. Although the attribute and property are supported in modern browsers, they are not used in practice yet. You can find details about the attribute at `http://www.w3.org/TR/REC-html40/struct/global.html#profiles`.

Related Items: `meta` element object

base Element Object

For HTML element properties, methods, and event handlers, see Chapter 26.

Properties	Methods	Event Handlers
href		
target		

Syntax

Accessing `base` element object properties or methods:

```
(IE4+)      [window.]document.all.elemID.property | method([parameters])
(IE5+/W3C) [window.]document.getElementById("elemID").property |
                  method([parameters])
```

About this object

The `base` element enables the page author to specify a default server directory and/or link target for the entire page. If you omit the `base` element from the HTML, which is often the case, browsers use the current page's path as the base URL and the current window or frame as the default target. Occasionally, a page generated entirely by way of `document.write()` has difficulty establishing the same `base` URL as the document that generates the content, particularly if the primary page is written out by a server script (in Perl or another language). Including a `<base>` tag in the dynamically-written new page solves the problem; the new page can fetch images or other external elements via relative URLs within the page.

The two distinctive properties of the `base` element object are rarely, if ever, scripted.

Properties

href

Value: URL string Read/Write

Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+, Opera+, Chrome+

The `href` property is generally an absolute URL to the directory you wish to declare as the default directory for the page. Even though browsers automatically set the `base href` to the document's own directory, this object and property do not have any values unless you explicitly set them in a `<base>` tag. In IE, changing this property after a page loads causes the page to re-resolve all relative URLs on the page to the new `base href`. Therefore, if images have relative URLs assigned to their `src` properties (either by way of the tag attribute or script), a change to the `base` element's `href` property forces the browser to look for those same relative URLs in the new directory. If the files aren't there, the images show up broken on the page.

target

Value: String Read/Write

Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+, Opera+, Chrome+

The `target` property governs the default window or frame that is to receive any content coming from a server in response to a click on a link or any other element that has its own `target` attribute. Valid values include the name of any frame (as assigned to the `name` attribute of the `<frame>` tag) or window (as defined by the second attribute of the `window.open()` method). You can also assign standard HTML targets (`_blank`, `_parent`, `_self`, and `_top`) to this property as strings.

link Element Object

For HTML element properties, methods, and event handlers, see Chapter 26.

Properties	Methods	Event Handlers
charset		Onload
disabled		
href		
hreflang		
media		
rel		
rev		
sheet		
styleSheet		
target		
type		

Syntax

Accessing `link` element object properties or methods:

```
(IE4+)     [window.]document.all.elemID.property | method([parameters])
(IE5+/W3C) [window.]document.getElementById("elemID").property |
                method([parameters])
```

About this object

The `link` element (not to be confused with the `a` element that is often referred to as a "link" element when it contains an `href` attribute pointing to another document) has many potential uses in pointing to external documents that relate to the current document. Its most common usage today is for linking an external style sheet specification to the document. In fact, it's not

uncommon for sophisticated site designs to use `document.write()` to generate the `<link>` tag so that operating system–specific style sheets are applied to the page. In the following code fragment (which goes inside a document's `head` element), the page loads a Macintosh-specific style sheet when the page is running on a Macintosh; otherwise, it loads a Windows-specific style sheet:

```
<script type="text/javascript">
var isMac = navigator.userAgent.indexOf("Mac") != -1;
var linkTagStart = "<link rel='stylesheet' type='text/css' href='";
var linkTagEnd = ".css'>";
   if (isMac)
   {
       document.write(linkTagStart + "mac" + linkTagEnd;
   }
   else
   {
       document.write(linkTagStart + "windows" + linkTagEnd;
   }
</script>
```

Although it may appear that the `link` element can load a variety of content into a page, do not use it for multimedia (for which you should use the `object` element) or external HTML (for which you should use an `iframe` element).

Many of the properties of the `link` element object are script representations of HTML 4.0 attributes for the element. However, browsers don't take full advantage of the possibilities available from the `link` element yet, at least not without an add-on browser extension. (For example, a browser can provide arrows to the previous and next documents in a series, as specified by the `rev` and `rel` attributes. But so far, no browser implements this by default.) Properties unique to this object offer scripted access (in various browser versions) to attribute values of the `link` element. Therefore, this chapter does not spend a lot of time on properties that are not in current use.

Properties

`charset`

Value: String Read/Write

Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+, Opera+, Chrome+

The `charset` property advises the browser about the character encoding of the content that will arrive from the external document (assuming you also have the `href` attribute set). Values for this property must match the encoding naming conventions defined in an industry standard registry (`http://www.iana.org/assignments/character-sets`).

`disabled`

Value: Boolean Read/Write

Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+, Opera+, Chrome+

By changing the `disabled` property (the default is `false`), you can turn externally linked content on and off. For example, you can define two different style sheet links in a document that has two `<link>` tags with the `disabled` attribute of one of them set. You can switch between the two style sheets by setting the `disabled` property of one to `true` and the other to `false`.

href

Value: String See Text

Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+, Opera+, Chrome+

Another way to swap style sheets is to modify the value of a single `link` element object's `href` property. The property's value is a URL string.

hrefLang

Value: String Read/Write

Compatibility: WinIE6+, MacIE6+, NN6+, Moz1+, Safari1+, Opera+ Chrome+

The `hrefLang` property is an advisory for the browser (if the browser takes advantage of it) about the written language used for the content to which the `link` element's `href` attribute points. Values for this property must be in the form of the standard language codes (for example, `en-us` for U.S. English). For a complete list of language codes, visit `http://www.devguru.com/technologies/vbscript/QuickRef/LCIDchart.html`.

media

Value: String Read/Write

Compatibility: WinIE4+, MacIE-, NN6+, Moz1+, Safari1+, Opera+, Chrome+

The `media` property is an advisory for the browser about the target output device intended for the content to which the `link` element's `href` attribute points. This is an outgrowth of HTML 4.0 efforts to make way for future browsers and content that can be optimized for devices such as printers, handheld computers, and audio digitizers. The W3C specifies a preliminary set of constant string values for this property's equivalent attribute. So far, browsers recognize (at most) `all` (default), `print`, and `screen`. The notable exception is Opera, which also supports `handheld` as an acceptable value for the `media` property.

rel
rev

Value: String Read/Write

Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+, Opera+, Chrome+

The `rel` and `rev` properties are intended to define relationships in the forward and backward directions with respect to the current document. Browsers have yet to exploit most of the potential of these attributes and properties without an add-on of some sort. For the most part, the attributes solely direct the browser to treat the external content as a style sheet definition file.

A long list of values is predefined for these properties, based on the corresponding attribute values specified in HTML 4.0. If the browser does not respond to a particular value, the value is

simply ignored. You can string together multiple values in a space-delimited list inside a single string. Accepted values are as follows:

alternate	contents	index	start
appendix	copyright	next	stylesheet
bookmark	glossary	prev	subsection
chapter	help	section	

sheet

Value: Object Read-Only

Compatibility: WinIE-, MacIE-, NN6+, Moz1+, Safari-, Opera-, Chrome-

When a `link` element loads an external style sheet, the W3C DOM `sheet` property of the `link` element object provides scripted access to the style sheet rules that belong to that external file. Use properties of the `sheet` object to access specifics about the imported rules.

styleSheet

Value: Object Read-Only

Compatibility: WinIE5+, MacIE-, NN-, Moz-, Safari-, Opera-, Chrome-

When a `link` element loads an external style sheet, the IE-specific `styleSheet` property of the `link` element object provides scripted access to the style sheet rules that belong to that external file. Use properties of the `styleSheet` object (see Chapter 38, "Style Sheet and Style Objects") to access specifics about the imported rules.

target

Value: String Read/Write

Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+, Opera+, Chrome+

In the context of using `link` elements to point to other content associated with the current document (for example, the next and previous documents within a series), the `target` property can advise the browser which frame or window to use to display that content. For example, a suitably equipped browser can display a glossary in a separate window. No browsers currently implement these extended features of the `link` element, so the property is provided in browsers only for compatibility with the W3C standards. If the property were truly functional, it would accept values in the form of a string name for a frame or one of the window constants (`_blank`, `_parent`, `_self`, or `_top`).

type

Value: String Read/Write

Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+, Opera+, Chrome+

The `type` property specifies the `mime` type for the content that will arrive from the external document to which the element's `href` attribute points. `link` elements are used primarily for Cascading Style Sheets, so the property value is `text/css`.

Event handlers

onload

Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-, Opera-, Chrome-

The `onload` event handler fires when the external content pointed to by the `link` element's `href` attribute completes loading. WinIE5 fires this event handler even if the loading does not succeed, so use this event handler with care.

meta Element Object

For HTML element properties, methods, and event handlers, see Chapter 26.

Properties	Methods	Event Handlers
charset		
content		
httpEquiv		
name		
url		

Syntax

Accessing `meta` element object properties or methods:

```
(IE4+)     [window.]document.all.elemID.property | method([parameters])
(IE5+/W3C) [window.]document.getElementById("elemID").property |
                     method([parameters])
```

About this object

In computer terminology, *metadata* usually consists of extra information about the primary data of a document or information collection. In HTML documents, metadata can be additional hidden information about the document, such as the name of the author and keywords. If the browser is suitably equipped, metadata can also include some instructions, such as when to reload the page by itself. `meta` elements add all of this metadata to HTML documents. Both fact

and folklore surround the application of `meta` elements within pages. One fact is that Internet search engine robots used to scour pages for certain kinds of keyword meta tags to help place your page within relevant categories when Web surfers are looking for specific content. (But thanks to keyword loading by unscrupulous web sites, search engines now rarely consider `meta` keywords.) Folklore holds that browsers always respond to `meta` element wording that prevents browsers from copying pages into the cache — when in fact, this behavior is not universal among browsers.

Complete details about `meta` element usage is beyond the scope of this JavaScript book, but you should be aware of one widely accepted composition that enables you to set a page to reload itself (or another page) at a fixed time interval. This is especially useful if your page retrieves very timely information from a database. The format is

```
<meta http-equiv="refresh" content="n,url=url" />
```

`n` is the number of seconds to delay before reloading the page, and `url` is the complete URL of the page to be reloaded. Note that you can specify any page you like. This allows for a kind of slide show to be sequenced in a freestanding kiosk, for example, because each page's `meta` element points to the next page in the series after a fixed amount of time.

Unique properties for the `meta` element object mimic the HTML attributes for the `<meta>` tag. Browsers read most attributes only at load time, which means that scripted changes to values after the page loads have no effect. These properties are rarely, if ever, accessed from a script, so we mention them here only briefly.

Properties

charset

Value: String Read/Write

Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-, Opera-, Chrome-

The `charset` property advises the browser about the character encoding of the content for the page. Values for this property must match the encoding naming conventions defined in an industry standard registry (`http://www.iana.org/assignments/character-sets`).

content

Value: String Read/Write

Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+, Opera+, Chrome+

For many applications of the `meta` element, the `content` property contains the primary value associated with the element. For example, search engines used to look for (and some still do) a `meta` element whose `name` attribute is `"keywords"`. The value of the `content` attribute is a comma-delimited string of keywords that the search engine reads and indexes in its own database. The `content` property simply represents the `content` attribute string. Changing the values by script obviously does nothing to alter the tag values of the page on the server.

`httpEquiv`

Value: String Read/Write

Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+, Opera+, Chrome+

A `meta` element can simulate and extend the transmission of server instructions to the browser — instructions that normally arrive in the form of HTTP headers. These header supplements are supplied in `meta` elements via the `http-equiv` attribute, which is represented in the object model by the `httpEquiv` property. Common values include `refresh` and `expires`. Each of these also requires a `content` attribute that provides necessary details for carrying out the instructions. If you assign a string value to the `httpEquiv` property, be sure the `content` property has a suitable string assigned to it.

`name`

Value: String Read/Write

Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+, Opera+, Chrome+

A `meta` element that includes genuine metadata about the page (for example, author or keywords) usually has a `name` attribute that identifies what the metadata is (analogous to the name of a name-value pair). The `name` and `content` properties go hand in hand because the content string usually must be in a particular form for an external process, such as a search engine, to read the data successfully. Values for the `name` attribute are rarely case-sensitive.

`url`

Value: String Read/Write

Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-, Opera-, Chrome-

If a `meta` element needs to point to a document on the Internet for any reason, the URL of that document is assigned to the `url` attribute of the element. You can modify the value via the `url` property of a `meta` element object. We recommend a complete URL string for the `url` property value.

script Element Object

For HTML element properties, methods, and event handlers, see Chapter 26.

Properties	Methods	Event Handlers
`defer`		
`event`		
`htmlFor`		
`src`		
`text`		
`type`		

Syntax

Accessing `script` element object properties or methods:

```
(IE4+)     [window.]document.all.elemID.property | method([parameters])
(IE5+/W3C) [window.]document.getElementById("elemID").property |
                    method([parameters])
```

About this object

The `<script>` tag is well known to scripters, and modern browsers treat the `script` element as an object that, itself, can be scripted. The circularity of this description isn't as far-fetched as it sounds. Although scripting an existing script is a rarity in practice, it is not out of the question to generate a new `script` element after the page loads. If you use W3C DOM syntax to create a new `script` element, you then need to assign values to the properties that are normally set via the tag's attributes. Thus, scripting a script does make sense.

Unless you have experience with IE's option of binding event handlers to `<script>` tags (see Chapter 25, "Document Object Model Essentials"), some of the properties described next will be foreign to you. Even so, these properties are now a part of the W3C DOM specification.

Properties

`defer`

Value: Boolean Read/Write

Compatibility: WinIE4+, MacIE4+, NN6-, Moz-, Safari-, Opera-, Chrome-

The default process of loading a page that contains scripts is to wait for any immediate script execution to complete before the rest of the page loads. But, if you include a `defer` attribute in the tag, modern browsers continue to load the rest of the page without waiting for immediate scripts to run. The `defer` property enables you to inspect or set that property; its default value is `false`. Once a page loads, any change you make to an existing `script` element's `defer` property has no effect. Although the property is available in modern browsers, only IE responds to setting the property to `true`.

`event`
`htmlFor`

Value: String Read-Only

Compatibility: WinIE4+, MacIE4+, NN6-, Moz-, Safari-, Opera-, Chrome-

Internet Explorer enables you to bind events to script statements when you specify both a `for` and `event` attribute in the `<script>` tag. Statements inside the tag execute only when the

object named by the `for` attribute receives the event named by the `event` attribute. You can examine the `event` attribute by way of the `script` element object's `event` property, and you can view the `for` attribute through the `htmlFor` property. Both properties simply mimic whatever values are assigned to their respective attributes, such as `onclick()` and `myDIV`. Only IE responds to the corresponding attributes of a `<script>` tag.

src

Value: String — See Text

Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+, Opera+, Chrome+

The `src` property is a string of the URL of an external `.js` script file to be linked into a page. You can change this property in IE after you load the external script, but the old script does not go away from the page. If the new script defines the same variable and function names, the new versions overwrite the old. Other browsers may not load a new external script.

text

Value: String — Read-Only

Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+, Opera+, Chrome+

The full text of a `script` element is available for reading through the `text` property. Although IE5+ may give the impression that you can modify this property, the script that loads with the page is the script that's stored in the browser's memory. Thus, the original script statements continue to work even though the object's property is different.

type

Value: String — Read-Only

Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+, Opera+, Chrome+

The `type` attribute was added to the `<script>` tag in HTML 4.0 to help resolve the conflict that the deprecated `language` attribute created for all HTML elements. The value of the attribute (and thus the `type` property) is a `mime` type string. For JavaScript, that value is `text/javascript`.

title Element Object

For HTML element properties, methods, and event handlers, see Chapter 26.

Properties	Methods	Event Handlers
`text`		

Syntax

Accessing `title` element object properties or methods:

```
(IE4+)     [window.]document.all.elemID.property | method([parameters])
(IE5+/W3C) [window.]document.getElementById("elemID").property |
                    method([parameters])
```

About this object

Before the `title` element was accessible to scripting as an object, the prescribed way to get to the content of the page's `<title>` tag was through the `document.title` property. Although that property is still available for backward compatibility, modern scripts should access the `text` property of the `title` element object. As a useful exercise, you can modify Listing 29-15 (loaded via Listing 29-14) to use the W3C DOM syntax to retrieve and display the document's title.

Property

text

Value: String Read/Write

Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+, Opera+, Chrome+

The `text` property represents the text between the start and end tags of the `title` element object. This is simply a convenience property because the text can be referenced by other ways in the IE4+ (`innerText` property), NN6+ (`innerHTML`), and W3C DOM (`firstChild.nodeValue`) syntaxes, with the latter being the preferred approach. For backward compatibility with earlier browsers, you can, alternatively, use the `document.title` property.

Related Items: `document.title` property

CHAPTER

Table and List Objects

IN THIS CHAPTER

Modifying table cell content

Adding and deleting table rows

table, caption, tbody, tfoot, thead, col, colgroup, th, tr, and td element objects

ol, ul, li, and dl list element objects

Tables are incredibly popular HTML constructions. When you consider that a lot of server applications search SQL databases and display data gathered from SQL tables, it's not unusual to find the table concept carried over from data storage to data display. Spreadsheet programs certainly put the notion of tabular display into the minds of most computer users.

One of the truly beneficial properties of tables in HTML is that they pack a lot of page organization and alignment punch in just a few tags and attributes. Even if you're not a graphic designer or a dedicated HTML jockey, you can get rows and columns of text and images to line up perfectly on the page. This behavior also lures many page designers to sculpt elaborately detailed pages out of what appear to be positioned elements. Earlier browsers didn't offer positioning facilities, so borderless tables were torqued into performing all kinds of placement tricks with the help of precisely-sized, transparent images, creating the illusion of white space between carefully placed elements.

Using tables to specify design and page layout is rapidly giving way to CSS techniques that achieve a similar look with less code. This trend is driven by web standards goals of using HTML markup to denote context rather than layout. In fact, it is now considered an antiquated practice to use tables purely for the purpose of laying out pages. Even so, many web pages need to display genuinely columnar data — a purpose for which HTML tables are still ideally suited. The first part of this chapter focuses on the scriptable aspects of `table` element objects and the shopping list of elements that support tables. Later in the chapter, we discuss element objects that create formatted lists in pages.

The Table Object Family Hierarchy

Although most of this discussion of table markup is best left to HTML texts, the structure of a full-fledged table and the relationships among the elements — particularly the parent-child relationships — may affect your scripting and event handling.

You are probably very familiar with the most basic table structure that has been around since the early days of HTML. Such a table (in a 2×2 layout) can have the following form:

```
<table>
   <tr>
      <td></td>
      <td></td>
   </tr>
   <tr>
      <td></td>
      <td></td>
   </tr>
</table>
```

If you want to place a row of cells at the top of each column such that the contents of the cells act as headers for each column, add a row as follows:

```
<table>
   <tr>
      <th></th>
      <th></th>
   </tr>
   <tr>
      <td></td>
      <td></td>
   </tr>
   <tr>
      <td></td>
      <td></td>
   </tr>
</table>
```

You can also include a caption associated with the table. Its tag goes immediately after the `table` element's start tag:

```
<table>
   <caption></caption>
   <tr>
      <th></th>
      <th></th>
   </tr>
   <tr>
      <td></td>
      <td></td>
```

```
    </tr>
    <tr>
       <td></td>
       <td></td>
    </tr>
 </table>
```

In line with its emphasis on providing contextual tags, HTML 4.0 added three tags that enable you to define groups of table rows according to whether they are the header, body, or footer of the table (`thead`, `tbody`, and `tfoot` elements, respectively). A table footer, for example, can display column totals. The only seemingly illogical rule about these elements is that you should define the `tfoot` element and its row contents before the `tbody` element(s) in the table. Even with this source code placement, the `tfoot` row appears at the bottom of the table.

Some browsers produce visual dividers between these sections (WinIE5+ does a nice job of this). Moreover, you can have multiple `tbody` sections within a table. Some browsers render dividers between these `tbody` sections (again, WinIE5+ does it well). Regardless of the built-in divider support, these contextual groupings also enable you to assign style sheets to HTML tag selectors, so that you don't have to dream up a scheme of class and `id` names tied to style sheet rules. Building upon the skeletal table shown thus far, you add the `thead` and `tbody` elements like this:

```
<table>
   <caption></caption>
   <thead>
      <tr>
         <th></th>
         <th></th>
      </tr>
   </thead>
   <tbody>
      <tr>
         <td></td>
         <td></td>
      </tr>
      <tr>
         <td></td>
         <td></td>
      </tr>
   </tbody>
</table>
```

That's the extent of table-oriented HTML containers. The remaining two elements, `colgroup` and `col`, provide a different "slice" of the table for style sheets and other visual groupings. One of the most obvious purposes of these two elements is to assign a width or other style to all cells in a particular column or group of columns. You can also use these elements to group adjacent columns so that dividers are drawn between groups of columns — if the browser (such as WinIE5+) supports dividers between column groups — without specifying global table borders. You can see an example of the HTML for a complex table in the HTML 4.0 specification (`http://www.w3.org/tr/REC-html40/struct/tables.html#h-11.5`). Elsewhere on

that same page, you can find the formal specification for all table-related tags and attributes as defined by the W3C.

Populating table cells

Source material for a table's content can come from many different places. Most of the tables you see on the Web are hard-coded in the HTML. That is, the content of the table is fixed inside a static HTML file on the server.

But tables may also convey content from live databases or content that changes more frequently than content that's manually updated by the web site's author. The source and your web development infrastructure (not to mention your technical skills) dictate other avenues for populating tables.

After hard-coded HTML files, the next most common way to generate tables is through server-based CGI programs. These programs (written in Perl, C, and many other languages, including server-side JavaScript on those few servers that support it) generally compose a query for the database and then repackage the data returned from the database into HTML-formatted pages.

A more client-side–oriented approach is to let JavaScript apply the `document.write()` method to compose the table's tags as the page loads. Data for the cells can come from JavaScript arrays defined at the beginning of the document or from arrays defined in external `.js` library files that are linked in as the page loads. In the newest browsers, the data may come from blocks of XML-formatted data stuffed into the document. These solutions can work in situations where you need to update the table data periodically, but the table delivered to the client does not reflect the instantaneous state of a database. For example, a daily batch program on a server can capture the day's sales totals and write out a `.js` text file to a known place on the server. The file consists entirely of JavaScript array definitions. When the HTML page loads, the current `.js` file is automatically loaded into the page, and `document.write()` statements compose the table's HTML from the data supplied in the arrays. Although the script that assembles the HTML for the tables might appear formidable to a nonscripter, a nonscripter can also manually update the array data by following a template format supplied by the programmer.

Finally, if your page visitors run IE4+, you can take advantage of a Microsoft-specific technology called *data binding*. Data binding invokes the powers of one or more ActiveX controls that come with the IE browser (simulated in MacIE). These objects (collectively called Data Source Objects) let HTML pages access ODBC databases and structured text files (MacIE works only with text files). As the page loads, the table fills with data pulled live from the database. You can see an example of data binding in Chapter 26, "Generic HTML Element Objects," under the description of the data binding property: `dataFld`. The HTML file carries tags for only one row of cells, but data binding fills in the rest of the rows and cells.

Modifying table cell content

You can modify the HTML content of a table cell directly, in modern browsers, through the `td` element's `innerHTML` property (a Microsoft invention that is not sanctioned by the W3C DOM

Level 2, but is supported elsewhere as a de facto standard). Even if the content is simply text that is to inherit the style format of the surrounding td element, you can still use the innerHTML property. If the size of the new content affects the dimensions of the cell's column width or row height, the browser reflows the rest of the table content around it.

Even better than the innerHTML property is the W3C DOM form of modifying an element's content, which enables you to generate the new content via the document.createElement() or document.createTextNode() sequence and assign that new content to the cell by way of the td element's replaceChild() method.

Listing 41-1 shows a synthesis of different techniques to effect cell content replacement, including script code branches that emulate the appearance of replacement in NN4. The table represents only one line of what might be an order form for several products. As the user makes a selection of the quantity, the extended total is displayed in the rightmost column.

Although the page shown in Listing 41-1 consists of only one row of data, the scripts and naming conventions are intended to be carried out among multiple rows. The product name appears in several object names and ids in each row, and the scripts count on the convention being followed throughout. In fact, the regularity of the namings can allow the content for a table's row to form a script function that is invoked for each table row. The product code name can be passed as the parameter, and all object names and ids can be assembled in that function. The regularity of table content often lends itself to script-generated construction.

LISTING 41-1

Replacing Table Cell Content

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Modifying Table Cell Content</title>
      <style type="text/css">
         .absoluteWrap {position:absolute;}
         .relativeWrap {position:relative;}
         .total {color:red;}
      </style>
      <script type="text/javascript">
         // calculate and display a row's total
         function showTotal(qtyList)
         {
            var qty = qtyList.options[qtyList.selectedIndex].value;
            var prodID = qtyList.name;
            var total = "US$" + (qty * parseFloat(qtyList.form.elements[prodID +
                 "Price"].value));
            var newCellHTML = "<span class='total'>" + total + "</span>";

            var prodElem = document.getElementById(prodID + "Total");
            while (prodElem.firstChild)
```

continued

LISTING 41-1 *(continued)*

```
            {
               prodElem.removeChild(prodElem.firstChild);
            }
            var totalElem = document.createElement("span");
            totalElem.appendChild(document.createTextNode(total));
            totalElem.style.color = "red";
            prodElem.appendChild(totalElem);
         }

         // display content for all products (e.g., in case of Back navigation)
         function showAllTotals(form)
         {
            for (var i = 0; i < form.elements.length; i++)
            {
               if (form.elements[i].type == "select-one")
               {
                  showTotal(form.elements[i]);
               }
            }
         }
      </script>
   </head>
   <body onload="showAllTotals(document.orderForm)">
      <h1>Modifying Table Cell Content</h1>
      <hr />
      <form name="orderForm">
         <table border="1">
            <colgroup width="150"></colgroup>
            <colgroup width="100"></colgroup>
            <colgroup width="50"></colgroup>
            <colgroup width="100"></colgroup>
            <tr>
               <th>Product Description</th>
               <th>Price Each</th>
               <th>Quantity</th>
               <th>Total</th>
            </tr>
            <tr>
               <td>Wonder Widget 9000</td>
               <td>US$125.00</td>
               <td>
                  <select name="ww9000" onchange="showTotal(this)">
                     <option value="0">0</option>
                     <option value="1">1</option>
                     <option value="2">2</option>
                     <option value="3">3</option>
                  </select>
                  <input type="hidden" name="ww9000Price" value="125.00" />
               </td>
```

```
            <td>
               <script type="text/javascript">
                  document.write("<span id='ww9000Total' ↩
                    class='relativeWrap'><p> <\/p><\/span>");
               </script>
            </td>
         </tr>
      </table>
   </form>
</body>
</html>
```

Modifying table rows

In modern browsers, all table-related elements are full-fledged objects within the browser's object model. This means that you are free to use your choice of DOM element modification techniques on the row and column makeup of a table. But due to the frequent complexity of tables and all of their nested elements, the code required to manage a table can balloon in size. To the rescue, come some methods that enable you to add and remove rows and cells from a table. Despite minor differences in the implementations of these methods across DOMs, the syntax exhibits sufficient unanimity to allow one set of code to work on both browsers — especially for adding elements to a table.

Table 41-1 provides a quick summary of the key methods used to add or remove elements within a `table`, a table section (`thead`, `tbody`, or `tfoot`), and a row (`tr`). For simple tables (in other words, those that do not define `thead` or `tfoot` segments), you can work exclusively with the row modification methods of the `table` element object (and then the cell modification methods of the rows within the `table` element). The reason for the duplication of the row methods in the table section objects is that instead of having to worry about row index numbers lining up among the combined total of head, body, and foot rows, you can treat each segment as a distinct unit. For example, if you want to add a row just to the beginning of the `tfoot` section, you don't have to count up the `tr` elements and perform arithmetic to arrive at the desired row number. Instead, simply use the `insertRow()` method on the `tfoot` element object and supply the method with parameters that ensure the row is inserted as the first row of the element.

Note

IE5 for the Macintosh offers unpredictable results when inserting rows of a table via these methods. The browser does behave when modifying the HTML elements by accumulating the HTML for a row as a string, and then adding the row to the table via IE DOM methods such as `insertAdjacentHTML()`. If your pages must modify the composition of tables after the page loads — and your audience includes MacIE5 users — use the element and node insertion techniques rather than the methods shown in Table 41-1 and the techniques described next. ■

TABLE 41-1

IE4+ and NN6+/W3C Table Modification Methods

table	thead, tbody, tfoot	tr
insertRow()	insertRow()	insertCell()
deleteRow()	deleteRow()	deleteCell()
createTHead()		
deleteTHead()		
createTFoot()		
deleteTFoot()		
createCaption()		
deleteCaption()		

The basic sequence for inserting a row into a table entails the following steps:

1. Invoke `insertRow()` and capture the returned reference to the new, unpopulated row.
2. Use the reference to the row to invoke `insertCell()` for each cell in the row, capturing the returned reference to each new, unpopulated cell.
3. Assign values to properties of the cell, including its content.

The following code fragment appends a new row to a table (`myTABLE`) and supplies information for the two cells in that row:

```
// parameter of -1 appends to table
// (you can use document.all.myTABLE.insertRow(-1) for IE4+ only)
   var newRow = document.getElementById("myTABLE").insertRow(-1);
// parameter of 0 inserts at first cell position
   var newCell = newRow.insertCell(0);
   newCell.innerHTML = "Mighty Widget 2000";
// parameter of 1 inserts at second cell position
   newCell = newRow.insertCell(1);
   newCell.innerHTML = "Release Date TBA";
```

Note

Although we encourage you to use the DOM approach (`removeChild()/appendChild()`) to modifying the content within a node, as is used in all of the complete code listings in this chapter, in this particular example the table-specific code is easier to read using the Microsoft-specific `innerHTML` approach. ■

A key point to note about this sequence is that the `insertRow()` and `insertCell()` methods do their jobs before any content is handed over to the table. In other words, you first create the HTML space for the content and then add the content.

Listing 41-2 presents a living environment that adds and removes `thead`, `tr`, and `tfoot` elements to an empty table in the HTML. The only subelement inside the `table` element is a `tbody` element, which directs the insertion of table rows so as not to disturb any existing `thead` or `tfoot` elements. You can also see how to add or remove a caption from a table via caption-specific methods.

Each table row consists of the hours, minutes, seconds, and milliseconds of a time stamp generated when you add the row. The color of any freshly added row in the `tbody` is a darker color than the normal `tbody` rows. This is so you can see what happens when you specify an index value to the `insertRow()` method. Some of the code here concerns itself with enabling and disabling form controls and updating `select` elements, so don't be deterred by the length of Listing 41-2.

LISTING 41-2

Inserting/Removing Row Elements

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Modifying Table Cell Content</title>
      <style type="text/css">
         thead {background-color:lightyellow; font-weight:bold;}
         tfoot {background-color:lightgreen; font-weight:bold;}
         #myTABLE {background-color:bisque;}
      </style>
      <script type="text/javascript">
         var theTable, theTableBody;
         function init()
         {
            theTable = (document.all) ? document.all.myTABLE
                                      : document.getElementById("myTABLE");
            theTableBody = theTable.tBodies[0];
         }
         function appendRow(form)
         {
            insertTableRow(form, -1);
         }

         function addRow(form)
         {
            insertTableRow(form, form.insertIndex.value);
         }

         function insertTableRow(form, where)
         {
            var now = new Date();
            var nowData = [now.getHours(), now.getMinutes(), now.getSeconds(),
                           now.getMilliseconds()];
```

continued

LISTING 41-2 *(continued)*

```
      clearBGColors();
      var newCell;
      var newRow = theTableBody.insertRow(where);
      for (var i = 0; i < nowData.length; i++)
      {
         newCell = newRow.insertCell(i);
           while(newCell.firstChild)
         {
            newCell.removeChild(newCell.firstChild);
         }

         newCell.appendChild(document.createTextNode(nowData[i]));
         newCell.style.backgroundColor = "salmon";
      }
      updateRowCounters(form);
   }

   function removeRow(form)
   {
      theTableBody.deleteRow(form.deleteIndex.value);
      updateRowCounters(form);
   }

   function insertTHEAD(form)
   {
      var THEADData = ["Hours","Minutes","Seconds","Milliseconds"];
      var newCell;
      var newTHEAD = theTable.createTHead();
      newTHEAD.id = "myTHEAD";
      var newRow = newTHEAD.insertRow(-1);
      for (var i = 0; i < THEADData.length; i++)
      {
         newCell = newRow.insertCell(i);
         while(newCell.firstChild)
         {
            newCell.removeChild(newCell.firstChild);
         }
         newCell.appendChild(document.createTextNode(THEADData[i]));
      }
      updateRowCounters(form);
      form.addTHEAD.disabled = true;
      form.deleteTHEAD.disabled = false;
   }

   function removeTHEAD(form)
   {
      theTable.deleteTHead();
      updateRowCounters(form);
      form.addTHEAD.disabled = false;
      form.deleteTHEAD.disabled = true;
   }
```

```
function insertTFOOT(form)
{
   var TFOOTData = ["Hours","Minutes","Seconds","Milliseconds"];
   var newCell;
   var newTFOOT = theTable.createTFoot();
   newTFOOT.id = "myTFOOT";
   var newRow = newTFOOT.insertRow(-1);
   for (var i = 0; i < TFOOTData.length; i++)
   {
      newCell = newRow.insertCell(i);
      while(newCell.firstChild)
      {
         newCell.removeChild(newCell.firstChild);
      }
      newCell.appendChild(document.createTextNode(TFOOTData[i]));
   }
   updateRowCounters(form);
   form.addTFOOT.disabled = true;
   form.deleteTFOOT.disabled = false;
}

function removeTFOOT(form)
{
   theTable.deleteTFoot();
   updateRowCounters(form);
   form.addTFOOT.disabled = false;
   form.deleteTFOOT.disabled = true;
}

function insertCaption(form)
{
   var captionData = form.captionText.value;
   var newCaption = theTable.createCaption();
   while(newCaption.firstChild)
   {
      newCaption.removeChild(newCaption.firstChild);
   }
   newCaption.appendChild(document.createTextNode(captionData));
   form.addCaption.disabled = true;
   form.deleteCaption.disabled = false;
}

function removeCaption(form)
{
   theTable.deleteCaption();
   form.addCaption.disabled = false;
   form.deleteCaption.disabled = true;
}

// housekeeping functions
function updateRowCounters(form)
{
   var sel1 = form.insertIndex;
```

continued

LISTING 41-2 *(continued)*

```
            var sel2 = form.deleteIndex;
            sel1.options.length = 0;
            sel2.options.length = 0;
            for (var i = 0; i < theTableBody.rows.length; i++)
            {
               sel1.options[i] = new Option(i, i);
               sel2.options[i] = new Option(i, i);
            }
            form.removeRowBtn.disabled = (i==0);
         }

         function clearBGColors()
         {
            for (var i = 0; i < theTableBody.rows.length; i++)
            {
               for (var j = 0; j < theTableBody.rows[i].cells.length; j++)
               {
                  theTableBody.rows[i].cells[j].style.backgroundColor = "";
               }
            }
         }
      </script>
   </head>
   <body onload="init()">
      <h1>Modifying Tables</h1>
      <hr />
      <form name="controls">
         <fieldset>
            <legend>Add/Remove Rows</legend>
            <table width="100%" cellspacing="20">
               <tr>
                  <td>
                     <input type="button" value="Append 1 Row"
                            onclick="appendRow(this.form)" />
                  </td>
                  <td>
                     <input type="button" value="Insert 1 Row"
                            onclick="addRow(this.form)" /> at index:
                     <select name="insertIndex">
                        <option value="0">0</option>
                     </select>
                  </td>
                  <td>
                     <input type="button" name="removeRowBtn"
                            value="Delete 1 Row" disabled="disabled"
                            onclick="removeRow(this.form)" /> at index:
                     <select name="deleteIndex">
                        <option value="0">0</option>
                     </select>
                  </td>
               </tr>
```

```
            </table>
         </fieldset>
         <fieldset>
            <legend>Add/Remove THEAD and TFOOT</legend>
            <table width="100%" cellspacing="20">
               <tr>
                  <td>
                     <input type="button" name="addTHEAD" value="Insert THEAD"
                            onclick="insertTHEAD(this.form)" />
                     <br />
                     <input type="button" name="deleteTHEAD"
                            value="Remove THEAD" disabled="disabled"
                            onclick="removeTHEAD(this.form)" />
                  </td>
                  <td>
                     <input type="button" name="addTFOOT" value="Insert TFOOT"
                            onclick="insertTFOOT(this.form)" />
                     <br />
                     <input type="button" name="deleteTFOOT"
                            value="Remove TFOOT" disabled="disabled"
                            onclick="removeTFOOT(this.form)" />
                  </td>
               </tr>
            </table>
         </fieldset>
         <fieldset>
            <legend>Add/Remove Caption</legend>
            <table width="100%" cellspacing="20">
               <tr>
                  <td>
                     <input type="button" name="addCaption" value="Add Caption"
                            onclick="insertCaption(this.form)" />
                  </td>
                  <td>Text:
                     <input type="text" name="captionText" size="40"
                            value="Sample Caption" />
                  </td>
                  <td>
                     <input type="button" name="deleteCaption"
                            value="Delete Caption" disabled="disabled"
                            onclick="removeCaption(this.form)" />
                  </td>
               </tr>
            </table>
         </fieldset>
      </form>
      <hr />
      <table id="myTABLE" cellpadding="10" border="1">
         <tbody></tbody>
      </table>
   </body>
</html>
```

Note

The property assignment event handling technique used in the examples throughout this chapter is a deliberate simplification to make the code more readable. It is generally better to use the more modern approach of binding events using the `addEventListener()` **(NN6+/Moz/W3C) or** `attachEvent()` **(IE5+) methods. A modern cross-browser event handling technique is explained in detail in Chapter 32, "Event Objects."** ■

Modifying table columns

Unlike the table row-oriented elements, such as `tbody`, the `col` and `colgroup` elements are not containers of cells. Instead, these elements serve as directives for the rendering of columns within a table. But through scripting, you can add or remove one or more columns from a table on the fly. There is no magic to it; you simply insert or delete the same-indexed cell from every row of the table.

Listing 41-3 demonstrates adding and removing a left-hand column of a table. The table presents the four longest rivers in Africa, and the new column provides the numeric ranking. Thanks to the regularity of this table, the values for the rankings can be calculated dynamically. Note, too, that the `className` property of each new table cell is set to a class that has a style sheet rule defined for it. Instead of inheriting the style of the table, the cells obey the more specific background color and font weight rules defined for the cells.

LISTING 41-3

Modifying Table Columns

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Modifying Table Columns</title>
      <style type="text/css">
         thead {background-color:lightyellow; font-weight:bold;}
         .rankCells {background-color:lightgreen; font-weight:bold;}
         #myTABLE {background-color:bisque;}
      </style>
      <script type="text/javascript">
         var theTable, theTableBody;
         function init()
         {
            theTable = (document.all) ? document.all.myTABLE
                                      : document.getElementById("myTABLE");
            theTableBody = theTable.tBodies[0];
         }
```

```
      function insertColumn(form)
      {
         var oneRow, newCell, rank;
            if (theTable.tHead)
            {
               oneRow = theTable.tHead.rows[0];
               newCell = oneRow.insertCell(0);
               while(newCell.firstChild)
               {
                  newCell.removeChild(newCell.firstChild);
               }
               newCell.appendChild(document.createTextNode("Ranking"));
            }
         rank = 1;
         for (var i = 0; i < theTableBody.rows.length; i++)
         {
            oneRow = theTableBody.rows[i];
            newCell = oneRow.insertCell(0);
            newCell.className = "rankCells";
            while(newCell.firstChild)
            {
               newCell.removeChild(newCell.firstChild);
            }
            newCell.appendChild(document.createTextNode(rank++));
         }
         form.addColumn.disabled = true;
         form.removeColumn.disabled = false;
      }

      function deleteColumn(form)
      {
         var oneRow;
            if (theTable.tHead)
            {
               oneRow = theTable.tHead.rows[0];
               oneRow.deleteCell(0);
            }
         for (var i = 0; i < theTableBody.rows.length; i++)
         {
            oneRow = theTableBody.rows[i];
            oneRow.deleteCell(0);
         }
         form.addColumn.disabled = false;
         form.removeColumn.disabled = true;
      }
   </script>
</head>
```

continued

LISTING 41-3 *(continued)*

```
<body onload="init()">
   <h1>Modifying Table Columns</h1>
   <hr />
   <form name="controls">
      <fieldset>
         <legend>Add/Remove Left Column</legend>
         <table width="100%" cellspacing="20">
            <tr>
               <td>
                  <input type="button" name="addColumn"
                         value="Insert Left Column"
                         onclick="insertColumn(this.form)" />
               </td>
               <td>
                  <input type="button" name="removeColumn"
                         value="Remove Left Column" disabled="disabled"
                         onclick="deleteColumn(this.form)" />
               </td>
            </tr>
         </table>
      </fieldset>
   </form>
   <hr />
   <table id="myTABLE" cellpadding="5" border="1">
      <thead id="myTHEAD">
         <tr>
            <td>River</td>
            <td>Outflow</td>
            <td>Miles</td>
            <td>Kilometers</td>
         </tr>
      </thead>
      <tbody>
         <tr>
            <td>Nile</td>
            <td>Mediterranean</td>
            <td>4160</td>
            <td>6700</td>
         </tr>
         <tr>
            <td>Congo</td>
            <td>Atlantic Ocean</td>
            <td>2900</td>
            <td>4670</td>
         </tr>
         <tr>
            <td>Niger</td>
            <td>Atlantic Ocean</td>
            <td>2600</td>
```

```
            <td>4180</td>
         </tr>
         <tr>
            <td>Zambezi</td>
            <td>Indian Ocean</td>
            <td>1700</td>
            <td>2740</td>
         </tr>
      </tbody>
   </table>
</body>
</html>
```

W3C DOM table object classes

If you ever read the W3C DOM Level 2 specification, notice that the objects defined for tables do not align themselves fully with the actual elements defined in the HTML 4.0 specification. That's not to say the DOM scoffs at the HTML spec; rather, the needs of a DOM with respect to tables differ a bit. For example, as far as the W3C DOM is concerned, the `thead`, `tbody`, and `tfoot` are all regarded as table sections and are thus known as `HTMLTableSectionElement` objects. In other words, in the W3C DOM, there is no particular distinction among the types of table section elements. They're all lumped together, and they bear the same properties and methods. With their strong adherence to the W3C DOM, Mozilla-based browsers and Safari stick to the W3C DOM object constructions.

When you work in both the IE and W3C DOMs at the same time, it's helpful to know the relationships between the object naming conventions used in each. Table 41-2 provides a quick cross-reference between the object types in both DOMs. None of the terminology in Table 41-2 affects the way scripts construct references to elements or the way elements are nested within one another. The containment hierarchy is driven by the HTML element containment — and that remains the same regardless of DOM exposure.

TABLE 41-2

Table Object Classifications

W3C DOM (NN6+/Mozilla/WebKit browsers)	IE4+ and HTML
`HTMLTableElement`	`table`
`HTMLTableCaptionElement`	`caption`
`HTMLTableColElement`	`col, colgroup`
`HTMLTableSectionElement`	`tbody, tfoot, thead`
`HTMLTableRowElement`	`tr`
`HTMLTableCellElement`	`td, th`

Although the following object-specific discussions list the objects according to their HTML tag names, we group these objects according to the W3C DOM classifications because element objects that share a classification also share the same properties, methods, and event handlers.

table Element Object

For HTML element properties, methods, and event handlers, see Chapter 26, "Generic HTML Element Objects."

Properties	Methods	Event Handlers
align	createCaption()	onscroll†
background	createTFoot()	
bgColor	createTHead()	
border	deleteCaption()	
borderColor	deleteRow()	
borderColorDark	deleteTFoot()	
borderColorLight	deleteTHead()	
caption	firstPage()	
cellPadding	insertRow()	
cells	lastPage()	
cellSpacing	moveRow()	
cols	nextPage()	
datePageSize	previousPage()	
frame	refresh()	
height		
rows		
rules		
summary		
tbodies		
tfoot		
thead		
width		

†See Chapter 29, "Document and Body Objects."

Syntax

Accessing `table` element object properties and methods:

```
(IE4+)        [window.]document.all.elemID.property | method([parameters])
(IE5+/W3C)    [window.]document.getElementById("elemID").property |
              method([parameters])
```

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

About this object

The `table` element object is the outermost container of table-related information. The HTML element has a large number of attributes, most of which are echoed by their counterpart properties in the object model. You will rarely modify these properties if the values are set in the tag's attributes. However, if you construct a new `table` element object for insertion into the page, use these properties to assign values to the equivalents of the element's attributes.

A number of additional properties return collections of cell, row, and row section objects; still more properties return references to other, singular objects within the table (such as the `caption` element object). For example, if your script needs to iterate through all rows within just the `tbody` elements (in other words, without affecting the rows in the `thead` element), your script can perform a nested `for` loop to access each row:

```
var oneTBody, oneRow;
for (var i = 0; i < tableRef.tBodies.length; i++)
{
   oneTBody = tableRef.tBodies[i];
   for (var j = 0; j < oneTBody.rows.length; j++)
   {
      oneRow = oneTBody.rows[j];
      // more stuff working on each row
   }
}
```

For a simple table that does not define table row sections, you can iterate through the `rows` collection property of a `table` element object. You can even access cells directly; but it may be easier to keep track of cells in a loop by going through them row by row (via the `cells` property of each `tr` element object).

A large number of methods enable you to modify the structure of a table (as described earlier in this chapter), but they primarily work with rows. Column modifications require a different approach, as also demonstrated earlier.

Properties

`align`

Value: String (`center`, `left`, `right`) Read/Write

Compatibility: WinIE5+, MacIE5+, NN6+, Moz+, Safari+, Opera+, Chrome+

The `align` property controls the horizontal alignment of the table with respect to the next outermost container that provides positioning context. Most typically, the next outermost positioning container is the `body` element. Modifications to this property on an existing table cause the surrounding content to reflow on the page. Be sure you test the consequences of any modification with a variety of browser window sizes.

Example

Use The Evaluator (Chapter 4, "JavaScript Essentials") to see the `align` property at work. The default value (`left`) is in force when the page loads. But you can shift the table to right-align with the body by entering the following statement into the top text box:

```
document.getElementById("myTable").align = "right"
```

Related Item: `style.align` property

`background`

Value: URL string Read/Write

Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-, Opera-, Chrome-

Only IE4+ makes a provision for assigning a background image to a table, and the `background` property controls that value. You can swap out an image by assigning a new URL to the `background` property. The image appears in front of any background color assigned to the table. Thus, you can assign attributes for both characteristics so that there is at least a background color (and an image for IE users).

Example

Treat the `background` property of a table like you do the `src` property of an `img` element object. If you pre-cache an image, you can assign the `src` property of the pre-cached `image` object to the `background` property of the table for quick image changing. Such an assignment statement looks like the following:

```
document.all.myTable.background = imgArray["myTableAlternate"].src;
```

Related Item: `IMG.src` property

`bgColor`

Value: Color value string Read/Write

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

The `bgColor` attribute controls the background color of a table (the `bgcolor` attribute). Colors assigned to the entire table are overridden if colors are assigned to row, row groups, or cells within the table. If you set the `bgColor` property, the `backgroundColor` style property is not affected. Assign values in any acceptable color string format, such as hexadecimal triplets (for example, `"#FCFC00"`) or the generally recognized plain-language names (for example, `"cornflowerblue"`).

Example

Use The Evaluator (Chapter 4, "JavaScript Essentials") to assign a color to the table. After looking at the table to see its initial state, enter the following statement into the top text box:

```
document.getElementById("myTable").bgColor = "lightgreen"
```

When you look at the table again, you see that only some of the cells turned green. This is because colors also are assigned to table elements nested inside the outermost table element, and the color specification closest to the actual element wins the contest. Opera currently does not support the HTML `bgcolor` attribute on the `table` element, so you'll see the entire table turn to green.

Related Item: `style.backgroundColor` property

`border`

Value: Integer Read/Write

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

The `border` property controls the thickness of the table's borders. Values indicate the number of pixels thick the border should be. A value of zero removes all visible borders surrounding the table. Different browsers render table cell borders differently, depending on background colors and other visual attributes of tables and table elements. Be sure to verify the appearance on as many browsers and operating systems as possible.

Example

To remove all traces of an outside border of a table (and, in some combinations of attributes of other table elements, borders between cells), use the following statement:

```
document.getElementById("myTable").border = 0;
```

Related Item: `borderColor` property

`borderColor`
`borderColorDark`
`borderColorLight`

Value: Color value string Read/Write

Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-, Opera-, Chrome-

IE4+ provides attributes and corresponding properties to control the border colors of a table. When table borders have enough thickness to display a three-dimensional raised look, the appearance is created by generating two dark and two light edges (simulating a light source coming from the upper-left or lower-right corner). If you want to do a better job of specifying the color combinations for the light and dark edges, you can control them individually via the `borderColorLight` and `borderColorDark` properties, respectively. You can assign colors in any valid color value (hexadecimal triplet or plain-language name); but when you read the property, the value is returned as a hexadecimal triplet (for example, `"#008000"`).

Example

Assuming that you have set the initial light and dark color attributes of a table, the following function swaps the light and dark colors to shift the light source to the opposite corner:

```
function swapColors(tableRef)
{
   var oldLight = tableRef.borderColorLight;
   tableRef.borderColorLight = tableRef.borderColorDark;
   tableRef.borderColorDark = oldLight;
}
```

Although you can easily invoke this function over and over by ending it with a `setTimeout()` method that calls this function after a fraction of a second, the results are very distracting to the person trying to read your page. Please don't do it.

Related Item: `td.borderColor` property

`caption`

Value: `caption` element object reference   Read/Write (see text)

Compatibility: WinIE4+, MacIE5+, NN6+, Moz+, Safari+, Opera+, Chrome+

The `caption` property returns a reference to the `caption` element object that is nested inside the current table. If there is no `caption` element, the value is `null`. You can use this property as a shortcut reference to the `caption` element if you need to read or modify that element's properties. The property is read/write, provided that you create a valid `caption` element object and assign that new object to the `caption` property.

Example

The following example, for use with The Evaluator (Chapter 4, "JavaScript Essentials"), demonstrates the sequence of assigning a new `caption` element object to a table. Although the table in The Evaluator already has a `caption` element, the following statements replace it with an entirely new one. Enter each of the following statements into the top text box, starting with the one that saves a long reference into a variable for multiple use at the end:

```
t = document.getElementById("myTable")
a = document.createElement("caption")
b = document.createTextNode("A Brand New Caption")
a.appendChild(b)
t.replaceChild(a, t.caption)
```

A view of the table shows that the new caption has replaced the old one because a table can have only one `caption` element.

Related Item: `caption` element object

cellPadding
cellSpacing

Value: Integer Read/Write

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

The `cellPadding` property is a table-wide specification for the blank space inserted between the edge of a table cell and the content of the cell. One value affects the padding on all four sides. The effect of cell padding is especially apparent when there are borders between cells; in this case, the padding provides welcome breathing space between the border and content. The `cellSpacing` property influences the thickness of borders between cells. If no visible borders are present between cells in a table, you can usually set either `cellpadding` or `cellspacing` to provide the desired blank space between cells.

Example

Use The Evaluator (Chapter 4, "JavaScript Essentials") to adjust the `cellPadding` and `cellSpacing` properties of the demonstrator table. First, adjust the padding:

```
document.getElementById("myTable").cellPadding = 50
```

Now, adjust the cell spacing:

```
document.getElementById("myTable").cellSpacing = 15
```

Notice how `cellSpacing` affects the thickness of inter-cell borders.

Related Item: `border` property

cells

Value: Array Read-Only

Compatibility: WinIE5+, MacIE-, NN-, Moz-, Safari-, Opera-, Chrome-

The `cells` property returns an array (collection) of all `td` and `th` element objects within the entire table. From the perspective of the `table` element object, this "view" encompasses all cells — whether they are inside a table row segment (for example, a `thead`) or in a freestanding row. In the W3C DOM, the `cells` collection is accessible only as a property of a `tr` object. However, a `rows` collection is available from all table container elements, thus enabling you to iterate through all cells of all rows.

Example

Use The Evaluator (Chapter 4, "JavaScript Essentials") with WinIE5+ to have JavaScript calculate the number of columns in the demonstrator table with the help of the `cells` and `rows` properties. Enter the following statement into the top text box:

```
document.all.myTable.cells.length/document.all.myTable.rows.length
```

The result is the number of columns in the table.

Related Items: `rows`, `tr.cells` properties

cols

Value: Integer Read/Write

Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-, Opera-, Chrome-

The `cols` property represents the IE-specific `cols` attribute for `table` elements. Specifying this attribute should speed table rendering. If you don't specify the attribute explicitly in your HTML, the property has a value of zero — the property does not tell you the size of your table dynamically. Although this property is read/write, you cannot use this property to add or remove columns from a table. Instead, use the table modification methods discussed later in this section.

Related Item: `rows` property

dataPageSize

Value: Integer Read/Write

Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-, Opera-, Chrome-

When using IE4+ data binding to obtain table data from a data source, there may be more rows or data (records) than you wish to display in one table. If so, you can define the number of rows (records) that constitutes a "page" of data within the table. With this limit installed for the table, you can then use the `firstPage()`, `previousPage()`, `nextPage()`, and `lastPage()` methods to access another page relative to the currently-viewed page. Although you usually establish this value via the `datapagesize` attribute of the `table` element, you can adjust it later via the `dataPageSize` property to show more or fewer records per "page" in the table.

Example

If you wanted to change the number of visible rows of linked data in the table to 15, you'd use the following statement:

```
document.all.myTable.dataPageSize = 15;
```

Or, you could use the more standardized W3C approach:

```
document.getElementById("myTable").dataPageSize = 15;
```

Related Items: `dataSrc`, `dataFld` properties; `firstPage()`, `lastPage()`, `nextPage()`, `previousPage()` methods

frame

Value: String constant Read/Write

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

The `frame` property enables you to control which side or sides of the table's border should be displayed. Values for this property can be any of a fixed set of string constants. Table 41-3 lists the acceptable values. Hiding or showing table border edges under script control can have an effect on the layout and placement of both the table and surrounding elements. Note that Safari versions 1.3 and earlier do not change the border rendering when you change this property value.

TABLE 41-3

Table frame Property Values

Value	Description
`above`	Top edge only
`below`	Bottom edge only
`border`	All four sides (same as box)
`box`	All four sides (same as border)
`hsides`	Horizontal (top and bottom) edges only
`lhs`	Left-hand side, edge only
`rhs`	Right-hand side, edge only
`void`	No borders
`vsides`	Vertical (left and right) edges only

Example

Listing 41-4 presents a page that cycles through all possible settings for the `frame` property. The `frame` property value is displayed in the table's caption.

LISTING 41-4

Cycling Through Table frame Property Values

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>table.frame Property</title>
      <script type="text/javascript">
         var timeoutID;
         var frameValues = ["box", "above", "rhs", "below", "lhs", "hsides",
             "vsides", "border", "void"];
         function rotateBorder(i)
         {
            document.getElementById("myTABLE").frame = frameValues[i];
            var captionElem = document.getElementById("myCAPTION");
            while(captionElem.firstChild)
            {
               captionElem.removeChild(captionElem.firstChild);
            }
            captionElem.appendChild(document.createTextNode(frameValues[i]));
            i = (++i == frameValues.length) ? 0 : i;
```

continued

LISTING 41-4 *(continued)*

```
            timeoutID = setTimeout("rotateBorder(" + i + ")", 2000);
         }
         function stopRotate()
         {
            clearTimeout(timeoutID);
            document.getElementById("myTABLE").frame = "box";
            var captionElem = document.getElementById("myCAPTION");
            while(captionElem.firstChild)
            {
               captionElem.removeChild(captionElem.firstChild);
            }
            captionElem.appendChild(document.createTextNode("box"));
         }
      </script>
   </head>
   <body>
      <h1>table.frame Property</h1>
      <hr />
      <form name="controls">
         <fieldset>
            <legend>Cycle Table Edge Visibility</legend>
            <table width="100%" cellspacing="20">
               <tr>
                  <td>
                     <input type="button" value="Cycle"
                            onclick="rotateBorder(0)" />
                  </td>
                  <td>
                     <input type="button" value="Stop"
                            onclick="stopRotate()" />
                  </td>
               </tr>
            </table>
         </fieldset>
      </form>
      <hr />
      <table id="myTABLE" cellpadding="5" border="3" align="center">
         <caption id="myCAPTION">
            Default
         </caption>
         <thead id="myTHEAD">
            <tr>
               <th>River</th>
               <th>Outflow</th>
               <th>Miles</th>
               <th>Kilometers</th>
            </tr>
         </thead>
```

```
      <tbody>
         <tr>
            <td>Nile</td>
            <td>Mediterranean</td>
            <td>4160</td>
            <td>6700</td>
         </tr>
         <tr>
            <td>Congo</td>
            <td>Atlantic Ocean</td>
            <td>2900</td>
            <td>4670</td>
         </tr>
         <tr>
            <td>Niger</td>
            <td>Atlantic Ocean</td>
            <td>2600</td>
            <td>4180</td>
         </tr>
         <tr>
            <td>Zambezi</td>
            <td>Indian Ocean</td>
            <td>1700</td>
            <td>2740</td>
         </tr>
      </tbody>
   </table>
  </body>
</html>
```

Related Items: `border`, `borderColor`, `rules` properties

height
width

Value: Integer or length string — Read/Write

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

The `height` (IE4+) and `width` (IE4+/W3C) properties represent the `height` and `width` attributes assigned to the `table` element. If no values are assigned to the element in the tag, the properties do not reveal the rendered size of the table (use the `offsetHeight` and `offsetWidth` properties for that information). Values for these properties can be integers representing pixel dimensions or strings containing percentage values, just like the attribute values. Scripts can shrink the dimensions of a table to no smaller than the minimum space required to render the cell content. Notice that only the `width` property is W3C DOM-sanctioned (as well as the corresponding property in the HTML 4.0 specification).

Example

Use The Evaluator (Chapter 4, "JavaScript Essentials") to adjust the width of the demonstrator table. Begin by increasing the width to the full width of the page:

```
document.getElementById("myTable").width = "100%"
```

To restore the table to its minimum width, assign a very small value to the property:

```
document.getElementById("myTable").width = 50
```

At this point the `width` property will remain 50, even though the table has been sized larger to the minimum size required to accommodate the cell content. To see the actual table width, enter this:

```
document.getElementById("myTable").offsetWidth
```

If you have IE4+, you can perform similar experiments with the `height` and `offsetHeight` properties of the table.

Related Items: `offsetHeight`, `offsetWidth` properties

rows

Value: Array of row objects Read-Only

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

The `rows` property returns an array (collection) of `tr` element objects in the current table. This array includes rows in the `thead`, `tbody`, and `tfoot` row sections, if the table is segmented. You can use the `rows` property to create a cross-browser script that accesses each cell of a table. Such a nested `for` loop looks like the following:

```
var oneCell;
for (var i = 0; i < tableRef.rows.length; i++)
{
   for (var j = 0; j < tableRef.rows[i].cells.length; j++)
   {
      oneCell = tableRef.rows[i].cells[j];
      // more statements working with the cell
   }
}
```

If you want to limit the scope of the `rows` property to rows within a row segment (for example, just in the `tbody`), you can access this property for any of the three types of row segment objects.

Example

Use The Evaluator (Chapter 4, "JavaScript Essentials") to examine the number of rows in the demonstrator table. Enter the following statement into the top text box:

```
document.getElementById("myTable").rows.length
```

In contrast, notice how the `rows` property sees only the rows within the demonstrator table's `tbody` element in IE:

```
document.getElementById("myTbody").rows.length
```

Related Items: `tbody.rows`, `tfoot.rows`, `thead.rows` properties

rules

Value: String constant Read/Write

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

In contrast to the `frame` property, the `rules` property governs the display of borders between cells. Values for this property can be any of a fixed set of string constants. Table 41-4 lists the acceptable values. Hiding or showing table cell border edges under script control can have an effect on the layout and placement of both the table and surrounding elements. Note that WebKit-based browsers and Opera partially implement the rules rendering when you change this property value.

TABLE 41-4

Table rules Property Values

Value	Description
`all`	Borders around every cell
`cols`	Vertical borders between columns
`groups`	Vertical borders between column groups; horizontal borders between row groups
`none`	No borders between cells
`rows`	Horizontal borders between row groups

Example

Listing 41-5 presents a page that cycles through all possible settings for the `rules` property. The `rules` property value is displayed in the table's caption. When you run this script, notice the nice border display for this table's combination of `colgroup` and table row segment elements.

LISTING 41-5

Cycling Through Table rules Property Values

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
```

continued

***tableObject*.rules**

LISTING 41-5 *(continued)*

```
    <title>table.rules Property</title>
    <script type="text/javascript">
       var timeoutID;
       var rulesValues = ["all", "cols", "groups", "none", "rows"];
       function rotateBorder(i)
       {
          document.getElementById("myTABLE").rules = rulesValues[i];
          var captionElem = document.getElementById("myCAPTION");
          while(captionElem.firstChild)
          {
             captionElem.removeChild(captionElem.firstChild);
          }
          captionElem.appendChild(document.createTextNode(rulesValues[i]));
          i = (++i == rulesValues.length) ? 0 : i;
          timeoutID = setTimeout("rotateBorder(" + i + ")", 2000);
       }
       function stopRotate()
       {
          clearTimeout(timeoutID);
          document.getElementById("myTABLE").rules = "all";
          var captionElem = document.getElementById("myCAPTION");
          while(captionElem.firstChild)
          {
             captionElem.removeChild(captionElem.firstChild);
          }
          captionElem.appendChild(document.createTextNode("all"));
       }
    </script>
 </head>
 <body>
    <h1>table.rules Property</h1>
    <hr />
    <form name="controls">
       <fieldset>
          <legend>Cycle Table Rule Visibility</legend>
          <table width="100%" cellspacing="20">
             <tr>
                <td>
                   <input type="button" value="Cycle"
                          onclick="rotateBorder(0)" />
                </td>
                <td>
                   <input type="button" value="Stop"
                          onclick="stopRotate()" />
                </td>
             </tr>
          </table>
       </fieldset>
    </form>
    <hr />
```

```
      <table id="myTABLE" cellpadding="5" border="3" align="center">
         <caption id="myCAPTION">
            Default
         </caption>
         <colgroup span="1">
         </colgroup>
         <colgroup span="3">
         </colgroup>
         <thead id="myTHEAD">
            <tr>
               <th>River</th>
               <th>Outflow</th>
               <th>Miles</th>
               <th>Kilometers</th>
            </tr>
         </thead>
         <tbody>
            <tr>
               <td>Nile</td>
               <td>Mediterranean</td>
               <td>4160</td>
               <td>6700</td>
            </tr>
            <tr>
               <td>Congo</td>
               <td>Atlantic Ocean</td>
               <td>2900</td>
               <td>4670</td>
            </tr>
            <tr>
               <td>Niger</td>
               <td>Atlantic Ocean</td>
               <td>2600</td>
               <td>4180</td>
            </tr>
            <tr>
               <td>Zambezi</td>
               <td>Indian Ocean</td>
               <td>1700</td>
               <td>2740</td>
            </tr>
         </tbody>
      </table>
   </body>
</html>
```

Related Items: `border`, `borderColor`, `frame` properties

`summary`

Value: String Read/Write

Compatibility: WinIE6+, MacIE-, NN6+, Moz+, Safari+, Opera+, Chrome+

The `summary` property represents the HTML 4.0 `summary` attribute. The text assigned to this attribute is intended for use by browsers that present a page's content through nonvisual means. For example, a browser equipped to use speech synthesis to read the page aloud can use the text of the summary to describe the table for the user.

Related Item: `caption` property

tBodies

Value: Array of `tbody` element objects Read-Only

Compatibility: WinIE4+, MacIE5+, NN6+, Moz+, Safari+, Opera+, Chrome+

The `tBodies` property returns an array of all `tbody` elements in the table. Even if you don't specify a `tbody` element, every table contains an implied `tbody` element. Thus, to access a batch of rows, other than the `thead` and `tfoot` sections, of a simple table, you can use the `tBodies[0]` array notation. From there, you can get the rows of the table body section via the `rows` property.

Example

Use The Evaluator (Chapter 4, "JavaScript Essentials") to access the `tBodies` array and reveal the number of rows in the one `tbody` segment of the demonstrator table. Enter the following statement into the top text box:

```
document.getElementById("myTable").tBodies[0].rows.length
```

Related Items: `tFoot`, `tHead` properties

tFoot
tHead

Value: Row segment element object Read/Write (see text)

Compatibility: WinIE4+, MacIE5+, NN6+, Moz+, Safari+, Opera+, Chrome+

Each table can have (at most) one `tfoot` and one `thead` element. If you specify one of these for the table, the `tFoot` and `tHead` properties return references to those element objects, respectively. These properties are read-only in IE, but NN6+/Moz enable you to assign valid `tfoot` and `thead` element objects to these properties in order to insert or replace the elements in the current table. The process for doing this is similar to the sequence described in the `caption` property. For either of these two elements, however, you have to construct the desired number of table cell objects (and row objects, if you want multiple rows) for the newly created row segment object. See the discussions of these two objects for details on accessing rows and cells of the segments.

Related Items: `tbody`, `tfoot`, `thead` objects

width

(See `height`)

Methods

createCaption()
deleteCaption()

Returns: Reference to new `caption` element object; Nothing

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

The `createCaption()` and `deleteCaption()` convenience methods enable you to add or remove a `caption` element object from the current table. When you create a new caption, the action simply inserts the equivalent of a blank `caption` element tag into the `table` element (this may not, however, be reflected in the source view of the page). You must populate the `caption` element with text or HTML before it appears on the page. Because the method returns a reference to the newly created object, you can use that reference to assign content to its `innerHTML` property, or you can append a child text node.

Because a table can have only one `caption` element nested within it, the `deleteCaption()` method belongs to the `table` element object. The method returns no value.

Example

See Listing 41-2 for an example of creating, inserting, and removing a `caption` element object from a table.

Related Item: `caption` element object

createTFoot()
createTHead()
deleteTFoot()
deleteTHead()

Returns: Element references (create methods); Nothing

Compatibility: WinIE4+, MacIE5+, NN6+, Moz+, Safari+, Opera+, Chrome+

These four methods enable you to add or remove `tfoot` and `thead` table row section objects. When you create a `thead` or `tfoot` element, the methods return references to the newly inserted elements. But, as with `createCaption()`, these methods do nothing to display content. Instead, use the returned references to populate the row(s) of the header and footer with cells. Regardless of the number of rows associated with a `thead` or `tfoot` element, the `deleteTFoot()` and `deleteTHead()` methods remove all associated rows and return no values.

Example

See Listing 41-2 for an example of creating, inserting, and removing `tfoot` and `thead` element objects from a table.

Related Items: `tfoot`, `thead` element objects

deleteRow(*rowIndex*)
insertRow(*rowIndex*)

Returns: Nothing; Reference to newly created row

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

The `insertRow()` and `deleteRow()` convenience methods assist in adding `tr` elements to, and removing them from, a `table` element. Inserting a row does little more than the equivalent of inserting a pair of empty `tr` element tags into the HTML (although you may not see them in the source view of the page). It is up to the rest of your scripts to assign properties to the row and populate it with new cells (see the `insertCell()` method of the `tr` element object).

Attributes for both methods are zero-based index numbers. In the case of `insertRow()`, the number indicates the row *before* which the new row is to be inserted. To append the row to the end of the table, use `-1` as a shortcut parameter. To delete a row, use the index value for that row. Be aware that if you intend to employ `deleteRow()` to remove all rows from a table (presumably to repopulate the table with a new set), the most efficient way is to use a `while` loop that continues to remove the first row until there are no more:

```
while (tableRef.rows.length > 0)
{
   tableRef.deleteRow(0);
}
```

Example

See Listing 41-2 for examples of inserting and deleting table rows.

Related Item: `td.insertCell()` method

firstPage()
lastPage()

Returns: Nothing

Compatibility: WinIE5+, MacIE5+, NN-, Moz-, Safari-, Opera-, Chrome-

For tables that are bound to external data sources via IE4+ data binding, the `firstPage()` and `lastPage()` methods zoom to the first and last pages of the data, respectively. You must specify the table's data page size for the Data Source Object to know how many records to assign to a "page" of data. Note that while related methods — `nextPage()` and `previousPage()` — are available in IE4, `firstPage()` and `lastPage()` entered the picture in IE5.

Related Items: `dataPageSize`, `dataSrc`, `dataFld` properties; `nextPage()`, `previousPage()` methods

moveRow(*sourceRowIndex*, *destinationRowIndex*)

Returns: Row element object

Compatibility: WinIE5+, MacIE-, NN-, Moz-, Safari-, Opera-, Chrome-

The IE5+ `moveRow()` convenience method enables you to move a row from one position to another within the same table. Both parameters are integer index values. The first parameter is the index of the row you want to move; the second is the index of the destination row. Because no movement takes place when the method is invoked, the removal of the source row does not impact the index count of the destination row. But after the method executes, the row that was in the destination row is pushed down one row. This method returns a reference to the moved row.

You can (and in most cases should) accomplish this same functionality in W3C DOM-compatible syntax (for IE5+, NN6+, Mozilla, Opera, and WebKit-based browsers) via the `replaceChild()` method of the `table` element.

Example

If you want to shift the bottom row of a table to the top, you can use the shortcut reference to the last item's index value (`-1`) for the first parameter:

```
var movedRow = document.getElementById("someTable").moveRow(-1, 0);
```

Related Item: `replaceChild()` method

nextPage()
previousPage()

Returns: Nothing

Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-, Opera-, Chrome-

For tables that are bound to external data sources via IE4+ data binding, the `nextPage()` and `previousPage()` methods jump ahead or back by one page of data, respectively. You must specify the table's data page size for the Data Source Object to know how many records to assign to a "page" of data. Typically, navigational buttons associated with the table invoke these methods.

Related Items: `dataPageSize`, `dataSrc`, `dataFld` properties; `firstPage()`, `lastPage()` methods

refresh()

Returns: Nothing

Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-, Opera-, Chrome-

For tables that are bound to external data sources via IE4+ data binding, the `refresh()` method retrieves the current data source data for display in the table. A script can use `setTimeout()` to invoke a function that calls this method at an interval of your choice. If you frequently update the database associated with the table, this method can help keep the table up to date, without requiring the client to download the entire page (and perhaps run into cache conflicts).

Related Items: `dataPageSize`, `dataSrc`, `dataFld` properties

tbody, tfoot, and thead Element Objects

For HTML element properties, methods, and event handlers, see Chapter 26, “Generic HTML Element Objects.”

Properties	Methods	Event Handlers
align†	deleteRow()†	
bgColor†	insertRow()†	
ch	moveRow()†	
chOff		
rows†		
vAlign		

†See table element object.

Syntax

Accessing tbody, tfoot, and thead element object properties and methods:

```
(IE4+)     [window.]document.all.elemID.property | method([parameters])
(IE5+/W3C) [window.]document.getElementById("elemID").property |
           method([parameters])
```

Accessing tbody element object properties and methods:

```
(IE4+)     [window.]document.all.tableID.tBodies[i].property |
           method([parameters])
(IE5+/W3C) [window.]document.getElementById("tableID").tBodies[i].property |
           method([parameters])
```

Accessing tfoot element object properties and methods:

```
(IE4+)     [window.]document.all.tableID.tFoot.property | method([parameters])
(IE5+/W3C) [window.]document.getElementById("tableID").tFoot.property |
           method([parameters])
```

Accessing thead element object properties and methods:

```
(IE4+)     [window.]document.all.tableID.tHead.property | method([parameters])
(IE5+/W3C) [window.]document.getElementById("tableID").tHead.property |
           method([parameters])
```

Compatibility: WinIE4+, MacIE5, NN6+, Moz+, Safari+, Opera+, Chrome+

About these objects

Each of these element objects represents a row grouping within a `table` element (an `HTMLTableSectionElement` in the syntax of the W3C DOM specification). A table can have only one `thead` and one `tfoot`, but it can have as many `tbody` elements as your table organization requires.

These elements share many properties and methods with the `table` element in that they all contain rows. The benefit of defining table segments is apparent if you use table rules (see the `table.rules` property earlier in this chapter) and if you wish to limit the scope of row activities only to rows within one segment. For instance, if your table has a `thead` that is to remain static, your scripts can merrily loop through the rows of only the `tbody` section without coming anywhere near the row(s) in the `thead`.

Properties

ch
chOff

Value: One-character string Read/Write

Compatibility: WinIE6+, MacIE5+, NN6+, Moz+, Safari+, Opera+, Chrome+

The `ch` and `chOff` properties represent the optional `char` and `charoff` attributes of table row section elements in the HTML 4.0 specification. These properties help align cell content within a column or column group, similar to the way word processors allow for formatting features such as decimal tabs; they are yet to be implemented in IE as of version 7. For details on these attributes, see `http://www.w3.org/tr/REC-html40/struct/tables.html#adef-char`.

Related Items: `col`, `colgroup` objects

vAlign

Value: String constant Read/Write

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

Providing the cell-oriented `vAlign` property for a table row section enables you to specify a vertical alignment to apply to all cells within that section rather than having to specify the `valign` attribute for each `td` element. By default, browsers render cell content with a middle vertical alignment within the cell. If you want to modify the setting for an existing table section (or assign the setting to a new one you create), the values must be one of the following string constants: `baseline`, `bottom`, `middle`, or `top`.

Example

Use The Evaluator (Chapter 4, "JavaScript Essentials") to modify the vertical alignment of the content of the `tbody` element in the demonstrator table. Enter the following statement in the top text box to shift the content to the bottom of the cells:

```
document.getElementById("myTBODY").vAlign = "bottom"
```

Notice that the cells of the thead element are untouched by the action imposed on the tbody element.

Related Item: td.vAlign property

caption Element Object

For HTML element properties, methods, and event handlers, see Chapter 26, "Generic HTML Element Objects."

Properties	Methods	Event Handlers
align†		
vAlign††		

†See table element object.

††See tbody element object.

Syntax

Accessing caption element object properties and methods:

```
(IE4+)      [window.]document.all.elemID.property | method([parameters])
(IE5+/W3C) [window.]document.getElementById("elemID").property |
            method([parameters])
```

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

About this object

A caption element is a simple HTML container whose only prerequisite is that it must be nested inside a table element. That nesting allows the table element object to control insertion and removal of a caption element at will. You can modify the content of a caption element just like you do any HTML element (in DOMs that allow such modification). You can see an example of how the table element object uses some of its methods to create and remove a caption element in Listing 41-2.

The only properties that lift the caption element object above a mere contextual element (as described in Chapter 26) are vAlign (IE4+) and the W3C DOM-sanctioned align (IE4+/NN6+/Mozilla/Opera and WebKit-based browsers). We describe these properties, and their values for other objects, in this chapter.

col and colgroup Element Objects

For HTML element properties, methods, and event handlers, see Chapter 26, "Generic HTML Element Objects."

Properties	Methods	Event Handlers
align†		
ch††		
chOff††		
span		
vAlign††		
width		

†See table element object.

††See tbody element object.

Syntax

Accessing col and colgroup element object properties and methods:

```
(IE4+)     [window.]document.all.elemID.property | method([parameters])
(IE5+/W3C) [window.]document.getElementById("elemID").property |
           method([parameters])
```

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

About these objects

The purpose of the col and colgroup elements is to allow cells within one or more columns to be treated as a single entity for purposes of style sheet and other style-related control. In other words, if you want one column of a table to be all boldface, you can assign that style sheet rule to the col element that encompasses that column. All cells within that column inherit the style sheet rule definition. Having two different element names allows for the nesting of column groups, which can come in handy for complex tables. For instance, consider a table that reports the forecasted and actual sales for a list of products across four quarters of a year. The left column of the table stands alone with the product item numbers. To the right is one large grouping of eight columns that encompasses the four pairs of forecasted/actual sales. All eight columns of cells are to be formatted with a particular font style to help differentiate the pairs of columns for

each quarter. You also want to assign a different background color. Therefore, you designate each pair of columns as its own subgroup within the eight-column master grouping. The `colgroup` and `col` tags for this nine-column table are as follows:

```
<col id="productIDs">
<colgroup id="fiscalYear" span="8" width="40">
   <col id="Q1" span="2">
   <col id="Q2" span="2">
   <col id="Q3" span="2">
   <col id="Q4" span="2">
</colgroup>
```

Up in the `head` section of this document are style sheet rules similar to the following:

```
<style type="text/css">
   #productIDs {font-weight:bold}
   #fiscalYear {font-family: Courier, "Courier New", monospace}
   #Q1 {background-color: lightyellow}
   #Q2 {background-color: pink}
   #Q3 {background-color: lightblue}
   #Q4 {background-color: lightgreen}
</style>
```

The HTML code for the column groups demonstrates the two key attributes: `span` and `width`. Both of these attributes are reflected as properties of the objects, and we describe them in the following section. Notice, however, that `col` and `colgroup` elements act cumulatively, and in source code order, to define the column groups for the table. In other words, if the style of the left-hand column is not important, the table still requires the initial one-column `col` element before the eight-column `colgroup` element. Otherwise, the browser makes the first eight columns the column group. Therefore, it is a good idea to account for every column with `col` and/or `colgroup` elements if you intend to use any column grouping in your table.

From a scripter's point of view, you are more likely to modify styles in an existing table for a column or column group than you are to alter properties such as `span` or `width`. But, if your scripts generate new tables, you may create new `col` or `colgroup` elements whose properties you then initialize.

Properties

span

Value: Integer Read/Write

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

The `span` property represents the number of columns that the column group should encompass. Don't confuse this property with the `colSpan` property of `td` and `th` elements. A `col` or `colgroup span` does not have any impact on the rendering or combination of multiple cells

into one. It simply draws an imaginary lasso around as many columns as are specified, signifying that these columns can be treated as a group for style purposes (and also for drawing divider rules, if you set the table's `rules` property to `groups`).

Example

The following statement assigns a `span` of 3 to a newly created `colgroup` element stored in the variable `colGroupA`:

```
colGroupA.span = 3;
```

Related Item: `width` property

width

Value: Length string Read/Write

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

The only reason the `width` property is highlighted for these objects is that the property (and corresponding attribute) impacts the width of table cells inside the scope of the column grouping. For example, if you assign a width of 50 pixels to a `colgroup` whose `span` attribute is set to 3, all cells in all three columns inherit the 50-pixel width specification. For more details on the values acceptable to this property, see the `table.width` property description earlier in this chapter.

Related Item: `table.width` property

tr Element Object

For HTML element properties, methods, and event handlers, see Chapter 26, "Generic HTML Element Objects."

Properties	Methods	Event Handlers
align†	deleteCell()	
bgColor†	insertCell()	
borderColor†		
borderColorDark†		
borderColorLight†		
cells		
ch††		
chOff††		

Properties	Methods	Event Handlers
height		
rowIndex		
sectionRowIndex		
vAlign††		

†See table element object.

††See tbody element object.

Syntax

Accessing tr element object properties and methods:

```
(IE4+)     [window.]document.all.elemID.property | method([parameters])
(IE4+)     [window.]document.all.tableID.rows[i].property | method
           ([parameters])
(IE4+)     [window.]document.all.tableRowSectionID.rows[i].property |
           method([parameters])
(IE5+/W3C) [window.]document.getElementById("elemID"). property |
           method([parameters])
(IE5+/W3C) [window.]document.getElementById("tableID").rows[i].property |
           method([parameters])
(IE5+/W3C) [window.]document.getElementById("tableRowSectionID").rows
           [i].property | method([parameters])
```

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

About this object

Table rows are important objects within the complex nesting of table-related elements and objects. When a table represents server database data, one row usually equals one record. And, although you can employ scripting to add columns to a table, the more common table modifications are to add or delete rows — hence the presence of the table element object's insertRow() and deleteRow() methods.

The primary job of the tr element is to act as a container for td elements. All the cells in a row inherit some attributes and properties that you apply to that row. An array of cell objects is available for iteration via for loops. A tr element object, therefore, also has methods that insert and remove individual cells in that row.

The number of columns in a row is determined by the number of td elements or, more specifically, by the number of columns that the cells intend to span. One row can have four td elements, whereas the next row can have only two td elements — each of which is defined to occupy two columns. The row of the table with the most td elements and column reservations determines the column width for the entire table.

Of the properties just listed, the ones related to border color are available in IE4+ only. In IE4+, the border is drawn around each cell of the row rather than the entire row. The HTML 4.0 specification (and, by extension, the W3C DOM Level 2 specification) does not recognize border colors for rows alone, nor are style sheet border rules inherited by the cell children of a row. However, you can define borders for individual cells or classes of cells.

Properties

`cells`

Value: Array of `td` element objects Read-Only

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

The `cells` property returns an array (collection) of `td` element objects nested inside the current `tr` object. The `length` property of this array indicates the number of actual `td` elements in the row, which may not be the number of columns if one or more cells occupy multiple columns.

Use the `cells` property in `for` loops to iterate through all cells within a row. Assuming your script has a reference to a single row, the loop should look like the following:

```
for (var i = 0; i < rowRef.cells.length; i++)
{
   oneCell = rowRef.cells[i];
   // more statements working with the cell
}
```

Example

Use The Evaluator (Chapter 4, "JavaScript Essentials") to retrieve the number of `td` elements in the second row of the demonstrator table. Enter the following statement into the top text box (W3C DOM syntax shown here):

```
document.getElementById("myTable").rows[1].cells.length
```

Related Items: `table.rows`, `td.cellIndex` properties

`height`

Value: Integer or length string Read/Write

Compatibility: WinIE5+, MacIE5+, NN-, Moz-, Safari-, Opera+, Chrome-

IE5+ and Opera enable page authors to predefine a height for a table row; this attribute is echoed by the `height` property. The value can be a number of pixels or a percentage length value. Note that this property does not reveal the rendered height of the row unless you explicitly set the attribute in the HTML. To get the actual height (in IE5+ and Mozilla), use the `offsetHeight` property. You cannot adjust the `height` property to be smaller than the table normally renders the row.

Example

Use The Evaluator (Chapter 4, "JavaScript Essentials") in IE5+ to expand the height of the second row of the demonstrator table. Enter the following statement into the top text box:

```
document.getElementById("myTable").rows[1].height="300"
```

If you attempt to set the value very low, the rendered height goes no smaller than the default height.

Related Item: `offsetHeight` property (Chapter 26, "Generic HTML Element Objects")

rowIndex
sectionRowIndex

Value: Integer — Read-Only

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

Each row occupies a position within the collection of rows in the table, as well as within the collection of rows for a table section (`thead`, `tbody`, or `tfoot`). The `rowIndex` property returns the zero-based index value of the row inside the `rows` collection for the entire table, regardless of table section composition. In contrast, the `sectionRowIndex` property returns the zero-based index value of the row inside its row section container. If the table has no row sections defined for it, a single, all-encompassing `tbody` element is assumed; in this case, the `sectionRowIndex` and `rowIndex` values are equal.

These properties serve in functions that are passed a reference to a row. However, the functions might also need to know the position of the row within the table or section. Although there is no `tr` object property that returns a reference to the next outermost table row section, or to the table itself, the `parent` and parent's `parent` elements, respectively, can reference these objects.

Example

Use The Evaluator (Chapter 4, "JavaScript Essentials") to explore the `rowIndex` and `sectionRowIndex` property values for the second physical row in the demonstrator table. Enter each of the following statements into the top text box:

```
document.getElementById("myTable").rows[1].rowIndex
document.getElementById("myTable").rows[1].sectionRowIndex
```

The result of the first statement is `1` because the second row is the second row of the entire table. But the `sectionRowIndex` property returns `0` because this row is the first row of the `tbody` element in this particular table.

Related Items: table.rows, tbody.rows, tfoot.rows, thead.rows properties

Methods

deleteCell(*cellIndex*)
insertCell(*cellIndex*)

Returns: Nothing; Reference to new cell

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

The act of inserting a row into a table is not complete until you also insert cells into the row. The `insertCell()` method does just that, with a parameter indicating the zero-based index of the cell's position among other cells in the row. A value of `-1` appends the cell to the end of existing cells in the row.

When you invoke the `insertCell()` method, it returns a reference to the new cell. This gives you the opportunity to adjust other properties of that cell before moving onto the next cell. For example, if you want to insert a cell that has a column span of `2`, you adjust the `colSpan` property of the cell whose reference just returned, as in the following:

```
var oneCell = tableRowRef.insertCell(-1);
oneCell.colSpan = 2;
```

Scripts that add rows and cells must make sure that they add the identical number of cells (or cell column spaces) from one row to the next. Otherwise, you have an unbalanced table with ugly blank spaces where you probably don't want them.

To remove a cell from a row, use the `deleteCell()` method. The parameter is a zero-based index value of the cell you want to remove. To rid yourself of all cells in a row, use the `deleteRow()` method of the `table` and table row section element objects.

Example

See Listing 41-2 for an example of inserting cells during the row insertion process.

Related Item: `table.insertRow()` method

td and th Element Objects

For HTML element properties, methods, and event handlers, see Chapter 26, "Generic HTML Element Objects."

Properties	Methods	Event Handlers
`abbr`		
`align`†		
`axis`		
`background`†		
`bgColor`†		

tdObject

Properties	Methods	Event Handlers
borderColor†		
borderColorDark†		
borderColorLight†		
cellIndex		
ch††		
chOff††		
colSpan		
headers		
height		
noWrap		
rowSpan		
vAlign††		
width		

†See `table` element object.

††See `tbody` element object.

Syntax

Accessing `td` and `th` element object properties and methods:

```
(IE4+)      [window.]document.all.elemID.property | method([parameters])
(IE4+)      [window.]document.all.tableID.cells[i].property |
            method([parameters])
(IE4+)      [window.]document.all.tableRowSectionID.cells[i].property |
            method([parameters])
(IE4+)      [window.]document.all.tableRowID.cells[i].property |
            method([parameters])
(IE5+/W3C) [window.]document.getElementById("elemID"). property |
            method([parameters])
(IE5+/W3C) [window.]document.getElementById("tableID").cells[i].property |
            method([parameters])
(IE5+/W3C) [window.]document.getElementById("tableRowSectionID").cells[i].
            property | method([parameters])
(IE5+/W3C) [window.]document.getElementById("tableRowID").rows[i].property |
            method([parameters])
```

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

About these objects

`td` (table data) and `th` (table header) elements create cells within a table. By common convention, a `th` element is rendered in today's browsers with a distinctive style — usually with a bold font and center alignment. A table cell is as deeply nested as you can get with table-related elements.

Properties of cells that are delivered in the HTML of the page are rarely modified (with the exception of the `innerHTML` property). But you still need full access to properties of cells if your scripts add rows to a table dynamically. After creating each blank table cell object, your scripts can adjust `colSpan`, `rowSpan`, `noWrap`, and other properties that influence the characteristics of that cell within the table.

See the beginning of this chapter for discussions and examples of how to add rows of cells and modify cell content under script control.

Properties

abbr
axis
headers

Value: See text Read/Write

Compatibility: WinIE6+, MacIE5+, NN6+, Moz+, Safari+, Opera+, Chrome+

These three properties are defined for table cell element objects in the W3C DOM. They all represent attributes for these elements in the HTML 4.0 specification. The purposes of these attributes and properties are geared toward browsers that provide alternate means of rendering content, such as through speech synthesis. Although these properties are definitely valid for W3C browsers, they have no practical effect. For general application, you can ignore these properties — consider them reserved for future use.

cellIndex

Value: Integer Read-Only

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

The `cellIndex` property returns an integer indicating the zero-based count of the current cell within its row. Thus, if a script is passed a reference to a cell, the `cellIndex` property reveals its position within the row. Inserting or deleting cells in the row at lower index values influences the `cellIndex` value after the alteration.

Example

You can rewrite the cell addition portion of Listing 41-2 to utilize the `cellIndex` property. The process entails modifying the `insertTableRow()` function so that it uses a `do...while` construction to keep adding cells to match the number of data slots. The function looks like the following (changes shown in boldface):

```
function insertTableRow(form, where)
{
   var now = new Date();
   var nowData = [now.getHours(), now.getMinutes(), now.getSeconds(),
      now.getMilliseconds()];
   clearBGColors();
```

```
    var newCell;
    var newRow = theTableBody.insertRow(where);
    var i = 0;
    do
    {
       newCell = newRow.insertCell(i);
       while(newCell.firstChild)
       {
          newCell.removeChild(newCell.firstChild);
       }
       newCell.appendChild(document.createTextNode(nowData[i++]));
       newCell.style.backgroundColor = "salmon";
    } while (newCell.cellIndex < nowData.length)
    updateRowCounters(form);
}
```

This version is merely for demonstration purposes and is not as efficient as the sequence shown in Listing 41-2. But the `cellIndex` property version can give you some implementation ideas for the property. It also shows how dynamic the property is, even for brand new cells.

Related Item: `tr.rowIndex` property

colSpan
rowSpan

Value: Integer Read/Write

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

The `colSpan` and `rowSpan` properties represent the `colspan` and `rowspan` attributes of table cell elements. Assign values to these properties only when you are creating new table rows and cells — and you are firm in your table cell design. If you fail to assign the correct values to either of these properties, your table cell alignment will get out of whack. Modifying these property values on an existing table is extremely risky unless you are performing other cell manipulation to maintain the balance of rows and columns. Values for both properties are integers greater than or equal to 1.

Example

Use The Evaluator (Chapter 4, "JavaScript Essentials") to witness how modifying either of these properties in an existing table can destroy the table. Enter the following statement into the top text box:

```
document.getElementById("myTable").rows[1].cells[0].colSpan = 3
```

Now that the first cell of the second row occupies the space of three columns, the browser has no choice but to shift the two other defined cells for that row out beyond the original boundary of the table. Experiment with the `rowSpan` property the same way. To restore the original settings, assign 1 to each property.

Related Item: `col.span` property

height
width

Value: Integer and length string Read/Write

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

Table cells may be specified to be larger than their default rendered size. This usually happens in the `height` and `width` attributes of the cell. Settings of the `width` attribute of a `col` or `colgroup` element may also govern the width of a cell. A cell's height can be inherited from the `height` attribute setting of a table row or row section. Both `height` and `width` attributes are deprecated in HTML 4.0 in favor of the `height` and `width` style sheet attributes. That said, the `height` and `width` properties of a table cell echo only the settings of the explicit attributes in the cell's tag. If a style sheet in the element tag governs a cell's dimensions, visit the cell object's `style` property to determine the dimensions. Explicit attributes override style sheet rules.

Values for these two properties are length values. These can be pixel integers or percentage values as strings. Attempts to set the sizes smaller than their default rendered sizes results in a cell of default size. Also be aware that enlarging a cell affects the width of the entire column and/or the height of the entire row occupied by that cell.

Example

Use The Evaluator (Chapter 4, "JavaScript Essentials") to see the results of setting the `height` and `width` properties of an existing table cell. Enter each of the following statements into the top text box and study the results in the demonstrator table (W3C DOM syntax used here):

```
document.getElementById("myTable").rows[1].cells[1].height = 100
document.getElementById("myTable").rows[2].cells[0].width = 300
```

You can restore both cells to their original sizes by assigning very small values, such as `1` or `0`, to the properties. The browser prevents the cells from rendering any smaller than is necessary to show the content.

Related Items: `col.width`, `tr.height` properties

noWrap

Value: Boolean Read/Write

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

The default behavior of a table cell is to wrap text lines within the cell if the text would extend beyond the right edge of the cell, as calculated from the width of the entire table. But you can force the table to be wider to accommodate an unwrapped line of text by setting the `noWrap` property (or `nowrap` attribute) of the cell to `true`. The `nowrap` attribute is deprecated in HTML 4.0.

Example

The following statement creates a new cell in a row and sets its `noWrap` property to prevent text from word-wrapping inside the cell:

```
newCell = newRow.insertCell(-1);
newCell.noWrap = true;
```

You need to set this property only if the cell must behave differently than the default, word-wrapping style.

rowSpan

(See `colSpan`)

width

(See `height`)

ol Element Object

For HTML element properties, methods, and event handlers, see Chapter 26, "Generic HTML Element Objects."

Properties	Methods	Event Handlers
`start`		
`type`		

Syntax

Accessing `ol` element object properties and methods:

```
(IE4+)     [window.]document.all.elemID.property | method([parameters])
(IE5+/W3C) [window.]document.getElementById("elemID"). property |
           method([parameters])
```

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

About this object

The `ol` (ordered list) element is a container of `li` (list item) elements. An *ordered list* means that the list items have a sequence and are preceded by a number or letter to signify the

position within the sequence. The few element-specific attributes are being deprecated in favor of style sheet definitions. For the sake of backward compatibility with existing content, however, it is likely that many future generations of browsers will continue to support these deprecated attributes. These attributes are therefore available as properties of the element object.

Most of the special appearance of a list (notably indentation) is handled automatically by the browser's interpretation of how an ordered list should look. You have control over the numbering or lettering schemes and the starting point for those sequences. With CSS you can significantly override the browser's formatting — even eliminating the traditional list appearance via other choices for the display `style` property.

Properties

start

Value: Integer Read/Write

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

The `start` property governs which number or letter begins the sequence of leading characters for nested `li` items. If the `type` attribute specifies numbers, the corresponding number is used; if it specifies letters, the letter of the alphabet corresponding to the number becomes the starting character. You can change the numbering in the middle of a sequence via the `li.value` property.

It is an extremely rare case that requires you to modify this property for an existing `ol` element. But if your script is creating a new element for a segment of ordered list items that has some other content intervening from an earlier `ol` element, you can use the property to assign a starting value to the `ol` group.

Example

The following statements generate a new `ol` element and assign a value to the `start` property:

```
var newOL = document.createElement("ol");
newOL.start = 5;
```

Related Items: `type`, `li.value` properties

type

Value: String constant Read/Write

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

***olObject*.type**

An ol element can use any of five different numbering schemes. Each scheme has a type code, whose value you can use for the type property. The following table shows the property values and examples:

Value	Example
A	A, B, C, ...
a	a, b, c, ...
I	I, II, III, ...
i	i, ii, iii, ...
1	1, 2, 3, ...

The default value is 1. You are free to adjust the property after the table has rendered, and you can even stipulate a different type for specific li elements nested inside (see the li.type property). If you want to have further nesting with a different numbering scheme, you can nest the ol elements and specify the desired type for each nesting level, as shown in the following HTML example:

```
<ol type="A">
   <li>One</li>
   <li>Two</li>
   <li>Three</li>
      <ol type="a">
         <li>Sub One</li>
         <li>Sub Two</li>
         <li>Sub Three</li>
      </ol>
   <li>Four</li>
</ol>
```

Indenting the HTML is optional, but it may help you to keep the nesting straight.

Example

The following statements generate a new ol element and assign a value to the type property so that the sequence letters are uppercase Roman numerals:

```
var newOL = document.createElement("ol");
newOL.type = "I";
```

Related Items: start, UL.type, LI.type properties

ul Element Object

For HTML element properties, methods, and event handlers, see Chapter 26, "Generic HTML Element Objects."

Properties	Methods	Event Handlers
`type`		

Syntax

Accessing `ul` element object properties and methods:

```
(IE4+)     [window.]document.all.elemID.property | method([parameters])
(IE5+/W3C) [window.]document.getElementById("elemID").property |
           method([parameters])
```

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

About this object

The `ul` (unordered list) element is a container of `li` (list item) elements. An *unordered list* means that the list items have no sequence and are preceded by symbols that don't signify any particular order. The few element-specific attributes are being deprecated in favor of style sheet definitions. For the sake of backward compatibility with existing content, however, it is likely that many future generations of browsers will continue to support these deprecated attributes. These attributes are therefore available as properties of the element object.

Most of the special appearance of a list (notably indentation) is handled automatically by the browser's interpretation of how an ordered list should look. You have control over the three possible characters that precede each item. With CSS you can significantly override the browser's formatting — even eliminating the traditional list appearance via other choices for the display `style` property.

Properties

`type`

Value: String constant   Read/Write

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

A ul element can use any of three different leading characters. Each leading character has a type code whose value you can employ for the type property. Property values are circle, disc, and square. The difference between a circle and disc is that the circle is unfilled, whereas the disc is solid. The default value is disc.

Example

The following statements generate a new ul element and assign a value to the type property so that the bullet characters are empty circles:

```
var newUL = document.createElement("ul");
newUL.type = "circle";
```

Related Items: OL.type, UL.type properties

li Element Object

For HTML element properties, methods, and event handlers, see Chapter 26, "Generic HTML Element Objects."

Properties	Methods	Event Handlers
type		
value		

Syntax

Accessing li element object properties and methods:

```
(IE4+)     [window.]document.all.elemID.property | method([parameters])
(IE5+/W3C) [window.]document.getElementById("elemID").property |
           method([parameters])
```

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

About this object

An li (list item) element contains the HTML that is displayed for each item within an ol or ul list. Note that you can put any HTML you want inside a list item, including images. Attributes and properties of this element enable you to override the specifications declared in the ol or ul containers (except in MacIE).

Properties

type

Value: String constant — Read/Write

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

Because either an `ol` or `ul` container can own an `li` element, the `type` property accepts any of the values that you assign to the `type` properties of both the `ol` and `ul` element objects. See the `ol.type` and `ul.type` properties earlier in this chapter for lists of those values.

Exercise caution, however, if you attempt to mix and match types. For example, if you try to set the `li.type` property of an `li` element to `circle` inside an `ol` element, the results vary from browser to browser. NN6+/Moz, for example, follows your command; however, IE may display some other characters.

Example

See the examples for the `ol.type` and `ul.type` properties earlier in this chapter.

Related Items: `ol.type`, `ul.type` properties

value

Value: Integer — Read/Write

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

The `value` property governs which number or letter is used for the current list item inside an ordered list. Employ this attribute and property to override the natural progression. Because these sequence characters can be letters, numbers, or Roman numerals, the integer you specify for this property is converted to the numbering scheme dictated by the `li` or `ol` element's `type` property.

Example

The following statements generate a new `li` element and assign a value to the `start` property:

```
var newLI = document.createElement("li");
newLI.start = 5;
```

Related Item: `ol.start` property

dl, dt, and dd Element Objects

For HTML element properties, methods, and event handlers, see Chapter 26, "Generic HTML Element Objects."

dlObject.compact

Properties	Methods	Event Handlers
compact		

Syntax

Accessing dl, dt, and dd element object properties and methods:

```
(IE4+)        [window.] document.all.elemID.property | method([parameters])
(IE5+/W3C)    [window.] document.getElementById("elemID"). property |
              method([parameters])
```

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

About these objects

Three elements — dl, dt, and dd — provide context and (optionally) formatting for definitions in a document. The dl element is the outer wrapper signifying a definition list. Each definition term should be inside a dt element, whereas the definition description should be in the nested dd element. The HTML for a simple definition list has the following structure:

```
<dl>
   <dt>First term</dt>
      <dd>First term's definition</dd>
   <dt>Second  term</dt>
      <dd>Second  term's definition</dd>
</dl>
```

Although there are no specific requirements for rendering definition lists, by convention, the term and description are usually on different lines, with the description indented. With CSS you can significantly override the browser's formatting — even eliminating the traditional list appearance via other choices for the display style property.

All three of these elements are treated as element objects, sharing the same properties, methods, and event handlers of generic element objects. The only one of the three that has anything special is the dl element, which has a compact property. WinIE4+ does respond to this attribute and property by putting the description and term on the same line if the term is shorter than the usual indentation space of the description.

Properties

compact

Value: Boolean Read/Write

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

Although the property is defined for the browsers just shown, the compact property (and the deprecated attribute it echoes) only has an impact on the dl element in WinIE browsers.

dir and menu Element Objects

For HTML element properties, methods, and event handlers, see Chapter 26, "Generic HTML Element Objects."

Syntax

Accessing `dir` and `menu` element object properties and methods:

```
(IE4+)     [window.]document.all.elemID.property | method([parameters])
(IE5+/W3C) [window.]document.getElementById("elemID"). property |
           method([parameters])
```

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

About these objects

The `dir` and `menu` elements are treated in modern browsers as if they were `ul` elements for unordered lists of items. Both elements are deprecated in HTML 4.0; yet, because they are acknowledged in that standard, they are also acknowledged in the W3C DOM (and the IE DOM, too). Originally intended to assist in creating single and double columns of text (long since supplanted by tables), usage of these elements has fallen out of favor and is discouraged.

CHAPTER

The Navigator and Other Environment Objects

IN THIS CHAPTER

Determining which browser the user has

Branching scripts according to the user's operating system

Detecting plug-in support

Client-side scripting primarily focuses on the document inside a browser window and the content of the document. As discussed in Chapter 27, "Window and Frame Objects," the window, too, is an important part of how you apply JavaScript on the client. Stepping out one more level, we get to the browser application itself. Scripts sometimes need to know about the browser and the computing environment in which it runs so that they can tailor dynamic content for the current browser and operating system.

To that end, browsers provide objects that expose as much about the client computer and browser as is feasible within accepted principles of preserving a user's privacy. In addition to providing some of the same information that server-side programs receive as environment variables, these browser-level objects may also include information about how well equipped the browser is with regard to plug-ins and Java. Another object reveals information about the user's video monitor, which may influence the way your scripts calculate information displayed on the page.

The objects in this chapter don't show up on the document object hierarchy diagrams, except as free-standing groups. The IE4+ object model, however, incorporates these environmental objects as properties of the `window` object. Because the `window` reference is optional, in many cases you can omit it and wind up with a cross-browser compatible script.

Although these objects were first implemented outside of the object model hierarchy, they are now treated as belonging to the `window` object. As you learn in this chapter, the IE for Windows methodology can be a bit roundabout. And yet the Macintosh version of IE5 adopted the approach initiated by NN3. Go figure!

clientInformation Object (IE4+) and navigator Object (All)

Properties	Methods	Event Handlers
appCodeName	javaEnabled()	
appMinorVersion	preference()	
appName		
appVersion		
browserLanguage		
cookieEnabled		
cpuClass		
language		
mimeTypes		
online		
oscpu		
platform		
plugins		
product		
productSub		
securityPolicy		
systemLanguage		
userAgent		
userLanguage		
userProfile		
vendor		
vendorSub		

Syntax

Accessing `clientInformation` and `navigator` object properties and methods:

```
(All)        navigator.property | method()
(IE4+/Moz)   [window.]navigator.property | method()
(IE4+/Moz)   [window.]clientInformation.property | method()
```

About this object

In Chapter 27, we repeatedly mention that the `window` object is the top banana of the document object hierarchy. In other programming environments, you likely can find a level higher than the window — perhaps referred to as the *application level*. You may think that an object known as the `navigator` object is that all-encompassing object in JavaScript. That is not the case, however.

Although Netscape originally invented the `navigator` object for the Navigator 2 browser, Microsoft Internet Explorer also supports this object in its object model. For those who have partisan feelings in favor of Microsoft, IE4+ provides an alternate object — `clientInformation` — that acts as an alias to the `navigator` object. You are free to use the IE-specific terminology if your development is intended only for IE browsers, but there is really no good reason not to use `navigator` in case you should ever need to support browsers beyond IE. All properties and methods of the `navigator` and `clientInformation` objects are identical. In the rest of this section, all references to the `navigator` object also apply to the `clientInformation` object.

Be aware that the number of properties for this object has grown with virtually every browser version. Moreover, other than some basic items that have been around since the early days, most of the more recent properties are browser-specific. Observe the compatibility ratings for each of the following properties very carefully.

Most of the properties of the `navigator` object deal with the browser program that the user runs to view documents. Properties include those for extracting the versions of the browser and of the platform of the client running the browser. Because so many properties of the `navigator` object are related to one another, we begin this discussion by grouping four of the most popular ones together.

Properties

appCodeName
appName
appVersion
userAgent

Value: String Read-Only

Compatibility: WinIE3+, MacIE3+, NN2+, Moz+, Safari+, Opera+, Chrome+

These four properties reveal just about everything that browser-sniffing code needs to know about the user's browser brand, version, and other tidbits. Of these four, only the last three are particularly valuable. The first property in the list, `appCodeName`, defines a class of client that encompasses essentially every standard browser. The value returned, `Mozilla`, is the code name of the first browser engine, on which NN and IE browsers at one time were based (the NCSA Mosaic browser). This information does nothing to help your scripts distinguish among browser flavors, so you can ignore the property. The other three properties are the ones with all the goodies.

The appName property returns the official name for the browser application. For Netscape and Mozilla browsers, the appName value is Netscape; for Internet Explorer, the value is Microsoft Internet Explorer. The situation is murkier for other browsers. For example, Opera gives users a preference option to have the browser identify itself as IE, NN, or Opera — and the appName property value changes accordingly. And Safari identifies itself as Netscape in the appName property. Thus, the appName property is no longer a reliable browser brand determinant.

The appVersion and userAgent properties provide more meaningful detail. We start with the appVersion property because it is revealing and, at times, misleading.

Using the appVersion property

A typical appVersion property value looks like the following (one is from FF1.5-3.5.6 running on Windows, the other from IE8):

```
5.0 (Windows; en-US)
4.0 (compatible; MSIE 8.0; Windows NT 5.1; Trident/4.0; .NET CLR 2.0.50727;
    .NET CLR 3.0.04506.30; .NET CLR 3.0.04506.648)
```

Because most version decisions are based on numeric comparisons (for example, the version is equal to or greater than 4), you frequently need to extract just the number part of the string returned by the appVersion property. The cleanest way to do this is through the parseInt() or parseFloat() methods. Use

```
parseInt(navigator.appVersion)
```

if you are interested only in the number to the left of the decimal; to get the complete leading floating-point number, use

```
parseFloat(navigator.appVersion)
```

All other characters after the leading numbers are ignored.

Also notice that the number does not always accurately represent the version of the browser at hand. For example, IE6 through IE8 report that they are version 4.0. The number is more indicative of a broad generation number than a specific browser version number. In other words, the browser exhibits characteristics of the first browsers to wear the appVersion of 4 (IE 4.0, it turns out). Although this means that IE8 can use everything that is in the language and object model of IE4, it obviously doesn't help your script to know if the browser is capable of IE8 scripting features.

At the same time, however, buried elsewhere in the appVersion string, is the wording MSIE 8.0 — the "true" version of the browser. IE uses this technique to distinguish the actual version number from the generational number. Therefore, for IE, you may have to dig deeper by using string methods such as indexOf() to see if the appVersion contains the desired string. For example, to see if the browser is a variant of IE8, you can test for just "MSIE 8" as follows:

```
var isIE8x = navigator.appVersion.indexOf("MSIE 8") != -1
```

***navigatorObject*.appVersion**

This kind of testing is not without risk, however. Going forward, your code will break if future versions of IE have larger version numbers. Therefore, if you want to use IE7 features with an IE8 browser, your testing for the presence of `"MSIE 7"` fails, and the script thinks that it cannot use IE7 features even though they most certainly would be available in IE8. To find out if the current IE browser is the same or newer than a particular version, you must use JavaScript string parsing to deal with the `MSIE x.x` substring of the `appVersion` (or `userAgent`) property. The following example shows one function that extracts the precise IE version name and another function that confirms whether the version is at least IE7.0 for Windows.

```
var ua = navigator.userAgent;
function getIEVersion()
{
   var IEOffset = ua.indexOf("MSIE ");
   return parseFloat(ua.substring(IEOffset + 5, ua.indexOf(";", IEOffset)));
}
function qualifyBrowser()
{
   var qualified = false;
      if (navigator.appName == "Microsoft Internet Explorer")
      {
         if (parseInt(getIEVersion()) >= 7)
         {
            if (ua.indexOf("Windows") != -1)
            {
               qualified = true;
            }
         }
      }
      if (!qualified)
      {
         var msg = "These scripts are currently certified to run on:\n";
         msg += " - MS Internet Explorer 7.0 or later for Windows\n";
         alert(msg);
      }
   return qualified;
}
```

As clever as the preceding code looks, using it assumes that the version string surrounding the `MSIE` characters will be immutable in the future. We do not have that kind of guarantee, so you have to remain vigilant for possible changes in future versions.

Thus, with each browser generation's pollution of the `appVersion` and `userAgent` properties, the properties become increasingly less useful for browser sniffing — unless you want to burden your code with a lot of general-purpose sniffing code, very little of which any one browser uses.

Even Mozilla is not free of problems. For example, the main numbering in the `appVersion` property for Firefox 3.5.6 is 5 (in other words, the fifth generation of the original Mozilla code). A potentially thornier problem arises due to the elimination of some nonstandard NN4 DOM features from the Mozilla DOM (`layer` objects and some event object behaviors). Many

scripters followed the previously recommended technique to “prepare for the future” by using an `appVersion` of 4 as a minimum:

```
var isNN4 = parseInt(navigator.appVersion) >= 4;
```

But any code that relies on the `isNN4` variable to branch to code that talks to the dead-end NN4 objects and properties breaks when it runs in Mozilla. This is the dreaded scenario where not every browser feature was retained for backward compatibility as the browser evolved.

The bottom-line question is, “What do I do for browser version detection?” Unfortunately, there are dozens of answers to that question, depending on what you need browser detection to do and what level of code you produce.

At one end of the spectrum is code that tries to be many things to many browsers, implementing multiple levels of features for many different generations of browser. This is clearly the most difficult tactic, and you have to create quite a long list of variables for the conditions for which you establish branches. Some branches may work on one combination of browsers, whereas you may need to split other branches differently because the scripted features have more browser-specific implementations.

At the other end of the spectrum is the code that tries to support, say, only IE5+ and other modern browsers with W3C DOM-compatible syntax to the extent that both browser families implement the object model features. Life for this scripter is much easier in that the amount of branching is little or none, depending on what the scripts do with the objects. Given that modern browsers are now prevalent in mainstream computing, it’s not unreasonable to take this approach and simplify the whole browser compatibility issue.

If you can’t make the assumption that your users have a modern browser, then there are other options for cluing in your scripts to what a particular browser is capable of handling. Object detection (for example, seeing if `document.images` exists before manipulating image objects) is a good solution at times, not so much for determining the browser version, but for knowing whether some code that addresses those objects works. As described in Chapter 25, “Document Object Model Essentials,” it is hazardous to use the existence of, say, `document.all` as an indicator that the browser is IE4+. Some other browser in the future may also implement the `document.all` property, but not necessarily all the other IE4+ objects and syntax. Code that thinks it’s running in IE4+ just because `document.all` exists can easily break if another browser implements `document.all` but not the rest of the IE4+ DOM. Using object detection to branch code that addresses the detected objects is, however, very desirable in the long run because it frees your code from getting trapped in the ever-changing browser version game.

Don’t write off the `appVersion` and `userAgent` properties entirely. The combination of features that you script may benefit from some of the data in that string, especially when the decisions are made in concert with the `navigator.appName` property. A number of other properties can also provide sufficient clues for your code to perform the branching that your application needs. For example, it may be very helpful to your scripts to know whether the `navigator.platform` property informs them that they are running in a Windows or Macintosh environment because of the way each operating system renders fonts.

userAgent property details

The string returned by the `navigator.userAgent` property contains a more complete rundown of the browser. The `userAgent` property is a string, similar to the `USER_AGENT` header, that the browser sends to the server at certain points during the connection process between client and server.

Unfortunately, there is no standard for the way information in the `userAgent` property is formatted or how details about operating systems and browser versions are presented. Table 42-1 shows some of the values that your scripts are likely to see. This table does not include, of course, the many values that are not reflected by browsers that do not support JavaScript. The purpose of the table is to show you just a sampling of data that the property can contain from a variety of browsers and operating systems (particularly enlightening if you do not have access to Macintosh or UNIX computers).

Because the `userAgent` property contains a lot of the same information as the `appVersion` property, the same cautions just described apply to the `userAgent` string and the environment data it returns.

Speaking of compatibility and browser versions, the question often arises whether your scripts should distinguish among incremental releases within a browser's generation (for example, 7.0, 7.01, 7.02, and so on). The latest incremental release occasionally contains bug fixes and (rarely) new features on which you may rely. If that is the case, we suggest you look for this information when the page loads and recommend to the user that he or she download the latest browser version. Beyond that, we suggest scripting for the latest version of a given generation and not bothering with branching for incremental releases.

See Chapters 4 and 25 for more information about designing pages for cross-platform deployment.

Example

Listing 42-1 provides a number of reusable functions that your scripts can employ to determine a variety of information about the currently running browser. This is not intended in any way to be an all-inclusive browser-sniffing routine; instead, we offer samples of how to extract information from the key `navigator` properties to determine various browser conditions. Note especially the test for Apple's Safari browser.

All functions in Listing 42-1 return a Boolean value inline with the pseudo-question presented in the function's name. For example, the `isWindows()` function returns `true` if the browser is any type of Windows browser; otherwise, it returns `false`. If this kind of browser detection occurs frequently in your pages, consider moving these functions into an external `.js` source library for inclusion in your pages (see Chapter 4).

When you load this page, it presents fields that display the results of each function, depending on the type of browser and client operating system you use. This page serves as a simple test bed for ensuring that a browser reveals the correct version information. (Note that Firefox is still considered a variant of Netscape Navigator, at least in terms of version info.) This example also provides a quick test for how a browser enables you to reference objects: whether it supports `document.all`, `getElementById`, or both.

TABLE 42-1

Typical navigator.userAgent Values

navigator.userAgent	Description
`Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.8.1) Gecko/20061003 Firefox/2.0`	Firefox 2.0 for Windows, running under Windows XP Professional
`Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.8.0.7) Gecko/20060909 Firefox/1.5.0.7`	Firefox 1.5 for Windows, running under Windows XP Home Edition
`Mozilla/5.0 (Macintosh; U; PPC Mac OS X Mach-O; en-US; rv:1.4) Gecko/20030624`	Mozilla 1.4 for MacOS X
`Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.4) Gecko/20030624 Netscape/7.1 (ax)`	Navigator 7.1 for Windows, running under Windows XP Home Edition
`Mozilla/4.74 [en] (X11; U; Linux 2.2.154mdksmp i686)`	Navigator 4.74, English edition for Linux with U.S. encryption
`Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1; .NET CLR 1.1.4322)`	IE 7.0 running under Windows XP Home Edition
`Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; .NET CLR 1.0.3705)`	IE 6.0 running under Windows XP Home Edition
`Mozilla/4.0 (compatible; MSIE 5.5; Windows NT 5.0)`	IE 5.5 running under Windows NT 5.0
`Mozilla/4.0 (compatible; MSIE 5.0; Windows 98; DigExt)`	IE 5.0 for Windows 98 with digital signature
`Mozilla/4.0 (compatible; MSIE 5.0; Mac_PowerPC)`	IE 5.0 running on a PowerPC-equipped Macintosh
`Mozilla/5.0 (Macintosh; U; PPC Mac OS X; en-us)`	Apple Safari 2.0.4 (release no. 419.3) for Mac OS X
`Opera/9.01 (WindowsNT 5.1; U; en) AppleWebKit /418.9 (KHTML, like Gecko) Safari/419.3`	Opera 9.01 (set to identify as Opera) for Windows XP Professional
`Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US) AppleWebKit/532.0 (KHTML, like Gecko) Chrome/3.0.195.38 Safari/532.0`	Google Chrome 3.0.195.38 for Windows XP Professional
`Opera/9.80 (Macintosh; Intel Mac OS X; U; en Presto/2.2.15 Version/10.00`	Opera 10.00 Build 6652 for Mac OS X 10.5.8 with the Java Runtime Environment installed

LISTING 42-1

Functions to Examine Browsers

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>UserAgent Property Library</title>
      <script type="text/javascript">
         // basic brand determination
         function isNav()
         {
            return (navigator.appName == "Netscape" &&
                    !isOpera() &&
                    !isSafari() &&
                    !isChrome() );
         }

         function isIE()
         {
            return (navigator.appName == "Microsoft Internet Explorer");
         }

         function isOpera()
         {
            return (navigator.userAgent.indexOf("Opera") != -1);
         }

         function isSafari()
         {
            return (navigator.userAgent.indexOf("Safari") != -1 &&
                    navigator.userAgent.indexOf("Chrome") == -1);
         }

         function isChrome()
         {
            return (navigator.userAgent.indexOf("Chrome") != -1);
         }

         // operating system platforms
         function isWindows()
         {
            return (navigator.appVersion.indexOf("Win") != -1);
         }

         function isWin95NT()
         {
            return (isWindows() && (navigator.appVersion.indexOf("Win16")
               == -1 && navigator.appVersion.indexOf("Windows 3.1") == -1));
         }
```

```
function isMac()
{
   return (navigator.appVersion.indexOf("Mac") != -1);
}

function isMacPPC()
{
   return (isMac() && (navigator.appVersion.indexOf("PPC")
      != -1 || navigator.appVersion.indexOf("PowerPC") != -1));
}

function isUnix()
{
   return (navigator.appVersion.indexOf("X11") != -1);
}

// browser versions
function isGeneration2()
{
   return (parseInt(navigator.appVersion) == 2);
}

function isGeneration3()
{
   return (parseInt(navigator.appVersion) == 3);
}

function isGeneration3Min()
{
   return (parseInt(navigator.appVersion.charAt(0)) >= 3);
}

function isNav4_7()
{
   return (isNav() && parseFloat(navigator.appVersion) == 4.7);
}

function isMSIE4Min()
{
   return (isIE() && navigator.appVersion.indexOf("MSIE") != -1);
}

function isMSIE8_0()
{
   return (navigator.appVersion.indexOf("MSIE 8.0") != -1);
}

function isNN6Min()
{
```

continued

LISTING 42-1 *(continued)*

```
            return (isNav() && parseInt(navigator.appVersion) >= 5);
         }

         // element referencing syntax
         function isDocAll()
         {
            return (document.all) ? true : false;
         }

         function isDocW3C()
         {
            return (document.getElementById) ? true : false;
         }

         // fill in the form's blanks
         function checkBrowser()
         {
            var form = document.forms[0];
            form.brandNN.value = isNav();
            form.brandIE.value = isIE();
            form.brandSaf.value = isSafari();
            form.brandChr.value = isChrome();
            form.brandOp.value = isOpera();
            form.win.value = isWindows();
            form.win32.value = isWin95NT();
            form.mac.value = isMac();
            form.ppc.value = isMacPPC();
            form.unix.value = isUnix();
            form.ver3Only.value = isGeneration3();
            form.ver3Up.value = isGeneration3Min();
            form.Nav4_7.value = isNav4_7();
            form.Nav6Up.value = isNN6Min();
            form.MSIE4.value = isMSIE4Min();
            form.MSIE8_0.value = isMSIE8_0();
            form.doc_all.value = isDocAll();
            form.doc_w3c.value = isDocW3C();
         }
      </script>
   </head>
   <body onload="checkBrowser()">
      <h1>About This Browser</h1>
      <form>
         <h2>Brand</h2>
         Netscape Navigator: <input type="text" name="brandNN" size="5" />
         Internet Explorer: <input type="text" name="brandIE" size="5" />
         Apple Safari: <input type="text" name="brandSaf" size="5" />
```

```
      Google Chrome: <input type="text" name="brandChr" size="5" />
      Opera: <input type="text" name="brandOp" size="5" />
      <hr />
      <h2>Browser Version</h2>
      3.0x Only (any brand): <input type="text" name="ver3Only" size="5" />
      <p>3 or Later (any brand):
         <input type="text" name="ver3Up" size="5" />
      </p>
      <p>Navigator 4.7: <input type="text" name="Nav4_7" size="5" /></p>
      <p>Navigator 6+: <input type="text" name="Nav6Up" size="5" /></p>
      <p>MSIE 4+: <input type="text" name="MSIE4" size="5" /></p>
      <p>MSIE 8.0: <input type="text" name="MSIE8_0" size="5" /></p>
      <hr />
      <h2>OS Platform</h2>
      Windows: <input type="text" name="win" size="5" />
      Windows 95/98/2000/NT: <input type="text" name="win32" size="5" />
      <p>
         Macintosh: <input type="text" name="mac" size="5" />
         Mac PowerPC: <input type="text" name="ppc" size="5" />
      </p>
      <p>Unix: <input type="text" name="unix" size="5" /></p>
      <hr />
      <h2>Element Referencing Style</h2>
      Use <tt>document.all</tt>:
         <input type="text" name="doc_all" size="5" />
      <p>Use <tt>document.getElementById()</tt>:
         <input type="text" name="doc_w3c" size="5" />
      </p>
    </form>
  </body>
</html>
```

Related Items: `appMinorVersion`, `cpuClass`, `oscpu`, `platform` properties

appMinorVersion

Value: One-character string — Read-Only

Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-, Opera+, Chrome-

In IE parlance, the *minor version* is indicated by the first digit to the right of the decimal in a full version number. But the “version number” referred to here is the number that the `navigator.appVersion` property reports, not the actual version of the browser. For example, although IE5.5 seems to have a version number of 5 and a minor version number of 5, the `appVersion` reports version 4.0. In this case, the `minorAppVersion` reports 0. Thus, you cannot use the `appMinorVersion` property to detect differences between, say, IE5 and IE5.5.

That information is buried deeper within the string returned by `appVersion` and `userAgent`. Note that although Opera supports this property, it currently does not populate it.

Example

Use The Evaluator (Chapter 4, "JavaScript Essentials") to examine the two related version properties of your IE browser(s). Type the following two statements into the top text box and observe the results:

```
navigator.appVersion
navigator.minorAppVersion
```

There is a good chance that the values returned are not related to the browser version number shown after `MSIE` in the `appVersion` value.

Related Item: `appVersion` property

browserLanguage

Value: String Read-Only

Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-, Opera+, Chrome-

The `browserLanguage` property in IE4+ (and the `language` property in NN4+/Moz) returns the identifier for a localized language version of the program (it has nothing to do with scripting or programming language). The value of the `browserLanguage` property almost always is the same as the other IE language-related properties, unless the user changes the Windows control panel for regional settings after installing IE. In that case, `browserLanguage` returns the original language of the browser application, whereas the other properties report the language indicated in the system-level preferences panel.

Note

Users of the multilanguage version of Windows XP and Windows Vista can choose alternate languages for menus and dialog boxes. The `browserLanguage` property returns the language you choose for those settings. ■

These short strings may resemble, but are not identical to, the URL suffixes for countries. Moreover, when a language has multiple dialects, the dialect can also be a part of the identifier. For example, `en` is the identifier for English. However, `en-us` (or `en-US`) represents the American dialect of English, whereas `en-gb` (or `en-GB`) represents the dialect recognized in Great Britain. NN sometimes includes these values as part of the `userAgent` data as well. Table 42-2 shows a sampling of language identifiers used for all language-related properties of the `navigator` object.

You can assume that a user of a particular language version of the browser or system is also interested in content in the same language. If your site offers multiple language paths, you can use this property setting to automate the navigation to the proper section for the user.

TABLE 42-2

Sample navigator.browserLanguage Values

navigator.language	Language
En	English
De	German
Es	Spanish
Fr	French
Ja	Japanese
Da	Danish
It	Italian
Ko	Korean
nl	Dutch
pt	Brazilian Portuguese
sv	Swedish

Related Items: navigator.userAgent, navigator.language, navigator.systemLanguage, navigator.userLanguage properties

cookieEnabled

Value: Boolean Read-Only

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

The cookieEnabled property allows your scripts to easily determine if the browser has cookie functionality turned on. You can surround cookie-related statements with an if construction as follows:

```
if (navigator.cookieEnabled)
{
   // do cookie stuff here
}
```

This works reliably only on browsers that implement the property, which includes all modern browsers. Because legacy browsers do not have this navigator object property, the if condition appears false (even though cookies may be turned on).

You can still check for cookie functionality in older browsers, but only clumsily. The technique entails assigning a "dummy" cookie value to the document.cookie property and attempting to read back the cookie value. If the value is there, cookies are enabled.

Example

Use The Evaluator (Chapter 4, "JavaScript Essentials") to see the value of the `navigator.cookieEnabled` property on your browsers. Enter the following statement into the top text box:

```
navigator.cookieEnabled
```

Feel free to change the cookie preferences setting temporarily to see the new value of the property. You do not have to relaunch the browser for the new setting to take effect.

Related Item: `document.cookie` property

cpuClass

Value: String Read-Only

Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-, Opera-, Chrome-

The `cpuClass` property returns one of several fixed strings that identifies the family of central processing units running IE. Possible values and their meanings are as follows:

cpuClass	Description
`x86`	Intel processor (and some emulators)
`PPC`	Motorola Power PC processor (for example, Macintosh)
`68K`	Motorola 68000-family processor (for example, Macintosh)
`Alpha`	Digital Equipment Alpha processor
`Other`	Other processors, such as SPARC

The processor is not a good guide to determining the operating system because you can run multiple operating systems on most of the preceding processor families. As an example, Apple's shift to Intel microprocessors in 2006 confused the familiar notion that Macs are Motorola and Windows PCs are Intel. Moreover, the `cpuClass` value represents the processor that the browser "thinks" it is running on. For example, when a Windows version of IE is hosted by the *Virtual PC* emulator on a PowerPC (Motorola) Macintosh, the `cpuClass` is reported as `x86` even though the actual hardware processor is `PPC`.

Example

Use The Evaluator (Chapter 4, "JavaScript Essentials") to see how IE reports the `cpuClass` of your PC. Enter the following statement into the top text box:

```
navigator.cpuClass
```

Related Item: `navigator.oscpu` property

`language`

Value: String Read-Only

Compatibility: WinIE-, MacIE-, NN4+, Moz+, Safari+, Opera+, Chrome+

The NN4+/Moz `language` property returns the language code for the browser application. Although the comparable IE property (`navigator.browserLanguage`) has morphed in later versions to focus on the operating system language, `language` deals exclusively with the language for which the browser application is written.

Related Item: `navigator.browserLanguage` property

`mimeTypes`

Value: Array of `mimeType` objects Read-Only

Compatibility: WinIE-, MacIE5, NN4+, Moz+, Safari+, Opera+, Chrome+

A `mime` *(Multipurpose Internet Mail Extension)* type is a file format for information that travels across the Internet. Browsers usually have a limited built-in capability for displaying or playing information beyond HTML text and one or two image standards (`.gif`, `.jpg`, and now, more frequently, `.png` are the most common formats). To fill in the gap, browsers maintain an internal list of `mime` types with corresponding instructions on what to do when information of a particular `mime` type arrives at the client. For example, when a server application serves up an audio stream in an audio format, the browser locates that `mime` type in its table (the `mime` type is among the first chunks of information to reach the browser from the server), and then launches a helper application or activates a plug-in capable of playing that `mime` type. Your browser is not equipped to display every `mime` type, but it does know how to alert you when you don't have the helper application or plug-in needed to handle an incoming file. For instance, the browser may ask if you want to save the file for later use or switch to a web page containing more information about the necessary plug-in.

The `mimeTypes` property of the `navigator` object is simply the array of `mime` types about which your browser knows (see the "mimeType Object" section later in this chapter). NN/Moz browsers come with dozens of `mime` types already listed in their tables (even if the browser doesn't have the capability to handle all those items automatically). If you have third-party plug-ins in Mozilla's plug-ins directory/folder, or helper applications registered with your browser, that array contains these new entries as well.

If your web pages are media-rich, you want to be sure that each visitor's browser is capable of playing the media your page has to offer. With JavaScript and NN/Moz, you can cycle through the `mimeTypes` array to find a match for the `mime` type of your media. Then use the properties of the `mimeType` object (detailed later in this chapter) to ensure that the optimum plug-in is available. If your media still requires a helper application instead of a plug-in, the array only lists the `mime` type; thus, you can't determine whether a helper application is assigned to this `mime` type from the array list.

Example

For examples of the `mimeTypes` property and details about using the `mimeType` object, see the discussion of this object later in the chapter. A number of simple examples showing how to use this property to see whether the `navigator` object has a particular MIME type do not go far enough in determining whether a plug-in is installed and enabled to play the incoming data.

Related Item: `navigator.plugins` property; `mimeType` object

onLine

Value: Boolean Read-Only

Compatibility: WinIE4+, MacIE4+, NN-, Moz-, FF+, Safari+, Opera+, Chrome+

The `onLine` property lets scripts determine the state of the offline browsing setting for the browser. Bear in mind that this property does not reveal whether the page is accessed via the Net or a local hard disk. The browser can be in online mode and still access a local page; in this case, the `onLine` property returns `true`.

With the offline browsing capabilities of many modern browsers, users may prefer to download copies of pages they wish to reference frequently (perhaps on a disconnected laptop computer). In such cases, your pages may want to avoid network-reliant content when accessed offline. For example, if your page includes a link to a live audio feed, you can dynamically generate that link with JavaScript — but only if the user is online:

```
if (navigator.onLine)
{
   document.write("<a href='broadcast.rna'>Listen to Audio</a>");
}
```

Example

Use The Evaluator (Chapter 4, “JavaScript Essentials”) to see the online state of your IE browsers. Enter the following statement into the top text box:

```
navigator.onLine
```

Verify your browsing mode by checking the Work Offline choice in the File menu. If it is checked, the `onLine` property should return `false`.

oscpu

Value: String Read-Only

Compatibility: WinIE-, MacIE-, NN6+, Moz+, Safari-, Opera-, Chrome-

The `oscpu` property of Mozilla-based browsers returns a string that reveals OS- or CPU-related information about the user’s environment. The precise string varies widely with the client OS.

For instance, a Windows 98 machine reports `Win98`, whereas a Macintosh running MacOS X reports `PPC Mac OS X Mach-O`. The string formats for Windows NT/XP versions are not standardized, so they offer values such as `WinNT4.0` and `Windows NT 5.1`. Windows XP reports as being `Windows NT 5.x` in the `oscpu` property (as it does in the `userAgent` property), whereas Windows Vista reports as being Windows NT 6.x. UNIX platforms reveal more details, such as the system version and hardware.

Example

Use The Evaluator (Chapter 4, "JavaScript Essentials") with NN6+/Moz to see what your client machine reports to you by entering the following statement into the top text box:

```
navigator.oscpu
```

Related Item: `navigator.cpuClass` property

platform

Value: String Read-Only

Compatibility: WinIE4+, MacIE4+, NN4+, Moz+, Safari+, Opera+, Chrome+

The `navigator.platform` value reflects the operating system according to the codes established initially by Netscape for its `userAgent` values. Table 42-3 lists the typical values of several operating systems.

TABLE 42-3

Sample navigator.platform Values

navigator.platform	Operating System
`Win32`	Windows XP
`Win98`	Windows 98
`WinNT`	Windows NT
`Win16`	Windows 3.*x*
`MacIntel`	Mac (CPU)
`MacPPC`	Mac (PowerPC CPU)
`SunOS`	Solaris

In the long list of browser detection functions in Listing 42-1, we elected not to use the `navigator.platform` property because it is not backward-compatible. Meanwhile, the other properties in that listing are available to all scriptable browsers.

Notice that the `navigator.platform` property does not go into versioning of the operating system. Only the raw name is provided. Also, there is a `Win64` value that should be reported for 64-bit Windows versions, such as the 64-bit version of Windows Vista. However, this value hasn't been reliably reported in previous 64-bit Windows operating systems.

Example

Use The Evaluator (Chapter 4, "JavaScript Essentials") to see what your computer reports as its operating system. Enter the following statement into the top text box:

```
navigator.platform
```

Related Item: `navigator.userAgent` property

plugins

Value: Array of plug-in objects Read-Only

Compatibility: WinIE-, MacIE5, NN3+, Moz+, Safari+, Opera+, Chrome+

You rarely find people involved with web page design who have not heard about *plug-ins* — the technology that enables you to embed new media types and foreign file formats directly into web documents. For instance, instead of requiring you to watch a video clip in a separate viewer atop the main browser window, a plug-in enables you to make that viewer as much a part of the page design as a static image. The same goes for audio players, 3-D animation, chat sessions — even the display of Microsoft Office documents, such as PowerPoint and Word.

When many browsers launch, they create an internal list of available plug-ins located in a special directory/folder (the name varies with the browser and operating system). The `navigator.plugins` array lists the items registered at launch time. Each plug-in is, itself, an object with several properties.

The Windows version of IE4+ supports this property only to return an empty array. In other words, the property is defined, but it does not contain `plugin` objects — a nonexistent object in IE for Windows. But on the Macintosh side, IE5 supports the way Netscape Navigator, Mozilla, and Safari allow script inspection of `mime` types and plug-ins. To see ways of determining plug-in support for WinIE, see the section "'Plug-in' detection in WinIE" later in this chapter.

Having your scripts investigate the visitor's browser for a particular installed plug-in is a valuable capability if you want to guide the user through the process of downloading and installing the plug-in if the system does not have it currently.

Example

For examples of the `plugins` property and for details about using the `plugin` object, see the section "`plugin` Object" later in this chapter. Also see Chapter 44, "Embedded Objects," for information on embedded element objects.

Related Items: `navigator.mimeTypes property`; `plugin` object

product
productSub
vendor
vendorSub

Value: String Read-Only

Compatibility: WinIE-, MacIE-, NN6+, Moz+, Safari+, Opera-, Chrome+

With the Mozilla browser engine being developed in an Open Source environment, any number of vendors might adapt the engine for any number of browser products. Some distributors of the browser, such as ISPs and computer manufacturers, may also tailor the browser slightly for their customers. These four properties can reveal some of the pedigree of the browser currently running scripts on the page.

Two categories of properties — one for the product, one for the vendor — each have a pair of fields (a primary and secondary field) that can be populated as the vendor sees fit. Some of this information may contain data, such as the identifying number of the *build* (development version) used to generate the product. A script at a computer maker's web site may look for a particular series of values in these properties to welcome the customer, or to advise the customer of a later build version that is recommended as an upgrade.

Example

Use The Evaluator (Chapter 4, "JavaScript Essentials") on NN6+/Moz and WebKit-based browers to see the values returned for these four properties (Safari 1.0 doesn't support `vendorSub`). Enter each of the following statements into the top text box of the page and see the values for each in the Results box:

```
navigator.product
navigator.productSub
navigator.vendor
navigator.vendorSub
```

Also check the value of the `navigator.userAgent` property to see how many of these four property values are revealed in the `userAgent` property.

Related Item: `navigator.userAgent` property

securityPolicy

Value: String Read-Only

Compatibility: WinIE-, MacIE-, NN4+, Moz+, Safari-, Opera-, Chrome-

The Netscape-specific `securityPolicy` property returns a string that indicates which cryptographic scheme is implemented in the current browser. Typical string values include `US and CA domestic policy` and `export policy`. Each policy indicates the number of bits

used for encryption, usually governed by technology export laws. The corresponding IE property is `document.security`. All Mozilla-based browsers have only one version, and the property's value is an empty string.

Related Item: `document.security` property

systemLanguage
userLanguage

Value: Language code string Read-Only

Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-, Opera+, Chrome-

These two IE-specific properties report the language code of the written language specified for the operating system. For most operating system versions, these two values are the same. Some Windows versions enable you to set system preferences differently for the base operating system and the language for a given user. Both of these property values can differ from the `navigator.browserLanguage` property if the user downloads and installs the browser with the system set to one language, and then changes the system settings to another language. Note that Opera only supports `systemLanguage`.

Example

Use The Evaluator (Chapter 4, "JavaScript Essentials") with your IE4+ browser to compare the values of the three language-related properties running on your computer. Enter each of the following statements into the top text box:

```
navigator.browserLanguage
navigator.systemLanguage
navigator.userLanguage
```

Don't be surprised if all three properties return the same value.

Related Item: `navigator.browserLanguage` property

userAgent

(See `appCodeName`)

userLanguage

(See `systemLanguage`)

userProfile

Value: `userProfile` object Read-Only

Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-, Opera-, Chrome-

The `userProfile` property returns a reference to the IE `userProfile` object. This object provides scripted access to a limited range of user profile settings, with the user's permission. For details, see the `userProfile` object discussion later in this chapter.

Related Item: `userProfile` object

`vendor`
`vendorSub`

(See product)

Methods

`javaEnabled()`

Returns: Boolean

Compatibility: WinIE4+, MacIE4+, NN3+, Moz+, Safari+, Opera+, Chrome+

Although most modern browsers ship with Java support turned on, a user can easily turn it off in a preferences dialog box (or even elect not to install it with the browser). Some corporate installations may also turn off Java as the default setting for their users. If your pages specify Java applets, you don't normally have to worry about this property because the applet tag's alternate text fills the page in the places where the applet normally goes. But if you script applets from JavaScript (see Chapter 47), you don't want your scripts making calls to applets or Java classes if Java support is turned off. In a similar vein, if you create a page with JavaScript, you can fashion two different layouts, depending on the availability of Java.

The `navigator.javaEnabled()` method returns a Boolean value reflecting the preferences setting. This value does not necessarily reflect Java support in the browser (and certainly not the Java version supported), but rather whether Java is turned on inside the browsers for which this method is supported. A script cannot change the browser's preference setting, but its value does change immediately when the user toggles the setting.

Related Items: `navigator.preference()` method; NPAPI (Chapter 47, "Scripting Java Applets and Plug-Ins")

`preference(`*`name`*`[, `*`val`*`])`

Returns: Preference value

Compatibility: WinIE-, MacIE-, NN4+, Moz1+, Safari-, Opera-, Chrome-

The user normally sets browser preferences. Until NN4 and the advent of signed scripts, almost all settings were completely out of view of scripts — even when it made sense to expose them. But with signed scripts and the `navigator.preference()` method, many NN preferences are now viewable and settable with the user's permission. These preferences were exposed to scripting primarily for the purposes of centralized configuration administration for enterprise installations. We don't recommend altering the browser preferences of a public web site visitor, even if given permission to do so — the user may not know how much trouble you can cause.

When you want to read a particular preference setting, you pass only the preference name parameter with the method. Reading a preference requires a signed script with the target of `UniversalPreferencesRead` (see Chapter 49, "Security and Netscape Signed Scripts," on the

navigatorObject.**preference()**

CD-ROM). To change a preference, pass both the preference name and the value (with a signed script target of `UniversalPreferencesWrite`).

Table 42-4 shows a handful of scriptable preferences in Mozilla-based browsers (learn more about Mozilla preferences at `http://www.mozilla.org/catalog/end-user/customizing/briefprefs.html`). Most items have corresponding entries in the preferences window in NN4+ (shown in parentheses) and are available in Firefox via the Options window. Notice that the preference name uses dot syntax. The cookie security level is a single preference value with a matrix of integer values indicating the level.

Tip

One preference to watch out for is the one that disables JavaScript. If you disable JavaScript, only the user can reenable JavaScript by manually changing the setting in the browser's preferences dialog box. ■

TABLE 42-4

navigator.preference() Values Sampler

navigator.preference	Value	Preference Dialog Listing
`security.enable_java`	Boolean	(Advanced) Enables Java
`javascript.enabled`	Boolean	(Advanced) Enables JavaScript
`autoupdate.enabled`	Boolean	(Advanced) Enables AutoInstall
`network.cookie.cookieBehavior`	0	(Advanced) Accepts all cookies
`network.cookie.cookieBehavior`	1	(Advanced) Accepts only cookies that get sent back to the originating server
`network.cookie.cookieBehavior`	2	(Advanced) Disables cookies
`network.cookie.warnAboutCookies`	Boolean	(Advanced) Warns you before accepting a cookie

Example

The page in Listing 42-2 displays check boxes or radio buttons for several preference settings, plus one text box, to show a preference setting value for the size of the browser's disk cache. You will receive a security warning each time the scripts enable the Privilege Manager.

One function reads all the preferences and sets the form control values accordingly. Another function sets a preference when you click its check box. Rerunning the `showPreferences()` function also helps verify that you set the preference.

LISTING 42-2

Reading and Writing Browser Preferences

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Reading/Writing Browser Preferences</title>
      <script type="text/javascript">
         function setPreference(pref, value)
         {
netscape.security.PrivilegeManager.enablePrivilege("UniversalPreferencesWrite");
            navigator.preference(pref, value);
netscape.security.PrivilegeManager.revertPrivilege("UniversalPreferencesWrite");
            showPreferences();
         }

         function showPreferences()
         {
            var form = document.forms[0];
               netscape.security.PrivilegeManager.enablePrivilege
                  ("UniversalPreferencesRead");
            form.cacheSize.value =
               navigator.preference("browser.cache.disk.capacity");
            form.autoIEnable.checked = navigator.preference("autoupdate.enabled");
            var cookieSetting =
               navigator.preference("network.cookie.cookieBehavior");
            for (var i = 0; i < 3; i++)
            {
               document.getElementById("cookie" + i).checked = (i == cookieSetting)
                  ? true : false;
            }
            var toolbarSetting =
               navigator.preference("browser.chrome.toolbar_style");
            for (var i = 0; i < 3; i++)
            {
               document.getElementById("toolbar" + i).checked = (i ==
               toolbarSetting) ? true : false;
            }
            form.cookieWarn.checked =
              navigator.preference("network.cookie.warnAboutCookies");
                 netscape.security.PrivilegeManager.revertPrivilege
                    ("UniversalPreferencesRead");
         }
      </script>
   </head>
   <body onload="showPreferences()">
      <h1>Mozilla Browser Preferences Settings Sampler</h1>
```

continued

***navigatorObject*.preference()**

LISTING 42-2 *(continued)*

```
      <hr />
      <form>
         <input type="checkbox" name="autoIEnable"
                onclick="setPreference('autoupdate.enabled',this.checked)" />
         AutoInstall Enabled<br />
         <b>Toolbar Buttons: </b>
         <input type="radio" name="toolbarPriv" id="toolbar0"
                onclick="setPreference('browser.chrome.toolbar_style',0)" />
         Toolbar Pictures Only
         <input type="radio" name="toolbarPriv" id="toolbar1"
                onclick="setPreference('browser.chrome.toolbar_styler',1)" />
         Toolbar Text Only
         <input type="radio" name="toolbarPriv" id="toolbar2"
                onclick="setPreference('browser.chrome.toolbar_style',2)" />
         Toolbar Pictures & Text
         <br />
         <b>Cookie Permissions: </b>
         <input type="radio" name="cookiePriv" id="cookie0"
                onclick="setPreference('network.cookie.cookieBehavior',0)" />
         Accept All Cookies
         <input type="radio" name="cookiePriv" id="cookie1"
                onclick="setPreference('network.cookie.cookieBehavior',1)" />
         Accept Only Cookies Sent Back to Server
         <input type="radio" name="cookiePriv" id="cookie2"
                onclick="setPreference('network.cookie.cookieBehavior',2)" />
         Disable Cookies<br />
         <input type="checkbox" name="cookieWarn"
             onclick="setPreference('network.cookie.warnAboutCookies',
                this.checked)" />
         Warn Before Accepting Cookies
         <br />
         Disk cache is <input type="text" name="cacheSize" size="10" />KB
         <br />
      </form>
   </body>
</html>
```

Note

The property assignment event handling technique used in this example and throughout the chapter is a deliberate simplification to make the code more readable. It is generally better to use the more modern approach of binding events using the `addEventListener()` **(NN6+/Moz/W3C) or** `attachEvent()` **(IE5+) methods. A modern cross-browser event handling technique is explained in detail in Chapter 32, "Event Objects."** ■

Related Item: `navigator.javaEnabled()` method

mimeType Object

Properties	Methods	Event Handlers
description		
enabledPlugin		
type		
suffixes		

Syntax

Accessing mimeType properties:

```
navigator.mimeTypes[i].property
navigator.mimeTypes["MIMEtype"].property
```

Compatibility: WinIE-, MacIE5, NN3+, Moz+, Safari+, Opera+, Chrome+

About this object

A mimeType object is essentially an entry in the internal array of mime types about which the browser knows. NN3+/Moz, for example, ships with an internal list of more than five dozen mime types. Only a handful of these types are associated with helper applications or plug-ins. But add to that list all of the plug-ins and other helpers you've installed, and the number of mime types can grow to more than a hundred.

The mime type for the data is usually among the first bits of information to arrive at a browser from the server. A mime type consists of two pieces of information: type and subtype. The traditional way of representing these pieces is as a pair separated by a slash, as in

```
text/html
image/gif
audio/wav
video/quicktime
application/pdf
application/x-zip-compressed
```

If a file does not contain the mime type "header" (or a CGI program sending the file does not precede the transmission with the mime type string), the browser receives the data as a text/plain mime type. When you load the file from a local hard drive, the browser looks to the filename's extension (the suffix after the period) to figure out the file's type.

Regardless of the way it determines the mime type of the incoming data, the browser then acts according to instructions it maintains internally. You can see these settings by looking at preferences settings usually associated with the name "Applications."

By having the `mimeType` object available to JavaScript, your page can query a visitor's NN3+/Moz, MacIE5, or WebKit-based browser to discover whether it has a particular `mime` type listed currently, and whether the browser has a corresponding plug-in installed and enabled. In such queries, the `mimeType` and `plugin` objects work together to help scripts make these determinations. (For plug-in detection in WinIE, see the section "'Plug-in' detection in WinIE," later in this chapter.)

Because of the close relationship between `mimeType` and `plugin` objects, we save the examples of using these objects and their properties for a section later in this chapter. There you can see how to build functions into your scripts that enable you to examine how well a visitor's browser is equipped for either a `mime` type or data that requires a specific plug-in. In the meantime, be sure that you understand the properties of both objects.

Properties

description

Value: String Read-Only

Compatibility: WinIE-, MacIE5, NN3+, Moz+, Safari+, Opera+, Chrome+

While registering with the browser at launch time, plug-ins provide the browser with an extra field of information: a plain-language description of the plug-in. If a particular `mime` type has a plug-in associated with it and enabled for it, the plug-in's description passes through to become the description of the `mimeType` object. For example, the Adobe Acrobat plug-in (whose `mime` type is `application/pdf`) supplies the following description fields:

```
(NN3/NN4)   Acrobat
(Moz)       Acrobat Portable Document Format
```

To verify this in your own browser, enter the following line of code into the Evaluator (with NN3+/Moz, MacIE5, and Safari):

```
navigator.mimeTypes["application/pdf"].description
```

When a `mime` type does not have a plug-in associated with it (either no plug-in is installed or a helper application is used instead), you often see the `type` property value repeated in the description field.

enabledPlugin

Value: `plugin` object Read-Only

Compatibility: WinIE-, MacIE5, NN3+, Moz+, Safari+, Opera+, Chrome+

The descriptions of the `mimeType` and `plugin` objects seem to come full circle when you reach the `mimeType.enabledPlugin` property. The reason is that the property is a vital link between a known `mime` type and the plug-in that the browser engages when data of that type arrives.

Knowing which plug-in is associated with a `mime` type is very important when you have more than one plug-in capable of playing a given `mime` type. For example, the Crescendo `midi` audio plug-in can take the place of the default audio plug-in if you set up your browser that way. Therefore, all `midi` data streams play through the Crescendo plug-in. If you prefer to have your web page's `midi` sound played only through another plug-in, your script needs to know which plug-in is set to receive your data and perhaps alert the user accordingly. These kinds of conflicts are not common, except where there is strong competition for players of various audio and video media. For other kinds of content, each plug-in developer typically creates a new type of data that has a unique `mime` type. But you have no guarantee of such uniqueness, so we highly recommend a careful check of `mime` type and plug-in if you want your page to look professional.

The `enabledPlugin` property evaluates to a `plugin` object. Therefore, you can dig a bit deeper with this information to fetch the name or filename properties of a plug-in directly from a `mimeType` object. You can use The Evaluator to study the relationship between `mimeType` and `plugin` objects:

1. Enter the following statement into the bottom text box to examine the properties of a `mimeType` object:

   ```
   navigator.mimeTypes[0]
   ```

   Notice that the `mimeTypes` array returns an object.

2. Inspect the `plugin` object from the bottom text box:

   ```
   navigator.mimeTypes[0].enabledPlugin
   ```

   You then see properties and values for a `plugin` object (described later in this chapter).

3. Check the `plugin` object for a different `mimeType` object by using a different index value:

   ```
   navigator.mimeTypes[7].enabledPlugin
   ```

The `mimeTypes` array index values vary with almost every browser, depending on what the user has installed. Therefore, do not rely on the index position in a script to assume that a particular `mimeType` object is in that position on all browsers.

Example

See the section "Looking for `mime` Types and Plug-ins" later in this chapter.

Related Item: `plugin` object

`type`

Value: String — Read-Only

Compatibility: WinIE-, MacIE5, NN3+, Moz+, Safari+, Opera+, Chrome+

A `mimeType` object's `type` property is the combination of the type and subtype commonly used to identify the kind of data coming from the server. Server applications, for

example, typically precede a data transmission with a special header string in the following format:

```
Content-type: type/subtype
```

This string prompts a browser to look up how to treat an incoming data stream of this kind. As you see later in this chapter, knowing whether a particular `mime` type is listed in the `navigator.mimeTypes` array is not enough. A good script must dig deeper to uncover additional information about what is truly available for your data.

The `type` property has a special place in the `mimeType` object, in that its string value can act as the index to the `navigator.mimeTypes` array. Therefore, to get straight to the `mimeType` object for, say, the `audio/wav mime` type, your script can reference it directly through the `mimeTypes` array:

```
navigator.mimeTypes["audio/wav"]
```

This same reference can then get you straight to the enabled plug-in (if any) for the `mime` type:

```
navigator.mimeTypes["audio/wav"].enabledPlugin
```

Example

See the section “Looking for `mime` Types and Plug-ins” later in this chapter.

Related Item: `description` property

suffixes

Value: String Read-Only

Compatibility: WinIE-, MacIE5, NN3+, Moz+, Safari+, Opera+, Chrome+

Every `mime` type has one or more filename extension, or suffix, associated with it. You can read this information for any `mimeType` object via the `suffixes` property. The value of this property is a string. If the `mime` type has more than one suffix associated with it, the string contains a comma-delimited listing, as in

```
mpg, mpeg, mpe
```

Multiple versions of a suffix have no distinction among them. Those `mime` types that are best described in four or more characters (derived from a meaningful acronym, such as mpeg) have three-character versions to accommodate the “8-dot-3” filename conventions of MS-DOS and its derivatives.

Example

See the section “Looking for `mime` Types and Plug-ins” later in this chapter.

plugin Object

Properties	Methods	Event Handlers
name	refresh()	
filename		
description		
length		

Syntax

Accessing plugin object properties or method:

```
navigator.plugins[i].property | method()
navigator.plugins["plugInName"].property | method()
```

Compatibility: WinIE-, MacIE5, NN3+, Moz+, Safari+, Opera+, Chrome+

About this object

The plugin object offers a view of the plug-in mechanism from the browser's perspective: The software items registered with the browser at launch time stand ready for any matching mime type that comes from the Net. One of the main purposes of having these objects scriptable is to let your scripts determine whether a desired plug-in is currently registered with the browser and to help with installing the plug-in if it isn't.

The close association between the plugin and mimeType objects, demonstrated by the mimeType.enabledPlugin property, is equally visible coming from the direction of the plug-in. A plugin object evaluates to an array of mime types that the plug-in interprets. Use The Evaluator (Chapter 4, "JavaScript Essentials") to experiment with mime types from the point of view of a plug-in. Begin by finding the name of the plug-in that your browser uses for a common video mime type:

1. Enter the following statement into the top text box:

   ```
   navigator.mimeTypes["video/quicktime"].enabledPlugin.name
   ```

   If you use NN7+/Moz, MacIE5, or a WebKit-based browser, the value returned is probably "QuickTime Plug-in 7.6.5", possibly with a different version number. Copy the name onto the clipboard so that you can use it in subsequent statements. The remaining examples show "QuickTime Plug-in 7.6.5" where you should paste in your plug-in's name.

2. Enter the following statement into the top text box:

```
navigator.plugins["QuickTime Plug-in 7.6.5"].length
```

   Instead of the typical index value for the array notation, use the actual name of the plug-in. This expression evaluates to a number indicating the total number of different `mime` types that the plug-in recognizes.

3. Look at the first `mime` type specified for the plug-in by entering the following statement into the top text box:

```
navigator.plugins["QuickTime Plug-in 7.6.5"][0].type
```

The two successive pairs of square brackets is not a typo: Because the entry in the `plugins` array evaluates to an array itself, the second set of square brackets describes the index of the array returned by `plugins["QuickTime Plug-in 7.6.5"]` — a period does not separate the sets of brackets. In other words, this statement evaluates to the `type` property of the first `mimeType` object contained by the Quicktime plug-in.

We doubt that you will have to use this kind of construction much; if you know the name of the desired plug-in, you know what `mime` types it already supports. In most cases, you come at the search from the `mime` type direction and look for a specific, enabled plug-in. See the section "Looking for `mime` Types and Plug-Ins" later in this chapter for details on how to use the `plugin` object in a production setting.

Properties

name
filename
description
length

Value: String Read-Only

Compatibility: WinIE-, MacIE5, NN3+, Moz+, Safari+, Opera+, Chrome+

The first three properties of the `plugin` object provide descriptive information about the plug-in file. The plug-in developer supplies the name and description. It's not a strict requirement that future versions of plug-ins differentiate themselves from earlier ones via either of these fields, although some do. Thus, while there is no explicit property that defines a plug-in's version number, that information may be part of the string returned by the `name` or `description` properties.

Be aware that plug-in authors may not assign the same name to every OS platform version of a plug-in. Be prepared for discrepancies across platforms. You should hope that the plug-in you're interested in has a uniform name across platforms, because the value of the `name` property can function as an index to the `navigator.plugins` array to access a particular `plugin` object directly.

Another piece of information available from a script is the plug-in's filename. On some platforms, such as Windows, this data comes in the form of a complete pathname to the plug-in DLL file; on other platforms, only the plug-in filename appears.

Finally, the `length` property of a `plugin` object counts the number of `mime` types that the plug-in recognizes (but not that are necessarily enabled). Although you can use this information to loop through all possible `mime` types for a plug-in, a more instructive way is to have your scripts approach the issue via the `mime` type (as discussed later in this chapter).

Example

See the section "Looking for `mime` Types and Plug-ins" later in this chapter.

Related Item: `mimeType.description` property

Methods

`refresh()`

Returns: Nothing

Compatibility: WinIE-, MacIE5, NN3+, Moz+, Safari+, Opera+, Chrome+

You may have guessed that many browsers determine their lists of installed plug-ins while they launch. If you drop a new plug-in file into the plug-ins directory/folder, you have to quit the browser and relaunch it before the browser sees the new plug-in file. But that isn't a very friendly approach if you've taken pains to guide a user through downloading and installing a new plug-in file. Once the user quits the browser, you have a slim chance of getting that person right back. That's where the `refresh()` method comes in.

The `refresh()` method is directed primarily at the browser, but the syntax of the call reminds the browser to refresh just the plug-ins:

```
navigator.plugins.refresh()
```

Interestingly, this command works only for adding a plug-in to the existing collection. If the user removes a plug-in and invokes this method, the removed one stays in the `navigator.plugins` array — although it may not be available for use. Only the act of quitting and relaunching the browser makes a plug-in removal take full effect.

Looking for mime Types and Plug-Ins

If you go to great lengths to add new media and data types to your web pages, you certainly want your visitors to reap the benefits of those additions. But you cannot guarantee that they have the requisite plug-ins installed to accommodate that fancy data. Most modern browser versions provide a bit of internal "smarts" by noticing when data requiring an uninstalled plug-in is

about to load, and trying to help the user install a missing plug-in. You may wish, however, to take more control over the process by examining the user's browser plug-in functionality prior to loading the external data file.

The best source of information, when available, is the software developer of the plug-in. Adobe, for example, provides numerous technical notes on their web site (`www.adobe.com`) about plug-in detection for their various plug-ins and versions. Unfortunately, that kind of assistance is not always easy to find from other vendors.

Much of the discussion thus far in this chapter addresses the objects that make plug-in and `mime` type support detection possible in some browsers. Microsoft makes it possible — but not easy — to determine whether a particular plug-in is available for WinIE. The approach for WinIE is entirely different from what we have covered so far; if you wish to perform cross-browser detection, you have to branch your code accordingly. We outline each approach next in its own section, starting with the NN3+/Moz/MacIE5/WebKit browser way.

Overview: Using mimeType and plugin objects

The value of performing your own inspection of plug-in support is that you can maintain better control of your site for visitors who don't have the necessary plug-in yet. Rather than merely providing a link to the plug-in's download site, you can build a more complete interface around the download and installation of the plug-in without losing your visitor. We have some suggestions about such an interface at the end of this discussion.

How you go about inspecting a visitor's plug-in library depends on what information you have about the data file or stream, and how precise you must be in locating a particular plug-in. Some plug-ins may override `mime` type settings that you normally expect to find in a browser. Another issue that complicates matters is that the same plug-in may have a different name (`navigator.plugins[i].name` property), depending on the operating system. Therefore, searching your script for the presence of a plug-in by name is not good enough if the name differs from the Macintosh version to the Windows version. Fortunately, this is less of an issue today than it was in earlier plug-in generations.

One other point that can help you decide the precise approach to take is which information about the plug-in is important to your page and scripts — support for the data `mime` type or the presence of a particular plug-in. If your scripts rely on the existence of a plug-in that you can script via the NPAPI, be sure that the plug-in is present and enabled for the desired `mime` type (so that the plug-in is ensured of loading when it encounters a reference to the URL of the external data). But if you care only that a plug-in of any kind supports your data's `mime` type, you can simply make sure that any plug-in is enabled for your MIME type.

To help you jump-start the process, we discuss three utility functions you can use in your own scripts. These functions are excerpts from a long listing (Listing 42-3), which is located in its entirety with the Chapter 42 listing files on the CD-ROM. The pieces not shown here are merely user interface elements that enable you to experiment with these functions.

Verifying a mime type

Listing 42-3A is a function whose narrow purpose is to determine if the browser currently has plug-in support enabled for a given `mime` type (as a string in the *type/subtype* format). The first `if` construction verifies that there is a `mimeType` object for the supplied `mime` type string. If such an object exists, the next `if` construction determines whether the `enabledPlugin` property of the `mimeType` object returns a valid object. If so, the function returns `true` — meaning that the `mime` type has a plug-in (of unknown supplier) available to play the external media.

LISTING 42-3A

Verifying a MIME Type

```
// Pass "<type>/<subtype>" string to this function to find
// out if the MIME type is registered with this browser
// and that at least some plug-in is enabled for that type.
function mimeIsReady(mime_type)
{
   if (navigator.mimeTypes[mime_type])
   {
      if (navigator.mimeTypes[mime_type].enabledPlugin)
      {
         return true;
      }
   }
   return false;
}
```

Verifying a plug-in

In Listing 42-3B, you let JavaScript see if the browser has a specific plug-in registered in the `navigator.plugins` array. This method approaches the installation question from a different angle. Instead of querying the browser about a known `mime` type, the function inquires about the presence of a known plug-in. But because more than one registered plug-in can support a given `mime` type, this function explores one step further to see whether at least one of the plug-in's `mime` types (of any kind) is enabled in the browser.

LISTING 42-3B

Verifying a Plug-in

```
// Pass the name of a plug-in for this function to see
// if the plug-in is registered with this browser and
// that it is enabled for at least one MIME type of any kind.
function pluginIsReady(plug_in)
{
```

continued

LISTING 42-3B *(continued)*

```
   plug_in = plug_in.toLowerCase();
   for (var i = 0; i < navigator.plugins.length; i++)
   {
      if (navigator.plugins[i].name.toLowerCase().indexOf(plug_in) != -1)
      {
         for (var j = 0; j < navigator.plugins[i].length; j++)
         {
            if (navigator.plugins[i][j].enabledPlugin)
            {
               return true;
            }
         }
         return false;
      }
   }
   return false;
}
```

The parameter for the `pluginIsReady()` function is a string consisting of the plug-in's name. As discussed earlier, the precise name may vary from OS to OS, or from version to version. The function here assumes that you aren't concerned about plug-in versioning. It also assumes (with reasonably good experience behind the assumption) that a brand-name plug-in contains a string with the brand in it. Thus, the `pluginIsRead()` function simply looks for the existence of the passed name within the `plugin` object's `name` property. For example, this function accepts `QuickTime` as a parameter and agrees that there is a match with the plug-in named `QuickTime Plug-in 7.6.5`. The script loops through all registered plug-ins for a substring comparison (converting both strings to all lowercase to help overcome discrepancies in capitalization).

Next comes a second repeat loop, which looks through the `mime` types associated with a plug-in (in this case, only a plug-in whose name contains the parameter string). Notice the use of the strange, double-array syntax for the most nested `if` statement: For a given plug-in (denoted by the `i` index), you have to loop through all items in the `mime` types array (`j`) connected to that plug-in. The conditional phrase for the last `if` statement has an implied comparison against `null`. (See another way of explicitly showing the `null` comparison in Listing 42-3A.) The conditional statement evaluates to either an object or `null`, which JavaScript can accept as `true` or `false`, respectively. The point is that if an enabled plug-in is found for the given `mime` type of the given plug-in, this function returns `true`.

Verifying both plug-in and mime type

The last utility function (Listing 42-3C) is the safest way of determining whether a visitor's browser is equipped with the "right stuff" to play your media. This function requires both a MIME type and a plug-in name as parameters and also makes sure that both items are supported and enabled in the browser before returning `true`.

LISTING 42-3C

Verifying Plug-in and mime Type

```
// Pass "<type>/<subtype>" and plug-in name strings for this
// function to see if both the MIME type and plug-in are
// registered with this browser, and that the plug-in is
// enabled for the desired MIME type.
function mimeAndPluginReady(mime_type,plug_in)
{
   if (mimeIsReady(mime_type))
   {
      var plugInOfRecord = navigator.mimeTypes[mime_type].enabledPlugin;
      plug_in = plug_in.toLowerCase();
      for (var i = 0; i < navigator.plugins.length; i++)
      {
         if (navigator.plugins[i].name.toLowerCase().indexOf(plug_in) != -1)
         {
            if (navigator.plugins[i] == plugInOfRecord)
            {
               return true;
            }
         }
      }
   }
   return false;
}
```

This function starts by calling the `mimeIsReady()` function from Listing 42-3A. After that, the function resembles the one in Listing 42-3B until you reach the most nested statements. Here, instead of looking for any old `mime` type, you insist on the existence of an explicit match between the `mime` type passed as a parameter and an enabled `mime` type associated with the plug-in. To see how these functions work on your NN3+/Moz, MacIE5, or Safari browser, open the complete file (`jsb42-03.html`) from this CD-ROM. The actual listing also includes code that branches around IE for Windows and other browsers that don't support this way of inspecting `mime` types and plug-ins.

Managing manual plug-in installation

If your scripts determine that a visitor does not have the plug-in your data expects, you may want to consider providing an electronic guide to installing the plug-in. One way to do this is to open a new frameset (in the main window). One frame can contain step-by-step instructions with links to the plug-in's download site. The download site's page can appear in the other frame of this temporary window. The steps must take into account all installation requirements for every platform, or, alternatively, you can create a separate installation document for each unique class of platform. For instance, you must frequently decode Macintosh files from binhex format

and then uncompress them before you move them into the plug-ins folder. Some plug-ins have their own, separate installation program. The final step should include a call to

```
navigator.plugins.refresh()
```

to make sure that the browser updates its internal listings. After that, the script can return to the `document.referrer`, which should be the page that sends the visitor to the installation pages. All in all, the process is cumbersome — it's not like downloading a Java applet. But if you provide some guidance, there's a better chance that the user will return to play your cool media. Also consider letting the browser's own updating facilities handle the job (albeit not as smoothly in many cases) by simply loading the data into the page, ready or not.

"Plug-in" detection in WinIE

WinIE4+ provides some built-in facilities that may take the place of plug-in detection in some circumstances. First of all, it's important to recognize that WinIE does not use the term "plug-in" in the same way that Netscape and other browsers use it. Due to the integration between IE and the Windows operating system, WinIE employs system-wide ActiveX controls to handle the job of rendering external content. Some of these controls are designed to be accessed from outside their walls, thus allowing client-side scripts to get and set properties or invoke methods built into the controls. These controls behave a lot like plug-ins, so you frequently see them referenced as "plug-ins," as they are in this book.

WinIE prefers the `<object>` tag for both loading the plug-in (ActiveX control) and assigning external content to it. One of the attributes of the `object` element is `classid`, which points to a monstrously long string of hexadecimal numbers known as the `guid` (Globally Unique Identifier). When the browser encounters one of these `guids`, it looks into the Windows Registry to get the path to the actual plug-in file. If the plug-in is not installed on the user's machine, the object doesn't load, and any other HTML nested inside the `<object>` tag renders instead. Thus, you can display a static image placeholder or HTML message about the lack of the plug-in. But plug-in detection comes in most handy when your scripts need to communicate with the plug-in, such as directing an embedded Windows Media Player plug-in to change sound files or to play. When you build code around a scriptable plug-in, your scripts should make sure that the plug-in object is indeed present so they don't generate errors.

The idea of using the `<object>` tag instead of the older `<embed>` tag is that the `<object>` tag loads a specific plug-in, whereas the `mime` type of the data referenced by the `<embed>` tag lets the browser determine which plug-in to use for that `mime` type. It's not uncommon, therefore, to see an `<object>` tag definition surround an `<embed>` tag — both referencing the same external data file. If the optimum plug-in fails to load, the `<embed>` tag is observed, and the browser tries to find any plug-in for the file's `mime` type. This two-tiered approach to referencing a plug-in is used primarily for backward compatibility with version 4 browsers, whereas pages designed exclusively for modern browsers typically utilize the `<object>` tag alone.

With an `object` element as part of the HTML page, the element itself is a valid object — even if the plug-in fails to load. Therefore, you must do more to validate the existence of the loaded

plug-in than simply test for the existence of the `object` element. To that end, you need to know at least one scriptable property of the plug-in. Unfortunately, not all scriptable plug-ins are fully documented, so you occasionally must perform some detective work to determine which scriptable properties are available. While you're on the search for clues, you can also determine the version of the plug-in and set the minimum version that your `object` element allows to load.

Tracking down plug-in details

Not everyone has access to the Microsoft programming development environments (for example, Visual Studio) through which you can find out all kinds of information about an installed ActiveX control. If you don't have access, you can still dig deep to get most (if not all) of the information you need. The tools you can use include the Windows Registry Editor (`regedit`), The Evaluator, and, of course, your text editor and WinIE4+ browser. The following steps take you through finding out everything you need to know about the Windows Media Player control:

1. If you don't know the `guid` for the Media Player (most people get it by copying someone else's code that employs it), you can use the Registry Editor (`regedit.exe`) to find it. Open the Registry Editor. In Win95/98/NT/XP/Vista, choose Run from the Start menu and enter `regedit`; if that option is not available in your Windows version, search for the file named `regedit`.
2. Expand the `HKEY_CLASSES_ROOT` folder.
3. Scroll down to the nested folder named `CLSID`, and click that folder.
4. Choose Edit/Find, and enter `Windows Media Player`. If you were searching for a different plug-in, you would enter an identifying name (usually the product name) in this place.
5. Keep pressing F3 (Find Next) until the editor lands upon a folder whose default value (in the right side of the Registry Editor window) shows `Windows Media Player`.
6. The number inside curly braces next to the highlighted folder is the plug-in's `guid`. Right-click the number and choose Copy Key Name. Paste the number into your document somewhere for future reference. Eventually, it will be part of the value assigned to the `classid` attribute of the `object` element.
7. Expand the highlighted folder.
8. Click the folder named `InprocServer32`. The default value should show a pathname to the actual ActiveX control for the Windows Media Player plug-in.
9. Right-click the (`Default`) name for the path and choose Modify. The full pathname is visible in an editable field.
10. Armed with this pathname information, open My Computer and locate the actual file inside a directory listing.
11. Right-click the file and choose Properties.
12. Click the Version tab (if present).

13. Copy the version number (generally four sets of numbers delimited by commas), and paste it into your document for future reference. Eventually, it will be assigned to the `codebase` attribute of the `object` element.

    You are now ready to try loading the plug-in as an object and look for properties you can test for.

14. Add an `object` tag to The Evaluator source code. This can go inside the `head` or just before the `</body>` tag. For example, your tag should look something like the following:

```
<object id="wmp" width="1" height="1"
        classid="CLSID:0A4286EA-E355-44FB-8086-AF3DF7645BD9"
        codebase="#Version=10,0,0,3802">
</object>
```

    Copy and paste the numbers for the `guid` and version. Two points to watch out for: First, be sure that the `guid` value is preceded by `CLSID:` in the value assigned to `classid`; second, be sure the version numbers are preceded by the prefix shown.

15. Load (or reload) the page in WinIE4+.

    At this point, the `wmp` object should exist. If the associated plug-in loads successfully, the `wmp` object's properties include properties exposed by the plug-in.

16. Enter `wmp` into the bottom text box to inspect properties of the `wmp` object. Be patient: It may take many seconds for the retrieval of all properties.

    In case you can't readily distinguish between the `object` element object properties and properties of the scriptable plug-in, scroll down to the `wmp.innerHTML` property and its values. When an object loads successfully, any parameters that it accepts are reflected in the `innerHTML` for the `object` element. Each `param` element has a name — the name of one of the scriptable properties of the plug-in.

17. Look for one of the properties that has some kind of value by default (in other words, other than an empty string or `false`). In Windows Media Player, this can be `CreationDate`. Use this property as an object detection condition in scripts that need to access the Windows Media Player properties or methods:

```
if (wmp && wmp.CreationDate)
{
   // statements that "talk to" plug-in
}
```

Setting a minimum version number

The four numbers that you grab in step 13 in the previous section represent the version of the plug-in as installed on your computer. Unless you have a way of verifying that your external content runs on earlier versions of the plug-in (if there are earlier versions), you can safely specify *your* version as the minimum.

Specific rankings for the four numbers of a version decrease as you move from left to right. For example, version 9,0,25,2 is later than 9,0,0,0; version 10,0,0,0 is later than both of them. If you specify 9,0,25,2, and the user has 9,0,24,0 installed, the plug-in does not load, and the

object isn't available for scripting. On the other hand, a user with 9,0,26,0 has the object present because the `codebase` attribute for the version specifies a minimum allowable version to load.

When an object requires VBScript

Not all objects that load via the `object` element are scriptable through JavaScript (or JScript, in this case). Occasionally, an object is designed so that its properties are exposed only to VBScript. This happens, for example, with the Microsoft Windows Media Rights Manager (DRM) object. To find out if the browser (and thus, the operating system) is equipped with DRM, your page loads the object via the `object` element as usual; however, a separate VBScript section must access the object to test for the existence of one of its properties. Because script segments written in either language can access each other, this isn't a problem, provided you know what the property or method is for the object. The following fragment from the `head` section of a document demonstrates how JavaScript and VBScript can interact so that JavaScript code can branch based on the availability of DRM:

```
<head>
   <object id="drmObj" height="1" width="1"
           classid="CLSID:760C4B83-E211-11D2-BF3E-00805FBE84A6"></object>
   <script type="text/vbscript">
      function hasDRM()
         on error resume next
         drmObj.StoreLicense("")
         if (err.number = 0) then
            hasDRM = true
         else
            hasDRM = false
         end if
      end function
   </script>
   <script type="text/javascript">
      var gHasDRM;
         if (drmObj && hasDRM())
         {
            gHasDRM = true;
         }
         else
         {
            gHasDRM = false;
         }
   </script>
</head>
```

The JavaScript segment sets a Boolean global variable to indicate whether the object has loaded correctly. Part of the job is accomplished via the `hasDRM()` function in the VBScript segment. From VBScript, the `drmObj` object responds to the `StoreLicense()` method call, but it throws a VBScript error indicating that no parameter was sent along with the method. Any subsequent scripts in this page can use the `gHasDRM` global variable as a conditional expression before performing any actions requiring the object (which works in tandem with the Windows Media Player).

screen Object

Properties	Methods	Event Handlers
availHeight		
availLeft		
availTop		
availWidth		
bufferDepth		
colorDepth		
fontSmoothingEnabled		
height		
pixelDepth		
updateInterval		
width		

Syntax

Accessing screen object properties:

```
(All)    screen.property
(IE4+)   [window.]navigator.screen.property
```

Compatibility: WinIE4+, MacIE4+, NN4+, Moz+, Safari+, Opera+, Chrome+

About this object

Browsers other than from the earliest generations provide a `screen` object that lets your scripts inquire about the size and color settings of the video monitor used to display a page. Properties are carefully designed to reveal not only the raw width and height of the monitor (in pixels), but also what the available width and height are once you take into account the operating system's screen-hogging interface elements (for example, the Windows taskbar and the Mac menu bar).

Internet Explorer 4 provides a `screen` object, although it appears as a property of the `window` object in the IE4 through IE6 object model. Only five properties of the IE4-6 `screen` object — `height`, `availHeight`, `width`, `availWidth`, and `colorDepth` — share the same syntax as NN4+'s `screen` object. However, as of IE7 you can now use the NN/Moz approach of accessing the `screen` object without the `window` object, as in `screen.availWidth`.

Properties

`availHeight`
`availWidth`
`height`
`width`

Value: Integer Read-Only

Compatibility: WinIE4+, MacIE4+, NN4+, Moz+, Safari+, Opera+, Chrome+

With the availability of window sizing methods in modern browsers, your scripts may want to know how large the user's monitor is. This is particularly important if you set up an application to run in kiosk mode, which occupies the entire screen. Two pairs of properties let scripts extract the dimensions of the screen. All dimensions are in pixels.

You can extract the gross height and width of the monitor from the `screen.height` and `screen.width` properties. Thus, a monitor rated as a 1024×768 monitor returns values of 1024 and 768 for `width` and `height`, respectively.

But not every pixel of the screen's gross size is available as displayable area for a window. To the rescue come the `screen.availWidth` and `screen.availHeight` properties. For example, modern Windows operating systems display the taskbar. The default location for this bar is at the bottom of the window, but users can reorient it along any edge of the screen. If the default behavior of always showing the taskbar is in force, the bar uses up real estate available for window display (unless you intentionally size or position a window so that part of the window extends under the bar). When along the top or bottom edge of the screen, the taskbar occupies 28 vertical pixels; when positioned along one of the sides, the bar occupies 60 horizontal pixels. On the Macintosh platform, the 22-pixel-deep menu bar occupies a top strip of the screen. Unlike previous versions, however, Mac OS X does not let you position a window behind the menu bar.

You can use the available screen size values as settings for window properties. For example, to arrange a window so that it occupies all available space on the monitor, you must position the window at the top left of the screen and then set the outer window dimensions to the available sizes, as follows:

```
function maximize()
{
   window.moveTo(0,0);
   window.resizeTo(screen.availWidth, screen.availHeight);
}
```

The preceding function positions a Safari window appropriately on the Macintosh, just below the menu bar, so that the menu bar does not obscure the window. If, however, the client is running Windows and the user positions the taskbar at the top of the screen, the window is partially

hidden under the taskbar (you cannot query the available screen space's coordinates). Also in Windows, the appearance is not exactly the same as a maximized window. See the discussion of the `window.resizeTo()` method in Chapter 27 for more details. Note that MacIE generally returns a value for `screen.availHeight` that is about 24 pixels fewer than the actual available height (even after taking into account the Mac menu bar).

For Navigator 3+ and Mozilla, you can use the NPAPI to access a native Java class that reveals the overall screen size (not the available screen size). If Java is enabled, you can place the following script fragment in the Head portion of your document to set variables with screen width and height:

```
var toolkit = java.awt.Toolkit.getDefaultToolkit();
var screenSize = toolkit.getScreenSize();
```

The `screenSize` variable is an object whose properties (`width` and `height`) contain the pixel measures of the current screen. This NPAPI technique works only in Netscape and Mozilla browsers (IE does not provide direct access to Java classes). In fact, you can also extract the screen resolution (pixels per inch) in the same manner. The following statement, added after the preceding ones, sets the variable resolution to that value:

```
var resolution = toolkit.getScreenResolution();
```

Related Items: `window.innerHeight`, `window.innerWidth`, `window.outerHeight`, `window.outerWidth` properties; `window.moveTo()`, `window.resizeTo()` methods

availLeft
availTop

Value: Integer Read-Only

Compatibility: WinIE-, MacIE-, NN4+, Moz+, Safari+, Opera-, Chrome+

The `availLeft` and `availTop` properties return the pixel measure of where the available space of the screen begins. The only time these values are anything other than zero is when a user positions the taskbar along the left or top edges of the screen. For example, if the user positions the taskbar along the top of the screen, you do not want to position a window any higher than the 28 pixels occupied by the taskbar. There are no corresponding properties for IE.

Example

If you are a Windows user, you can experiment with these NN4+/Moz properties via The Evaluator (Chapter 4, "JavaScript Essentials"). With the taskbar at the bottom of the screen, enter these two statements into the top text box:

```
screen.availLeft
screen.availTop
```

Next, drag the taskbar to the top of the screen and try both statements again. Now, drag the taskbar to the left edge of the screen and try the statements once more.

Related Items: `screen.availWidth`, `screen.availHeight` properties; `window.moveTo()` method

bufferDepth

Value: Integer Read/Write

Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-, Opera-, Chrome-

By default, IE does not use any offscreen buffering of page content. But adjusting the `bufferDepth` property enables you to turn on offscreen buffering and control the color depth of the buffer. Using offscreen buffering may improve the smoothness of path-oriented animation through positioning.

The default value (buffering turned off) is `0`. By setting the property to `-1`, you instruct IE to set the color depth of the offscreen buffer to the same color depth as the screen (as set in the control panel). This should be the optimum value, but you can also force the offscreen buffer to have one of the following bit depths: `1`, `4`, `8`, `15`, `16`, `24`, or `32`.

Related Items: `screen.colorDepth`, `screen.pixelDepth` properties

colorDepth
pixelDepth

Value: Integer Read-Only

Compatibility: WinIE4+, MacIE4+, NN4+, Moz+, Safari+, Opera+, Chrome+

You can design a page with different color models in mind because your scripts can query the client to find out how many colors the user sets the monitor to display. This is helpful if you have more subtle color schemes that require 32-bit color settings, or images tailored to specific palette sizes.

Both the `screen.colorDepth` and `screen.pixelDepth` properties return the number of color bits to which the client computer's video display control panel is set. The `screen.colorDepth` value may take into account a custom color palette; so, for NN4+, you may prefer to rely only on the `screen.pixelDepth` value. (IE4+, however, supports only the `screen.colorDepth` property of this pair.) You can use this value to determine which of two image versions to load, as shown in the following script fragment that runs as the document loads:

```
if (screen.colorDepth > 8 )
{
   document.write("<img src='logoHI.jpg' height='60' width='100' />");
}
else
{
   document.write("<img src='logoLO.jpg' height='60' width='100' />");
}
```

In this example, the `logoHI.jpg` image is designed for 16-bit displays or better, whereas the colors in `logoLO.jpg` are tuned for 8-bit display.

Related Item: `screen.bufferDepth` property

fontSmoothingEnabled

Value: Boolean Read-Only

Compatibility: WinIE4+, MacIE-, NN-, Moz-, Safari-, Opera-, Chrome-

Some versions of the Windows OS have a Display control panel setting for "Smooth Edges" on screen fonts. The `fontSmoothingEnabled` property lets your script see the state of that setting. This setting can affect, for example, which stylesheet you enable because it has font specifications that look good only when smoothing is enabled. A default installation of Windows has this feature turned off.

updateInterval

Value: Integer Read/Write

Compatibility: WinIE4+, MacIE-, NN-, Moz-, Safari-, Opera-, Chrome-

The `updateInterval` property is the number of milliseconds between screen updates. The default value of zero lets IE arbitrate among the demands for screen updates in a highly animated setting. If you set this value to a large number, more screen updates are accumulated in a buffer — preventing some animated steps from updating the screen.

userProfile Object

Properties	Methods	Event Handlers
	`addReadRequest()`	
	`clearRequest()`	
	`doReadRequest()`	
	`getAttribute()`	

Syntax

Accessing `userProfile` object methods:

```
(IE4+)  [window.]navigator.userProfile.method()
```

Compatibility: WinIE-6, MacIE-, NN-, Moz-, Safari-, Opera-, Chrome-

About this object

The `userProfile` object is an IE-specific (and Windows, at that) property that acts as the gateway to the user profile information that the client computer collects from the user. You can retrieve none of this information via JavaScript without permission from the user. Access to this information is performed in a strict sequence, part of which enables you to define how the request for this private information is worded when the user is presented with the request. As of IE7, this object no longer appears to be accessible, thereby relegating its usage to IE versions 4 through 6.

User profile data consists of nearly 30 fields of personal information about the user's contact information. Each of these fields has a name, which by and large conforms to the vCard standard. Your scripts can request one or more specific fields from the list, rather than having to deal with the entire set of fields.

The sequence for accessing this data entails four basic steps:

1. Put the request for each vCard field into a queue that is maintained in the browser's memory (via the `addReadRequest()` method).
2. Execute the batch request, which displays a detailed dialog box to the user (via the `doReadRequest()` method). If a user profile is in effect, the user sees which fields you are requesting, plus the data in the vCard. The user then has the chance to deselect one or more of your choices — or to disallow access completely.
3. Get each attribute by name (via the `getAttribute()` method). You invoke this method once for each vCard field.
4. Clear the queue of requests (via the `clearRequest()` method).

Returned values are strings. Thus, you can prefill the customer information for an order form or capture the information in hidden fields that are submitted with a visible form.

Listing 42-4 demonstrates the use of the four key methods of the `userProfile` object. After the page loads, it attempts to extract the data from every vCard field and displays both the attribute name and the value, as associated with the current user profile, in a table. Notice that the names of the attributes are hard-wired because the object does not provide a list of implemented attributes.

LISTING 42-4

Accessing userProfile Data

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>userProfile Object</title>
      <script type="text/javascript">
         var attrs = ["Business.City","Business.Country","Business.Fax",
                      "Business.Phone","Business.State",
                      "Business.StreetAddress","Business.URL",
                      "Business.Zipcode","Cellular","Company","Department",
                      "DisplayName","Email","FirstName","Gender","Home.City",vfill
```

continued

LISTING 42-4 *(continued)*

```
                  "Home.Country","Home.Fax","Home.Phone","Home.State",
                  "Home.StreetAddress","Home.Zipcode","Homepage",
                  "JobTitle","LastName","MiddleName","Office","Pager"];
         function loadTable()
         {
            // make sure this executes only in IE4+ for Windows
            if ((navigator.userAgent.indexOf("Win") != -1) && navigator.userProfile)
            {
               var newRow, newCell, attrValue;
               // queue up requests for every vCard attribute
               for (var i = 0; i < attrs.length; i++)
               {
                  navigator.userProfile.addReadRequest("vCard." + attrs[i]);
               }
               // dispatch the request to let user accept or deny access
               navigator.userProfile.doReadRequest(1, "JavaScript Bible");
               // append rows to the table with attribute/value pairs
               for (var j = 0; j < attrs.length; j++)
               {
                  newRow = document.all.attrTable.insertRow(-1);
                  newRow.bgColor = "#FFFF99";
                  newCell = newRow.insertCell(0);
                  newCell.innerText = "vCard." + attrs[j];
                  newCell = newRow.insertCell(1);
                  // get the actual value
                  attrValue = navigator.userProfile.getAttribute
                     ("vCard." + attrs[j]);
                  newCell.innerHTML = (attrValue) ? attrValue : " ";
               }
               // clean up after ourselves
               navigator.userProfile.clearRequest();
            }
            else
            {
               alert("This example requires IE4+ for Windows.");
            }
         }
      </script>
   </head>
   <body onload="loadTable()">
      <h1>userProfile Object</h1>
      <hr />
      <table id="attrTable" border="1" cellpadding="5">
         <tr bgcolor="#CCFFFF">
            <th>vCard Property</th>
            <th>Value</th>
         </tr>
      </table>
   </body>
</html>
```

Note

As handy as it may be, in a strict W3C approach to JavaScript, you wouldn't use the `innerHTML` property since it isn't officially part of the W3C standard. However, it is often too powerful a convenience property to ignore, as much of the code throughout this book is a testament. The book does show the W3C node manipulation alternative to `innerHTML` in some examples. Refer to Chapter 29, "Document and Body Objects," for a thorough explanation and examples of the W3C alternative to `innerHTML`. ■

It appears that the newer the version of Windows that the user runs, the more likely that user profile data is available. Even so, there may be little more than name and address data for those users who are careful not to fill out optional fields of Microsoft web site forms requesting personal information.

Comparable information may be available from NN4+/Moz users on any OS platform, via signed scripts that access `ldap` preferences. See the discussion earlier in this chapter about the `navigator.preference()` method.

Methods

addReadRequest("*attributeName*")

Returns: Boolean

Compatibility: WinIE-6, MacIE-, NN-, Moz-, Safari-, Opera-, Chrome-

Before the user is asked for permission to reveal any personal information, you must queue up requests — even if there is just one field in which you are interested. For each field, use the `addReadRequest()` method and specify a string of the attribute name as the parameter. Acceptable attribute names are as follows:

```
vCard.Business.City
vCard.Business.Country
vCard.Business.Fax
vCard.Business.Phone
vCard.Business.State
vCard.Business.StreetAddress
vCard.Business.URL
vCard.Business.Zipcode
vCard.Cellular
vCard.Company
vCard.Department
vCard.DisplayName
vCard.Email
vCard.FirstName
vCard.Gender
vCard.Home.City
vCard.Home.Country
vCard.Home.Fax
vCard.Home.Phone
vCard.Home.State
```

```
vCard.Home.StreetAddress
vCard.Home.Zipcode
vCard.Homepage
vCard.JobTitle
vCard.LastName
vCard.MiddleName
vCard.Office
vCard.Pager
```

All attribute values are case-insensitive.

This method returns a Boolean value of `true` if the addition to the queue succeeds. A returned value of `false` usually means that the attribute value is not valid or that a request for that attribute name is already in the queue. If you fail to clear the queue after compiling one list of attributes, attempts to read the attribute result in a return value of `false`.

Example

See Listing 42-4 for an example of the `addReadRequest()` method in action. You can also invoke it from the top text box in The Evaluator. For example, enter the following statement to queue one request:

```
navigator.userProfile.addReadRequest("vCard.LastName")
```

To continue the process, see examples for `doReadRequest()` and `getAttribute()` later in this chapter.

Related Items: `clearRequest()`, `doReadRequest()`, and `getAttribute()` methods

clearRequest()

Returns: Nothing

Compatibility: WinIE-6, MacIE-, NN-, Moz-, Safari-, Opera-, Chrome-

After retrieving the attributes whose names are stacked in the request queue, invoke the `clearRequest()` method to empty the queue. It is always good programming practice to clean up after yourself, especially when security concerns are involved.

Related Items: `addReadRequest()`, `doReadRequest()`, and `getAttribute()` methods

doReadRequest(*reasonCode*, *identification*[, *domain*[, *path*[, *expiration*]]])

Returns: Nothing

Compatibility: WinIE-6, MacIE-, NN-, Moz-, Safari-, Opera-, Chrome-

Once the names of the desired vCard attributes are stacked in the queue (via the `addReadRequest()` method), invoke the `doReadRequest()` method to prompt the user

for the permission that your scripts need to gain access to the data. The user sees a detailed dialog box that lists the vCard fields you are requesting, as well as a description about your reason for wanting the data and who you are.

The first required parameter is an integer representing one of the standard descriptions as defined by the Internet Privacy Working Group. Associated text is displayed in the permission request dialog box that the user sees. The codes and their strings are as follows:

Code	Description String
0	Used for system administration.
1	Used for research and/or product development.
2	Used for completion and support of current transaction.
3	Used to customize the content and design of a site.
4	Used to improve the content of the site, including advertisements.
5	Used for notifying visitors about updates to the site.
6	Used for contacting visitors for marketing of services or products.
7	Used for linking other collected information.
8	Used by site for other purposes.
9	Disclosed to others for customization or improvement of the content and design of the site.
10	Disclosed to others, who may contact you, for marketing of services and/or products.
11	Disclosed to others, who may contact you, for marketing of services and/or products; you have the opportunity to ask a site not to do this.
12	Disclosed to others for any other purpose.

Although these description strings are fixed, you do have an opportunity to include some customized information in the second parameter. This parameter is intended to enable you to identify the web site or organization requesting the information. Standards recommendations suggest you include a URL to the site as well. In any case, the second parameter can be any string. But it is not treated like HTML, so do not attempt to include a clickable link here.

Two optional parameters enable you to specify a domain and path within that domain for which the user permissions are to apply. Both of these parameters closely mirror their usage in cookies, but they also depend on the capability to set an expiration date via the fifth parameter. Through IE6, however, the expiration date parameter is ignored. Therefore, permissions expire when the user quits the browser (just like temporary cookies do).

Example

See Listing 42-4 for an example of the `doReadRequest()` method in action. If you entered the `addReadRequest()` example for The Evaluator earlier in this chapter, you can now bring up the permissions dialog box (if you have a user profile for your version of Windows) by entering the following statement into the top text box:

```
navigator.userProfile.doReadRequest(1, "Just me!")
```

Related Items: `addReadRequest()`, `clearRequest()`, and `getAttribute()` methods

getAttribute("attributeName")

Returns: String

Compatibility: WinIE-6, MacIE4+, NN-, Moz-, Safari-, Opera-, Chrome-

The `getAttribute()` method attempts to retrieve the vCard data based on the items queued via the `addReadRequest()` method. A permission dialog box provides the user an opportunity to choose which of the requested items to reveal, or to deny all access to the information. Only one attribute name is permitted as a parameter to the `getAttribute()` method, requiring that you invoke the method for each attribute you want to fetch.

Example

See Listing 42-4 for an example of the `getAttribute()` method in action. Also, if you followed The Evaluator examples for this object, you can now extract the desired information (provided it is in your user profile). Enter the following statement into the top text box:

```
navigator.userProfile.getAttribute("vCard.LastName")
```

Related Items: `addReadRequest()`, `clearRequest()`, and `doReadRequest()` methods

CHAPTER

Positioned Objects

IN THIS CHAPTER

Layer concepts

Moving, hiding, and showing content

Layering objects in the modern DOM

This chapter tackles positioned objects and layers, briefly acknowledging the early contribution that Netscape Navigator 4 made to the subject. Having survived a turbulent time of severe browser compatibility problems in this area, modern browsers have now adopted W3C standards for positioned content. Since current browsers explicitly do not provide backward compatibility with the original NN4 scripted `layerelement` object, this chapter won't do so either.

The modern solution to scripted layering is through Cascading Style Sheets (CSS) and the scripting thereof. This chapter therefore focuses on how to apply CSS and modern DOM techniques to manage positioned elements in modern browsers. When reading and working through this chapter, keep in mind something we pointed out in Chapter 38, "Style Sheet and Style Objects": although defining styles in style sheets is a best practice, if an element object presents a scriptable property that reflects an attribute for that element's tag, the first time a script tries to read that property, a value will be associated with that property *only* if the attribute is explicitly assigned in the HTML code. You'll notice in this chapter's listings, therefore, that we have put the styles we are not manipulating into a style sheet, and the ones we are into `style` attributes.

What Is a Layer?

Terminology in the area of positioned elements has become a bit confusing over time. Because NN4 was the earliest browser to be released with positioned elements (the `layer` element), the term *layer* became synonymous with any positioned element. When IE4 came on the scene, it was convenient to call a stylesheet-positioned element (in other words, an element governed by a stylesheet rule with the `position` attribute) a *layer*, as a

generic term for any positioned element. In fact, NN4 even treated an element that was positioned through style sheets as if it were a genuine layer object (although with some minor differences).

In the end, the layer term made good sense because no matter how it was achieved, a positioned element acted like a layer in front of the body content of a page. Perhaps you have seen how animated cartoons were created before computer animation changed the art. Layers of clear acetate sheets were assembled atop a static background. Each sheet contained one character or portion of a character. When all the sheets were carefully positioned on top of each other, the view (as captured by a still camera) formed a composite frame of the cartoon. To create the next frame of the cartoon, the artist moved one of the layers a fraction of an inch along its intended path and then took another picture.

If you can visualize how that operation works, you have a good starting point for understanding how layers work. Each layer contains some kind of HTML content that exists in its own plane above the main document that loads in a window. You can change or replace the content of an individual layer on-the-fly without affecting the layout of the other layers; you can also reposition, resize, or hide the entire layer under script control.

One aspect of layers that goes beyond the cartoon analogy is the ability to contain other layers. When that happens, any change that affects the primary layer — such as moving the layer 10 pixels downward — also affects the layers nested inside. It's as if the nested layers are passengers of the outer layer. When the outer layer goes somewhere, the passengers do, too. And yet, within the vehicle, the passengers may change seats by moving around without regard for what's going on outside.

With this analogy in mind, many commercial DHTML development tools and content authors refer to positioned elements as layers, which you can move, resize, stack, and hide independently of the body background. Therefore, even though this chapter focuses on style sheet–positioned elements, we'll continue to use the *layer* term in a generic sense.

Positioned Elements in the Modern DOM

Thanks to the tireless efforts of the W3C, modern browsers all share the `style` property of every renderable element object. Most adjustments to the location, layering, size, and visibility of positioned elements use the `style` object associated with each element. Even so, differences still exist between the IE and non-IE browsers with respect to the event objects — how to reference the `event` object and the names of its properties.

Changing element backgrounds

Listing 43-1 demonstrates the syntax and behavior of setting background images through the `style.backgroundImage` property. Note the CSS-style syntax for the URL value assigned to the `style.backgroundImage` property. It's a good lesson to learn that most `style` properties are strings, and their values are in the same format as the values normally assigned in a stylesheet definition.

Removing a background image requires setting the URL to an empty string or `null`. We use an empty string because that will work in all browsers, whereas a `null` will not. Also, a background image overlays whatever color (if any) you assign to the element. If the background image has transparent regions, the background color shows through, as you can see when you click the Not So Usual button. (Try experimenting with different background colors.)

LISTING 43-1

Setting Layer Backgrounds

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Layer Backgrounds</title>
      <style type="text/css">
         #bgExpo { position:absolute;
                   top:100px; left:250px;
                   width:300px; height:260px;
                   background-color:gray;
                 }
         #buttons { position:absolute; top:100px;
                  }
         #someText { font-weight:bold; color:white:
                   }
      </style>
      <script type="text/javascript">
         function setBg(URL)
         {
            document.getElementById("bgExpo").style.backgroundImage =
              "url(" + URL + ")";
         }
      </script>
   </head>
   <body>
      <h1>Layer Backgrounds</h1>
      <hr />
      <div id="buttons">
         <form>
            <input type="button" value="The Usual"
                   onclick="setBg('cr_kraft.gif')" /><br />
            <input type="button" value="A Big One"
                   onclick="setBg('arch.gif')" /><br />
            <input type="button" value="Not So Usual"
                   onclick="setBg('wh86.gif')" /><br />
            <input type="button" value="Decidedly Unusual"
                   onclick="setBg('sb23.gif')" /><br />
            <input type="button" value="Quick as..."
                   onclick="setBg('lightnin.gif')" />
            <p><input type="button" value="Remove Image"
                      onclick="setBg('')" /><br />
            </p>
```

continued

LISTING 43-1 *(continued)*

```
      </form>
    </div>
    <div id="bgExpo">
      <span id="someText">Some text, which may or
      may not read well with the various backgrounds.</span>
    </div>
  </body>
</html>
```

Note

The property assignment event handling technique employed throughout the code in most of this chapter is a deliberate simplification to make the code more readable. It is generally better to use the more modern approach of binding events using the `addEventListener()` (NN6+/Moz/W3C) or `attachEvent()` (IE5+) methods. A modern cross-browser event handling technique is explained in detail in Chapter 32, "Event Objects." Listings 43-9 and 43-10, later in the chapter, demonstrate how to use the modern technique to bind and unbind mouse events. ■

Listing 43-2 focuses on background color. A color palette is laid out as a series of rectangles. As the user rolls atop a color in the palette, the color is assigned to the background of the layer. Because of the regularity of the `div` elements generated for the palette, this example uses scripts to write them dynamically to the page as the page loads. This lets the `for` loop handle all the positioning math based on initial values set as global variables.

Perhaps of more interest here than the background color setting is the event handling. First of all, because the target browsers all employ event bubbling, the page lets a single event handler at the document level wait for `mouseover` events to bubble up to the document level. But because the `mouseover` event of every element on the page bubbles there, the event handler must filter the events and process only those on the palette elements.

The `setColor()` method begins by equalizing the IE4+ and W3C DOM event object models. If an object is assigned to the `evt` parameter variable, that means the W3C DOM browser is processing the event; otherwise, it's IE4+ — meaning that the `window.event` object contains the event information. Whichever browser performs the processing, the event object is assigned to the `evt` variable. After verifying that a valid event triggered the function, the next step is to equalize the different, event-model–specific property names for the event's target element. For W3C DOM browsers, the property is `target`, whereas IE4+ uses `srcElement`. The final validation is to check the `className` property of the event's target element. Because all elements acting as palette colors share the same `class` attribute, the `className` property is examined. If the value is `palette`, the `mouseover` event has occurred on one of the colors. Now it's time to extract the target element's `style.backgroundColor` property and assign that color to the same property of the main positioned element.

LISTING 43-2

Layer Background Colors

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Layer Background Colors</title>
      <style type="text/css">
         #display { position:absolute;
                    top:150px; left:80px;
                    width:200px; height:200px;
                    background-color:gray;
                  }
         #someText { font-weight:bold; color:white;
                     text-align:center;
                   }
      </style>
      <script type="text/javascript">
         function setColor(evt)
         {
            evt = (evt) ? evt : (window.event) ? window.event : "";
               if (evt)
               {
                  var elem = (evt.target) ? evt.target : evt.srcElement;
                     if (elem.className == "palette")
                     {
                        document.getElementById("display").style.backgroundColor =
                           elem.style.backgroundColor;
                     }
               }
         }
         document.onmouseover = setColor;
      </script>
   </head>
   <body>
      <h1>Layer Background Colors</h1>
      <hr />
      <script type="text/javascript">
         var oneLayer;
         var colorTop = 100;
         var colorLeft = 20;
         var colorWidth = 40;
         var colorHeight = 40;
         var colorPalette = new Array("aquamarine","coral","forestgreen",
                "goldenrod","red","magenta","navy","teal");
         for (var i = 0; i < colorPalette.length; i++)
         {
           oneLayer = "<div id='swatch" + i + "' class='palette' ";
           oneLayer += "style='position:absolute; top:" + colorTop + "px; ";
```

continued

LISTING 43-2 *(continued)*

```
          oneLayer += "left:" + ((colorWidth * i) + colorLeft) + "px; ";
          oneLayer += "width:" + colorWidth + "px; height:" + colorHeight + "px; ";
          oneLayer += "background-color:" + colorPalette[i] + ";'><\/div>\n";
          document.write(oneLayer);
        }
      </script>
      <div id="display">
        <span id="someText">Some
        reversed text to test against background colors.</span>
      </div>
   </body>
</html>
```

Layer clipping

Working with clipping rectangles is a bit cumbersome using CSS syntax because the object model standard does not provide separate readouts or controls over individual edges of a clipping rectangle. IE5+ enables you to read individual edge dimensions via the `currentStyle` object (for example, `currentStyle.clipTop`), but these properties are read-only.

Based on these limitations, Listing 43-3 is implemented in a way that, for the sake of convenience, preserves the current clipping rectangle edge values as global variables. Any adjustments to individual edge values are first recorded in those variables (in the `setClip()` function), and then the `style.clip` property is assigned the long string of values in the required format (in the `adjustClip()` function). The `showValues()` function reads the variable values and displays updated values after making the necessary calculations for the width and height of the clipping rectangle.

As a demonstration of a “reveal” visual effect (which you can carry out more simply in WinIE4+ via a transition filter), the `revealClip()` function establishes beginning clip values at the midpoints of the width and height of the layer. Then the `setInterval()` method loops through `stepClip()` until the clipping rectangle dimensions match those of the layer.

LISTING 43-3

Adjusting Layer Clip Properties

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Layer Clip</title>
      <style type="text/css">
         #form4clipValues
         {
```

```
         position:absolute; left:10px; top:130px;
      }
      #display
      {
         position:absolute;top:130px; left:220px;
         clip:rect(0px 360px 180px 0px);
         background-color:coral;
      }
   </style>
   <script type="text/javascript">
      var origLayerWidth = 0
      var origLayerHeight = 0
      var currTop, currRight, currBottom, currLeft
      function init()
      {
         origLayerWidth = 360;
         origLayerHeight = 180;
         currTop = 0;
         currRight = origLayerWidth;
         currBottom = origLayerHeight;
         currLeft = 0;
         showValues();
      }

      function setClip(field)
      {
         var val = parseInt(field.value);
         switch (field.name)
         {
            case "top" :
               currTop = val;
               break;
            case "right" :
               currRight = val;
               break;
            case "bottom" :
               currBottom = val;
               break;
            case "left" :
               currLeft = val;
               break;
            case "width" :
               currRight = currLeft + val;
               break;
            case "height" :
               currBottom = currTop + val;
               break;
         }
         adjustClip();
         showValues();
      }
```

continued

LISTING 43-3 *(continued)*

```
function adjustClip()
{
   document.getElementById("display").style.clip = "rect(" + currTop +
                                                   "px " + currRight +
                                                   "px " + currBottom +
                                                   "px " + currLeft +
                                                   "px)";
}

function showValues()
{
   var form = document.forms[0];
   form.top.value = currTop;
   form.right.value = currRight;
   form.bottom.value = currBottom;
   form.left.value = currLeft;
   form.width.value = currRight - currLeft;
   form.height.value = currBottom - currTop;
}
var intervalID;
function revealClip()
{
   var midWidth = Math.round(origLayerWidth /2);
   var midHeight = Math.round(origLayerHeight /2);
   currTop = midHeight;
   currBottom = midHeight;
   currRight = midWidth;
   currLeft = midWidth;
   intervalID = setInterval("stepClip()",1);
}
function stepClip()
{
   var widthDone = false;
   var heightDone = false;
      if (currLeft > 0)
      {
         currLeft += -2;
         currRight += 2;
      }
      else
      {
         widthDone = true;
      }
      if (currTop > 0)
      {
         currTop += -1;
         currBottom += 1;
      }
```

```
            else
            {
               heightDone = true;
            }
         adjustClip();
         showValues();
         if (widthDone && heightDone)
         {
            clearInterval(intervalID);
         }
      }
   </script>
</head>
<body onload="init()">
   <h1>Layer Clipping Properties</h1>
   <hr />
   Enter new clipping values to adjust the visible area of the layer.
   <div id="form4clipValues">
      <form>
         <table>
            <tr>
               <td align="right">layer.style.clip (left):</td>
               <td>
                  <input type="text" name="left" size="3"
                         onchange="setClip(this)" />
               </td>
            </tr>
            <tr>
               <td align="right">layer.style.clip (top):</td>
               <td>
                  <input type="text" name="top" size="3"
                         onchange="setClip(this)" />
               </td>
            </tr>
            <tr>
               <td align="right">layer.style.clip (right):</td>
               <td>
                  <input type="text" name="right" size="3"
                         onchange="setClip(this)" />
               </td>
            </tr>
            <tr>
               <td align="right">layer.style.clip (bottom):</td>
               <td>
                  <input type="text" name="bottom" size="3"
                         onchange="setClip(this)" />
               </td>
            </tr>
            <tr>
               <td align="right">layer.style.clip (width):</td>
               <td>
```

continued

LISTING 43-3 *(continued)*

```
                    <input type="text" name="width" size="3"
                           onchange="setClip(this)" />
                 </td>
              </tr>
              <tr>
                 <td align="right">layer.style.clip (height):</td>
                 <td>
                    <input type="text" name="height" size="3"
                           onchange="setClip(this)" />
                 </td>
              </tr>
           </table>
           <input type="button" value="Reveal Original Layer"
                  onclick="revealClip()" />
        </form>
     </div>
     <div id="display">
        <h2>ARTICLE I</h2>
        <p>Congress shall make no law respecting an establishment of religion,
           or prohibiting the free exercise thereof; or abridging the freedom
           of speech, or of the press; or the right of the people peaceably to
           assemble, and to petition the government for a redress of
           grievances.
        </p>
     </div>
  </body>
</html>
```

Listing 43-4 enables you to compare the results of adjusting a clipping rectangle versus the size of a positioned element. This example is interesting in that it enables you to adjust the dimensions of the entire layer (through the `style.left` and `style.right` properties), as well as the right and bottom edges of the clipping rectangle associated with the layer. Additionally, the code includes a function that converts the `style.clip` string into an object representing the rectangle of the clipping rectangle (in other words, with four properties, one for each edge). Values from that `rectangle` object populate two of the fields on the page, providing dynamic readouts of the clipping rectangle's right and bottom edges.

Global variables temporarily store the clipping rectangle values so that the `adjustClip()` function can operate just as it does in Listing 43-3. Note that the clipping rectangle is explicitly defined in the style sheet rule for the positioned element. This is necessary for the element's `style.clip` property to have some values with which to start. Lastly, note the differences in how the different browsers render the results (see Chapter 38, "Style Sheet and Style Objects").

LISTING 43-4

Comparison of Layer and Clip Location Properties

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Layer vs. Clip</title>
      <style type="text/css">
         #form4clipValues
         {
            position:absolute; left:10px; top:130px;
         }
         #display
         {
            position:absolute;
            background-color:coral;
         }
      </style>
      <script type="text/javascript">
         var currClipTop = 0;
         var currClipLeft = 0;
         var currClipRight = 360;
         var currClipBottom = 180;
         function setClip(field)
         {
            var val = parseInt(field.value);
            switch (field.name)
            {
               case "clipBottom" :
                  currClipBottom = val;
                  break;
               case "clipRight" :
                  currClipRight = val;
                  break;
            }
            adjustClip();
            showValues();
         }
         function adjustClip()
         {
            document.getElementById("display").style.clip = "rect(" + currClipTop +
                                                            "px " + currClipRight +
                                                            "px " + currClipBottom +
                                                            "px " + currClipLeft +
                                                            "px)";
         }
```

continued

LISTING 43-4 *(continued)*

```
         function setLayer(field)
         {
            var val = parseInt(field.value);
            switch (field.name)
            {
               case "width" :
                  document.getElementById("display").style.width = val + "px";
                  break;
               case "height" :
                  document.getElementById("display").style.height = val + "px";
                  break;
            }
            showValues();
         }
         function showValues()
         {
            var form = document.forms[0];
            var elem = document.getElementById("display");
            var clipRect = getClipRect(elem);
            form.width.value = parseInt(elem.style.width);
            form.height.value = parseInt(elem.style.height);
            form.clipRight.value = clipRect.right;
            form.clipBottom.value = clipRect.bottom;
         }
         // convert clip property string to an object
         function getClipRect(elem)
         {
            var clipString = elem.style.clip;
            // assumes "rect(npx, npx, npx, npx)" form
            // get rid of "rect("
            clipString = clipString.replace(/rect\(/,"");
            // get rid of "px)"
            clipString = clipString.replace(/px\)/,"");
            // get rid of remaining "px" strings
            clipString = clipString.replace(/px/g,",");
            // get rid of any double commas that are left after the above
            // getting rids
            clipString = clipString.replace(/,,/g,",");
            // turn remaining string into an array
            clipArray = clipString.split(",");
            // make object out of array values
            var clipRect = {top:parseInt(clipArray[0]),
               right:parseInt(clipArray[1]),
               bottom:parseInt(clipArray[2]), left:parseInt(clipArray[3])};
            return clipRect;
         }
      </script>
   </head>
   <body onload="showValues()">
```

```
      <h1>Layer vs. Clip Dimension Properties</h1>
      <hr />
      Enter new layer and clipping values to adjust the layer.
      <div id="form4clipValues">
         <form>
            <table>
               <tr>
                  <td align="right">layer.style.width:</td>
                  <td>
                     <input type="text" name="width" size="3"
                            onchange="setLayer(this)" />
                  </td>
               </tr>
               <tr>
                  <td align="right">layer.style.height:</td>
                  <td>
                     <input type="text" name="height" size="3"
                            onchange="setLayer(this)" />
                  </td>
               </tr>
               <tr>
                  <td align="right">layer.style.clip (right):</td>
                  <td>
                     <input type="text" name="clipRight" size="3"
                            onchange="setClip(this)" />
                  </td>
               </tr>
               <tr>
                  <td align="right">layer.style.clip (bottom):</td>
                  <td>
                     <input type="text" name="clipBottom" size="3"
                            onchange="setClip(this)" />
                  </td>
               </tr>
            </table>
         </form>
      </div>
      <div id="display"
           style="top:130px; left:250px; width:360px; height:180px;
               clip:rect(0px 360px 180px 0px);">
         <h2>ARTICLE I</h2>
         <p>Congress shall make no law respecting an establishment of religion,
            or prohibiting the free exercise thereof; or abridging the freedom
            of speech, or of the press; or the right of the people peaceably to
            assemble, and to petition the government for a redress of
            grievances.
         </p>
      </div>
   </body>
</html>
```

Scripting nested layers

Working with nested layer locations, especially in a cross-browser manner, presents numerous browser-specific syntax problems that need equalization to behave the same to all users. Some discrepancies even appear between Windows and Macintosh versions of IE. (Granted, MacIE is in limited use these days, but it's still worth mentioning the inconsistencies between IE browsers on different platforms.)

The scenario for Listing 43-5 consists of one positioned layer (greenish) nested inside another (reddish). The inner layer is initially sized and positioned so that the outer layer extends 5 pixels in each direction. Text boxes enable you to adjust the coordinates for either layer relative to the entire page, as well as the layer's positioning context. If you make a change to any one value, all the others are recalculated and displayed to show you the effect the change has on other coordinate values.

As you see when you load the page, the outer element's positioning context is the page, so the *page* and *container* coordinates are the same (although the calculations to achieve this equality are not so simple across all browsers). The inner layer's initial page coordinates are to the right and down 5 pixels in each direction, and the coordinates within the container show those 5 pixels.

Because of browser idiosyncrasies, calculating the coordinates within the page takes the most work. The `getGrossOffsetLeft()` and `getGrossOffsetTop()` functions perform those calculations in the page. Passed a reference to the positioned element to be measured, the first number to grab is whatever the browser returns as the `offsetLeft` or `offsetTop` value of the element (see Chapter 26, "Generic HTML Element Objects"). These values are independent of the `style` property, and they can report different values for different browsers. IE, for example, measures the offset with respect to whatever it determines as the next outermost positioning context. Mozilla browsers, on the other hand, treat the page as the positioning context regardless of nesting. So, as long as there is an `offsetParent` element, a `while` loop starts accumulating the `offsetLeft` measures of each succeeding offset parent element going outward from the element. But even before that happens, a correction for MacIE must be accounted for. If there is a difference between the `style.left` and `offsetLeft` property values of an element, that difference is added to the offset. In MacIE5+, for example, failure to correct this results in the page and container values of the outer layer being 10 pixels different in each direction. Values returned from these two gross measures are inserted in the readouts for the page measures of both inner and outer elements.

Reading the coordinates relative to each element's container is easy: The `style.left` and `style.top` properties have the correct values for all browsers. Moving a layer with respect to its positioning context (the container values) is equally easy: assign the entered values to the same `style.left` and `style.top` properties.

Moving the layers with respect to the page coordinate planes (via the `setOuterPage()` and `setInnerPage()` functions) involves going the long way to assign values that take each browser's positioning idiosyncrasies into account. The way you move a positioned element (cross-browser, anyway) is to assign a value to the `style.left` and `style.top` properties. These values are relative to their positioning context, but Mozilla doesn't offer any shortcuts to reveal what element defines the positioning context for a nested element. Calls

to the getNetOffsetLeft() and getNetOffsetTop() functions do the inverse of the getGrossOffsetLeft() and getGrossOffsetTop() functions. Because the values received from the text box are relative to the entire page, the values must have any intervening positioning contexts subtracted from that value in order to achieve the net positioning values that can be applied to the style.left and style.top properties. To get there, however, a call to the getParentLayer() function cuts through the browser-specific implementations of container references to locate the positioning context so that its coordinate values can be subtracted properly. The same kind of correction for MacIE is required here as in the gross offset calculations; but here, the correction is subtracted from the value that eventually is returned as the value for either the style.left or style.top of the layer.

Let us add one quick word about the condition statements of the while constructions in the getNetOffsetLeft() and getNetOffsetTop() functions. You see here a construction not used frequently in this book, but one that is perfectly legal. When the conditional expression evaluates, the getParentLayer() method is invoked, and its returned value is assigned to the elem variable. That expression evaluates to the value returned by the function. As you can see from the getParentLayer() function definition, a value is returned as either an element reference or null. The while condition treats a value of null as false; any reference to an object is treated as true. Thus, the conditional expression does not use a comparison operator, but rather executes some code, and then branches based on the value returned by that code. Mozilla reports JavaScript warnings for this construction because it tries to alert you to a common scripting bug that occurs when you use the = operator when you really mean the == operator. But a warning is not the same as a script error, so don't be concerned when you see these messages in the JavaScript Console window during your debugging.

LISTING 43-5

Testing Nested Layer Coordinate Systems

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Nested Layer Coordinates</title>
      <style type="text/css">
         #outerDisplay
         {
            background-color:coral;
         }
         #innerDisplay
         {
            background-color:aquamarine;
         }
         #identifyOuter
         {
            color:coral;
            font-weight:bold;
         }
```

continued

LISTING 43-5 *(continued)*

```
      #identifyInner
      {
         color:aquamarine;
         font-weight:bold;
      }
      #settingTextBoxes
      {
         position:absolute; left:10px; top:130px;
      }
      .outerSettingLabel
      {
         text-align:right; background-color:coral;
      }
      .outerSettingTextBox
      {
         background-color:coral;
      }
      .innerSettingLabel
      {
         text-align:right; background-color:aquamarine;
      }
      .innerSettingTextBox
      {
         background-color:aquamarine;
      }
   </style>
   <script type="text/javascript">
      // offsets within page
      function getGrossOffsetLeft(elem)
      {
         var offset = 0;
         while (elem.offsetParent)
         {
            // correct for IE/Mac discrepancy between offset and style
            // coordinates, but not if the parent is HTML element (NN6)
            offset += (elem.offsetParent.tagName != "HTML")
                     ? parseInt(elem.style.left) - parseInt(elem.offsetLeft) : 0;
            elem = elem.offsetParent;
            offset += elem.offsetLeft;
         }
         return offset;
      }
      function getGrossOffsetTop(elem)
      {
         var offset = 0;
         while (elem.offsetParent)
         {
            // correct for IE/Mac discrepancy between offset and style
            // coordinates, but not if the parent is HTML element (NN6)
            offset += (elem.offsetParent.tagName != "HTML")
                      ? parseInt(elem.style.top) - parseInt(elem.offsetTop) : 0;
```

```
         elem = elem.offsetParent;
         offset += elem.offsetTop;
      }
      return offset;
   }

   // offsets within element's positioning context
   function getNetOffsetLeft(offset, elem)
   {
      while (elem = getParentLayer(elem))
      {
         // correct for IE/Mac discrepancy between offset and style
         // coordinates, but not if the parent is HTML element (NN6)
         offset -= (elem.offsetParent.tagName != "HTML")
                 ? parseInt(elem.style.left) - parseInt(elem.offsetLeft) : 0;
         offset -= elem.offsetLeft;
      }
      return offset;
   }
   function getNetOffsetTop(offset, elem)
   {
      while (elem = getParentLayer(elem))
      {
         // correct for IE/Mac discrepancy between offset and style
         // coordinates, but not if the parent is HTML element (NN6)
         offset -= (elem.offsetParent.tagName != "HTML")
                   ? parseInt(elem.style.top) - parseInt(elem.offsetTop) : 0;
         offset -= elem.offsetTop;
      }
      return offset;
   }
   // find positioning context parent element
   function getParentLayer(elem)
   {
      if (elem.parentNode)
      {
         while (elem.parentNode != document.body)
         {
            elem = elem.parentNode;
            while (elem.nodeType != 1)
            {
               elem = elem.parentNode;
            }
            if (elem.style.position == "absolute" ||
                elem.style.position == "relative")
            {
               return elem;
            }
            elem = elem.parentNode;
         }
         return null;
      }
      else if (elem.offsetParent && elem.offsetParent.tagName != "HTML")
```

continued

LISTING 43-5 *(continued)*

```
      {
         return elem.offsetParent;
      }
      else
      {
         return null;
      }
   }

   // functions that respond to changes in text boxes
   function setOuterPage(field)
   {
      var val = parseInt(field.value);
      var elem = document.getElementById("outerDisplay");
      switch (field.name)
      {
         case "pageX" :
            elem.style.left = ((elem.offsetParent)
                                 ? getNetOffsetLeft(val, elem) : val)
                               + "px";
            break;
         case "pageY" :
            elem.style.top = ((elem.offsetParent)
                                 ? getNetOffsetTop(val, elem) : val)
                               + "px";
            break;
      }
      showValues();
   }
   function setOuterLayer(field)
   {
      var val = parseInt(field.value);
      switch (field.name)
      {
         case "left" :
            document.getElementById("outerDisplay").style.left = val + "px";
            break;
         case "top" :
            document.getElementById("outerDisplay").style.top = val + "px";
            break;
      }
      showValues();
   }
   function setInnerPage(field)
   {
      var val = parseInt(field.value);
      var elem = document.getElementById("innerDisplay");
      switch (field.name)
      {
         case "pageX" :
```

```
                elem.style.left = ((elem.offsetParent)
                                  ? getNetOffsetLeft(val, elem) : val)
                                + "px";
                break;
             case "pageY" :
                elem.style.top = ((elem.offsetParent)
                                 ? getNetOffsetTop(val, elem) : val)
                               + "px";
                break;
          }
          showValues();
       }
       function setInnerLayer(field)
       {
          var val = parseInt(field.value);
          switch (field.name)
             {
             case "left" :
                document.getElementById("innerDisplay").style.left = val + "px";
                break;
             case "top" :
                document.getElementById("innerDisplay").style.top = val + "px";
                break;
             }
          showValues();
       }
       function showValues()
       {
          var form = document.forms[0];
          var outer = document.getElementById("outerDisplay");
          var inner = document.getElementById("innerDisplay");
          form.elements[0].value = outer.offsetLeft +
                         ((outer.offsetParent) ? getGrossOffsetLeft(outer) : 0);
          form.elements[1].value = outer.offsetTop +
                         ((outer.offsetParent) ? getGrossOffsetTop(outer) : 0);
          form.elements[2].value = parseInt(outer.style.left);
          form.elements[3].value = parseInt(outer.style.top);
          form.elements[4].value = inner.offsetLeft +
                         ((inner.offsetParent) ? getGrossOffsetLeft(inner) : 0);
          form.elements[5].value = inner.offsetTop +
                         ((inner.offsetParent) ? getGrossOffsetTop(inner) : 0);
          form.elements[6].value = parseInt(inner.style.left);
          form.elements[7].value = parseInt(inner.style.top);
       }
    </script>
 </head>
 <body onload="showValues()">
    <h1>Nested Layer Coordinates</h1>
    <hr />
    Enter new page and layer coordinates for the <span id="identifyOuter">outer
    layer</span> and <span id="identifyInner">inner layer</span> objects.
    <div id="settingTextBoxes">
       <form>
```

continued

LISTING 43-5 *(continued)*

```
<table>
   <tr>
      <td class="outerSettingLabel">Page X:</td>
      <td class="outerSettingTextBox">
         <input type="text" name="pageX" size="3"
                onchange="setOuterPage(this)" />
      </td>
   </tr>
   <tr>
      <td class="outerSettingLabel">Page Y:</td>
      <td class="outerSettingTextBox">
         <input type="text" name="pageY" size="3"
                onchange="setOuterPage(this)" />
      </td>
   </tr>
   <tr>
      <td class="outerSettingLabel">Container X:</td>
      <td class="outerSettingTextBox">
         <input type="text" name="left" size="3"
                onchange="setOuterLayer(this)" />
      </td>
   </tr>
   <tr>
      <td class="outerSettingLabel">Container Y:</td>
      <td class="outerSettingTextBox">
         <input type="text" name="top" size="3"
                onchange="setOuterLayer(this)" />
      </td>
   </tr>
   <tr>
      <td class="innerSettingLabel">Page X:</td>
      <td class="innerSettingTextBox">
         <input type="text" name="pageX" size="3"
                onchange="setInnerPage(this)" />
      </td>
   </tr>
   <tr>
      <td class="innerSettingLabel">Page Y:</td>
      <td class="innerSettingTextBox">
         <input type="text" name="pageY" size="3"
                onchange="setInnerPage(this)" />
      </td>
   </tr>
   <tr>
      <td class="innerSettingLabel">Container X:</td>
      <td class="innerSettingTextBox">
         <input type="text" name="left" size="3"
                onchange="setInnerLayer(this)" />
      </td>
```

```
            </tr>
            <tr>
               <td class="innerSettingLabel">Container Y:</td>
               <td class="innerSettingTextBox">
                  <input type="text" name="top" size="3"
                         onchange="setInnerLayer(this)" />
               </td>
            </tr>
         </table>
      </form>
   </div>
   <div id="outerDisplay"
   style="position:absolute; top:130px; left:200px; width:370px; height:190px;">
      <div id="innerDisplay"
      style="position:absolute; top:5px; left:5px; width:360px; height:180px;">
         <h2>ARTICLE I</h2>
         <p>Congress shall make no law respecting an establishment of
            religion, or prohibiting the free exercise thereof; or abridging
            the freedom of speech, or of the press; or the right of the
            people peaceably to assemble, and to petition the government for
            a redress of grievances.
         </p>
      </div>
   </div>
  </body>
</html>
```

Try entering a variety of values in all text boxes to see what happens. Here is one possible sequence of tests and explanations:

1. Increase the red Page X value to 250. This moves the outer layer to the right by 50 pixels. Because the green layer is nested inside, it moves along with it. The green Page X value also increases by 50, but its Container X value remains the same because the inner layer maintains the same relationship with the outer layer as before.
2. Increase the green Page X value to 300. This action shifts the position of the green inner layer by 45 pixels, making it a total of 50 pixels inset within its positioning context. Because the outer layer does not have its clipping rectangle set, the inner layer's content bleeds beyond the width of the red layer.
3. Set the green Container Y value to −50. This action moves the green inner layer upward so that its top is 50 pixels above the top of its red container. As a result, the Page Y value of the green inner layer is 80, while the Page Y value of the red outer layer remains at 130 (thus, the 50-pixel difference).

As you experiment with moving the layers around, you may encounter some screen refresh problems where traces of the inner layer remain when moved beyond the outer layer's rectangle. Take these bugs into account when you design the actions of your script-controlled positioning.

Loading external HTML into a layer

A handy approach to loading external HTML content into a layer involves the `iframe` element. This element makes it possible to assign a style sheet rule that absolute-positions the `iframe` precisely on the page where you want it. Be sure to set the `frameborder` attribute to `0` unless you want the border to be visible to the user (and then watch out for content that may overrun the rectangle and cause scrollbars to appear). In this case, you must then leave all the formatting and style sheet control of that content to the HTML loaded into the `iframe`, just as if it were in a separate window or frame. To load different content into the element, assign a different URL to the `src` property of the `iframe` element object.

As an example that more closely simulates the loading of external content into a layer, Listing 43-6 demonstrates a somewhat ugly workaround that lets a layer's background color or image show through some kinds of HTML content.

A hidden `iframe` element is the initial recipient of the external HTML file, as loaded by the `loadOuter()` method. When that file loads, the `transferHTML()` method is invoked to copy the `innerHTML` of just the `body` element of the content window of the `iframe` (note the different syntax for Mozilla — the `contentDocument` property — and IE5.5+ — the `contentWindow` property). By eliminating the `body` element and any tags in the `head`, you prevent the tags in the layer from conflicting with the tags for the main document. As a result, however, notice how the background color you've set for the layer shows through the HTML plugged into the layer.

HTML element objects (other than `iframe`) were not designed to get their content from external files. But, as Listing 43-6 shows, where there is a will there is a way — even if the workaround isn't pretty.

LISTING 43-6

Setting Layer Source Content

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Loading External Content into a Layer</title>
      <style type="text/css">
         #identifyLayer
         {
            color:coral;
            font-weight:bold;
         }
         #settingButtonsBackground
         {
            background-color:coral;
            position:absolute; top:150px; width:200px;
         }
```

```
      #outerDisplay
      {
         background-color:coral;
         position:absolute; top:150px; left:250px; width:370px; height:190px;
      }
      #someText
      {
         font-weight:bold;
      }
   </style>
   <script type="text/javascript">
      function loadOuter(doc)
      {
         document.getElementById("hiddenContent").src = doc;
         // workaround for missing onload event in iframe for Moz
            if (!document.getElementById("hiddenContent").onload)
            {
               setTimeout("transferHTML()", 1000);
            }
      }
      function transferHTML()
      {
         var srcFrame = document.getElementById("hiddenContent");
         var srcContent = (srcFrame.contentDocument) ?
            srcFrame.contentDocument.getElementsByTagName("BODY")[0].innerHTML :
            (srcFrame.contentWindow) ?
            srcFrame.contentWindow.document.body.innerHTML : "";
         document.getElementById("outerDisplay").innerHTML = srcContent;
      }
   </script>
</head>
<body>
   <h1>Loading External Content into a Layer</h1>
   <hr />
   <p>Click the buttons to see what happens when you load new source
      documents into the <span id="identifyLayer">layer</span> object.
   </p>
   <div id="settingButtonsBackground">
      <form>
         Load into outer layer:<br />
         <input type="button" value="Article I"
                onclick="loadOuter('article1.html')" />
         <br />
         <input type="button" value="Entire Bill of Rights"
                onclick="loadOuter('bofright.html')" />
         <br />
      </form>
   </div>
   <div id="outerDisplay">
      <p id="someText">Placeholder text for layer.</p>
   </div>
```

continued

LISTING 43-6 *(continued)*

```
      <iframe id="hiddenContent" style="visibility:hidden;"
              onload="transferHTML()">
      </iframe>
   </body>
</html>
```

Note

As handy as it may be, in a strict W3C approach to JavaScript, you wouldn't use the `innerHTML` property since it isn't officially part of the W3C standard. However, it is often too powerful a convenience property to ignore, as much of the code throughout this book is a testament. The book does show the W3C node manipulation alternative to `innerHTML` in some examples. Refer to Chapter 29, "The Document and Body Objects," for a thorough explanation and examples of the W3C alternative to `innerHTML`. ■

Positioned element visibility behavior

There is very little code in Listing 43-7 because it simply adjusts the `style.visibility` property of an outer layer and a nested, inner layer. You can see that when the page loads, the green inner layer's `visibility` is automatically set to inherit the `visibility` of its containing outer layer. When you click the outer layer buttons, the inner layer blindly follows the settings.

Things change, however, when you start adjusting the properties of the inner layer independently of the outer layer. With the outer layer hidden, you can show the inner layer. Only by setting the `visibility` property of the inner layer to `inherit` can you make it rejoin the outer layer in its behavior.

LISTING 43-7

Nested Layer Visibility Relationships

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>layer.style.visibility</title>
      <style type="text/css">
         #identifyOuter
         {
            color:coral;
            font-weight:bold;
         }
         #identifyInner
         {
            color:aquamarine;
            font-weight:bold;
         }
```

```
      #outerButtonsBackground
      {
         background-color:coral;
         position:absolute; top:150px; width:180px;
      }
      #innerButtonsBackground
      {
         background-color:aquamarine;
         position:absolute; top:270px; width:180px;
      }
      #someText
      {
         font-weight:bold;
      }
      #outerDisplay
      {
         position:absolute; top:150px; left:200px;
         width:370px; height:190px;
         background-color:coral;
      }
      #innerDisplay
      {
         position:absolute; top:5px; left:5px;
         width:360px; height:180px;
         background-color:aquamarine;
      }
   </style>
   <script type="text/javascript">
      function setOuterVis(type)
      {
         document.getElementById("outerDisplay").style.visibility = type;
      }
      function setInnerVis(type)
      {
         document.getElementById("innerDisplay").style.visibility = type;
      }
   </script>
</head>
<body>
   <h1>Setting the <tt>layer.style.visibility</tt> Property of Nested Layers</h1>
   <hr />
   Click the buttons to see what happens when you change the visibility of
   the <span id="identifyOuter">outer layer</span> and
   <span id="identifyInner">inner layer</span> objects.
   <div id="outerButtonsBackground">
      <form>
         Control outer layer visibility property:
         <br />
         <input type="button" value="Hide Outer Layer"
                onclick="setOuterVis('hidden')" />
         <br />
         <input type="button" value="Show Outer Layer"
                onclick="setOuterVis('visible')" />
```

continued

LISTING 43-7 *(continued)*

```
            <br />
         </form>
      </div>
      <div id="innerButtonsBackground">
         <form>
            Control inner layer visibility property:
            <br />
            <input type="button" value="Hide Inner Layer"
                   onclick="setInnerVis('hidden')" />
            <br />
            <input type="button" value="Show Inner Layer"
                   onclick="setInnerVis('visible')" />
            <br />
            <input type="button" value="Inherit Outer Layer"
                   onclick="setInnerVis('inherit')" />
            <br />
         </form>
      </div>
      <div id="outerDisplay">
         <div id="innerDisplay">
            <p id="someText">Placeholder text for raw inner layer.</p>
         </div>
      </div>
   </body>
</html>
```

Scripting layer stacking order

Unlike layers in earlier Netscape browsers, the modern W3C DOM does not have properties that reveal the equivalent of the `layerObject.above` or `layerObject.below` properties. Therefore, Listing 43-8 confines itself to enabling you to adjust the `style.zIndex` property values of three overlapping layers. All three layers (none of which are nested inside another) initially set their `zIndex` values to `0`, meaning that the source code order rules the stacking order.

LISTING 43-8

Relationships Among zIndex Values

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>layer.style.zIndex</title>
      <style type="text/css">
         #settingOriginalTopLayer
         {
```

```
         position:absolute; top:140px; left:10px;
         width:240px;
         background-color:coral;
      }
      #settingOriginalMidLayer
      {
         position:absolute; top:220px; left:10px;
         width:240px;
         background-color:aquamarine;
      }
      #settingOriginalBotLayer
      {
         position:absolute; top:300px; left:10px;
         width:240px;
         background-color:yellow;
      }
      #bottomLayer
      {
         position:absolute; top:140px; left:260px;
         width:300px; height:190px;
         background-color:coral;
      }
      #middleLayer
      {
         position:absolute; top:160px; left:280px;
         width:300px; height:190px;
         background-color:aquamarine;
      }
      #topLayer
      {
         position:absolute; top:180px; left:300px;
         width:300px; height:190px;
         background-color:yellow;
      }
      .someText
      {
         font-weight:bold;
      }
   </style>
   <script type="text/javascript">
      function setZ(field)
      {
         switch (field.name)
         {
            case "top" :
               document.getElementById("topLayer").style.zIndex
                  = parseInt(field.value);
               break;
            case "mid" :
               document.getElementById("middleLayer").style.zIndex
                  = parseInt(field.value);
               break;
            case "bot" :
```

continued

LISTING 43-8 *(continued)*

```
                document.getElementById("bottomLayer").style.zIndex
                   = parseInt(field.value);
            }
            showValues();
         }
         function showValues()
         {
            var botLayer = document.getElementById("bottomLayer");
            var midLayer = document.getElementById("middleLayer");
            var topLayer = document.getElementById("topLayer");

            document.forms[0].bot.value = botLayer.style.zIndex;
            document.forms[1].mid.value = midLayer.style.zIndex;
            document.forms[2].top.value = topLayer.style.zIndex;
         }
      </script>
   </head>
   <body onload="showValues()">
      <h1><tt>layer.style.zIndex</tt> Property of Sibling Layers</h1>
      <hr />
      Enter new zIndex values to see the effect on three layers.
      <div id="settingOriginalTopLayer">
         <form>
            Control Original Bottom Layer:
            <br />
            <table>
               <tr>
                  <td align="right">Layer zIndex:</td>
                  <td>
                     <input type="text" name="bot" size="3"
                            onchange="setZ(this)" />
                  </td>
               </tr>
            </table>
         </form>
      </div>
      <div id="settingOriginalMidLayer">
         <form>
            Control Original Middle Layer:
            <br />
            <table>
               <tr>
                  <td align="right">Layer zIndex:</td>
                  <td>
                     <input type="text" name="mid" size="3"
                            onchange="setZ(this)" />
                  </td>
               </tr>
            </table>
```

```
      </form>
   </div>
   <div id="settingOriginalBotLayer">
      <form>
         Control Original Top Layer:
         <br />
         <table>
            <tr>
               <td align="right">Layer zIndex:</td>
               <td>
                  <input type="text" name="top" size="3"
                         onchange="setZ(this)" />
               </td>
            </tr>
         </table>
      </form>
   </div>
   <div id="bottomLayer" style="z-Index:0;">
      <span class="someText">Original Bottom Layer</span>
   </div>
   <div id="middleLayer" style="z-Index:0;">
      <span class="someText">Original Middle Layer</span>
   </div>
   <div id="topLayer" style="z-Index:0;">
      <span class="someText">Original Top Layer</span>
   </div>
  </body>
</html>
```

Dragging and resizing a layer

Listing 43-9 shows a script that enables you to click and drag a layer around the screen. The script employs the coordinate values of the `mousemove` event; after compensating for the offset within the layer at which the click occurs, the script moves the layer to track the mouse action.

Note in the following listing how several mouse events are handled. Event bubbling is used so that all mouse events are handled at the document level. Thus, all of the event handlers need to equalize the `event` object and the `event` target element, as well as filter events, so that the action occurs only when a draggable element (as identified by its `className` property) is the target of the event action. Note also how the `mousedown` event is bound for the entire life of the application, whereas the `mouseup` and `mousemove` events are bound and unbound with each new drag operation.

Within the `engage()` function, the `mousedown` event sets a global variable (`draggedElem`) that contains the element being dragged. At the same time, the script records how far away from the layer's top-left corner the `mousedown` event occurred. This offset information is needed so that any setting of the layer's location takes this offset into account (otherwise, the top-left corner of the layer would jump to the cursor position and be dragged from there).

The engage() function is a bit tricky because it must use the two different event and element object models to establish the offset of the mousedown event within the draggable element. For WinIE, this also means taking the scrolling of the body element into account. To get the element to reposition itself with mouse motion, the dragIt() function applies browser-specific coordinate values to the style.left and style.top properties of the draggable element. This function is invoked very frequently in response to the mousemove event.

During the drag (which are mousedown events firing with each mouse movement), the dragIt() function checks whether the drag mode is engaged. If so, the layer is moved to the page location calculated by subtracting the original downstroke offset from the mousemove event location on the page. When the user releases the mouse button, the mouseup event turns off the drag mode by clearing the global dragged element object.

It's worth noting that nothing in this example treats the zIndex stacking order, which must be addressed if the page contains multiple draggable items.

Cross-Reference

See the map puzzle game in Chapter 59, "Application: Cross-Browser DHTML Map Puzzle" (on the CD-ROM) for an example of processing multiple draggable items. ■

LISTING 43-9

Dragging a Layer

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Layer Dragging</title>
      <style type="text/css">
         #myLayer
         {
            position:absolute;
         }
         .draggable
         {
            cursor:hand;
            font-weight:bold;
            background-color:lightgreen;
         }
      </style>
      <script type="text/javascript">
         var draggedElem;
         var offsetX = 0;
         var offsetY = 0;
         function dragIt(evt)
         {
```

```
    evt = (evt) ? evt : (window.event) ? window.event : "";
    var targElem = (evt.target) ? evt.target : evt.srcElement;
       if (draggedElem)
       {
          targElem = draggedElem;
             if (targElem.className == "draggable")
             {
                while (targElem.id != "myLayer" && targElem.parentNode)
                {
                   targElem = targElem.parentNode;
                }
                   if (evt.pageX)
                   {
                      targElem.style.left = evt.pageX - offsetX + "px";
                      targElem.style.top = evt.pageY - offsetY + "px";
                   }
                   else
                   {
                      targElem.style.left = evt.clientX - offsetX + "px";
                      targElem.style.top = evt.clientY - offsetY + "px";
                   }
                return false;
             }
       }
 }
 function engage(evt)
 {
    evt = (evt) ? evt : (window.event) ? window.event : "";
    var targElem = (evt.target) ? evt.target : evt.srcElement;
       if (targElem.className == "draggable")
       {
          while (targElem.id != "myLayer" && targElem.parentNode)
          {
             targElem = targElem.parentNode;
          }
             if (targElem.id == "myLayer")
             {
                addEvent(document, "mouseup", release);
                addEvent(document, "mousemove", dragIt);
                draggedElem = targElem;
                   if (evt.pageX)
                   {
                      offsetX = evt.pageX - targElem.offsetLeft;
                      offsetY = evt.pageY - targElem.offsetTop;
                   }
                   else
                   {
                      offsetX = evt.offsetX - document.body.scrollLeft;
                      offsetY = evt.offsetY - document.body.scrollTop;
                         if (navigator.userAgent.indexOf("Win") == -1)
                         {
                            offsetX += document.body.scrollLeft;
```

continued

LISTING 43-9 *(continued)*

```
                        offsetY += document.body.scrollTop;
                     }
                 }
              return false;
           }
       }
   }
   function release(evt)
   {
      evt = (evt) ? evt : (window.event) ? window.event : "";
      var targElem = (evt.target) ? evt.target : evt.srcElement;
         if (targElem.className == "draggable")
         {
            while (targElem.id != "myLayer" && targElem.parentNode)
            {
               targElem = targElem.parentNode;
            }
               if (draggedElem && targElem.id == "myLayer")
               {
                  draggedElem = null;
                  removeEvent(document, "mouseup", release);
                  removeEvent(document, "mousemove", dragIt);
               }
         }
   }

   // bind/unbind the event handlers
   function addEvent(elem, evtType, func)
   {
      if (elem.addEventListener)
      {
         elem.addEventListener(evtType, func, false);
      }
      else if (elem.attachEvent)
      {
         elem.attachEvent("on" + evtType, func);
      }
      else
      {
         elem["on" + evtType] = func;
      }
   }
   function removeEvent(elem, evtType, func)
   {
      if (elem.removeEventListener)
      {
         elem.removeEventListener(evtType, func, false);
      }
      else if (elem.detachEvent)
      {
```

```
            elem.detachEvent("on" + evtType, func);
         }
         else
         {
            elem["on" + evtType] = null;
         }
      }
      addEvent(window, "load", function()
      {
         addEvent(document, "mousedown", engage);
      });
   </script>
</head>
<body>
   <h1>Dragging a Layer</h1>
   <hr />
   <div id="myLayer" class="draggable"
        style="top:90px; left:100px; width:300px; height:190px;">
      <span class="draggable">Drag me around the window.</span>
   </div>
</body>
</html>
```

Listing 43-10 applies many of the example components used thus far to let scripts control the resizing of a positionable element by dragging the lower-right, 20-pixel region. A lot of the hairy code in the `engage()` function is for determining if the `onmousedown` event occurs in the invisible 20-pixel square.

The `resizeIt()` function resembles the `dragIt()` function of Listing 43-9, but the adjustments are made to the width and height of the positionable element. A fair amount of math determines the width of the element in response to the cursor's instantaneous location and sets the `style.width` and `style.height` properties accordingly.

LISTING 43-10

Resizing a Layer

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Layer Resizing</title>
      <style type="text/css">
         #myLayer
         {
            background-color:lightblue;
            position:absolute;
         }
      </style>
```

continued

LISTING 43-10 *(continued)*

```
<script type="text/javascript">
   var draggedElem;
   var offsetX = 0;
   var offsetY = 0;

   function resizeIt(evt)
   {
      evt = (evt) ? evt : (window.event) ? window.event : "";
      var targElem = (evt.target) ? evt.target : evt.srcElement;
         if (draggedElem)
         {
            targElem = draggedElem;
               if (evt.pageX)
               {
                  targElem.style.width = (evt.pageX - targElem.offsetLeft
                     - offsetX)
                                    + "px";
                  targElem.style.height = (evt.pageY - targElem.offsetTop
                     - offsetY)
                                    + "px";
               }
               else
               {
                  var elemWidth = evt.clientX - targElem.offsetLeft
                              - offsetX - (parseInt(targElem.style.left)
                              - parseInt(targElem.offsetLeft));
                  var elemHeight = evt.clientY - targElem.offsetTop
                               - offsetY - (parseInt(targElem.style.top)
                               - parseInt(targElem.offsetTop));
                  targElem.style.width = elemWidth + "px";
                  targElem.style.height = elemHeight + "px";
               }
         }
   }

   function engage(evt)
   {
      evt = (evt) ? evt : (window.event) ? window.event : "";
      var targElem = (evt.target) ? evt.target : evt.srcElement;
         if (targElem.className == "draggable")
         {
            while (targElem.id != "myLayer" && targElem.parentNode)
            {
               targElem = targElem.parentNode;
            }
            if (targElem.id == "myLayer")
            {
               if (evt.pageX &&
                    (evt.pageX >
                    ((parseInt(targElem.style.width) - 20)
```

```
                        + targElem.offsetLeft)) &&
                     (evt.pageY > ((parseInt(targElem.style.height) - 20) +
                                   targElem.offsetTop)
                     )
                    )
                {
                   offsetX = evt.pageX - parseInt(targElem.style.width)
                           - targElem.offsetLeft;
                   offsetY = evt.pageY - parseInt(targElem.style.height)
                           - targElem.offsetTop;
                   addEvent(document, "mouseup", release);
                   addEvent(document, "mousemove", resizeIt);
                   draggedElem = targElem;
                }
                else if ((evt.offsetX > parseInt(targElem.style.width) - 20) &&
                         (evt.offsetY > parseInt(targElem.style.height) - 20))
                {
                   offsetX = evt.offsetX - parseInt(targElem.style.width)
                           - document.body.scrollLeft;
                   offsetY = evt.offsetY - parseInt(targElem.style.height)
                           - document.body.scrollTop;
                   addEvent(document, "mouseup", release);
                   addEvent(document, "mousemove", resizeIt);
                   draggedElem = targElem;
                      if (navigator.userAgent.indexOf("Win") == -1)
                      {
                         offsetX += document.body.scrollLeft;
                         offsetY += document.body.scrollTop;
                      }
                }
                return false;
             }
          }
}
function release(evt)
{
   evt = (evt) ? evt : (window.event) ? window.event : "";
   var targElem = (evt.target) ? evt.target : evt.srcElement;
      if (targElem.className == "draggable")
      {
         while (targElem.id != "myLayer" && targElem.parentNode)
         {
            targElem = targElem.parentNode;
         }
         if (draggedElem && targElem.id == "myLayer")

         {
            draggedElem = null;
            removeEvent(document, "mouseup", release);
            removeEvent(document, "mousemove", resizeIt);
         }
      }
}
```

continued

LISTING 43-10 *(continued)*

```
         // bind/unbind the event handlers
         function addEvent(elem, evtType, func)
         {
            if (elem.addEventListener)
            {
               elem.addEventListener(evtType, func, false);
            }
            else if (elem.attachEvent)
            {
               elem.attachEvent("on" + evtType, func);
            }
            else
            {
               elem["on" + evtType] = func;
            }
         }
         function removeEvent(elem, evtType, func)
         {
            if (elem.removeEventListener)
            {
               elem.removeEventListener(evtType, func, false);
            }
            else if (elem.detachEvent)
            {
               elem.detachEvent("on" + evtType, func);
            }
            else
            {
               elem["on" + evtType] = null;
            }
         }
         addEvent(window, "load", function()
            {
               addEvent(document, "mousedown", engage);
            }
         );
      </script>
   </head>
   <body>
      <h1>Resizing a Layer</h1>
      <hr />
      <div id="myLayer" class="draggable"
           style="top:170px; left:100px; width:300px; height:190px;">
         Here is some content inside the layer. See what happens to it as
         you resize the layer via the bottom-right 20-pixel handle.
      </div>
   </body>
</html>
```

This chapter only scratches the surface of the kinds of positioned element actions you can control with scripts. You may have seen examples of positioned element scripting at sites around the Web. For example, some pages have subject headers fly into place — even bounce around until they settle into position. Or elements can go in circles or spirals to get your attention (or distract you, as the case may be). The authors of those tricks apply formulas from other disciplines (such as games programming) to the `style` object properties of a positioned element.

Sometimes, the effects are there just for the sake of looking cool (at first, anyway) or because the page author knows how to script those effects. Your chief guide in implementing such features, however, should be whether the scripting genuinely adds value to the content offering. If you don't improve the content by adding flying doo-dads or pulsating images, then leave them out. A greater challenge is finding meaningful ways to apply positioning techniques. Done the right way and for the right reason, they can significantly enhance the visitor's enjoyment of your application.

CHAPTER

Embedded Objects

IN THIS CHAPTER

Using applet and embed element objects

Exploring the object element object

Understanding the unusual param element

In addition to the typical content that you see in web pages — primarily text and images — you can embed other kinds of content into the page. Such embedded content usually requires the powers of additional software, such as plug-in players or other external code processors, to load and display the content. Historically, all of this external content has been added to web pages by one of three HTML elements: `applet`, `embed`, or `object`. In the HTML 4.0 standard, the `applet` element, which was intended originally for loading Java applets, is deprecated in favor of the newer `object` element. That has not stopped modern browser makers from supporting the `applet` element (which they must for backward compatibility anyway), and even Java's maker recommends continued use of the `applet` element for loading applets over the Internet.

A similar standards trend exists for the `embed` element, which, in theory anyway, may ultimately be replaced exclusively by the `object` element. Mozilla browsers, however, prefer that embedded content requiring plug-ins (such as Flash content) be loaded through an `embed` element.

An `object` element is intended to be more extensible than `applet` and `embed`, meaning that it has enough attributes and power to summon the Java virtual machine if the incoming code is a Java applet, or to run an ActiveX program (in Internet Explorer for Windows, that is). In time, the `object` element may be the primary element for this kind of content, but for now it is the preferred approach for Internet Explorer and WebKit-based browsers.

It is possible to combine support for both the `object` and `embed` elements in one chunk of source code, and let the browser deploy the version itknows best. By nesting an `embed` element within an `object` element, you can cover both worlds easily, as in the following example:

```
<object width="425" height="350">
    <param name="movie" value="http://www.example.com/vid/2bq"></param>
    <embed src="http://www.example.com/vid/2bq"
          type="application/x-shockwave-flash" width="425" height="350">
    </embed>
</object>
```

In all cases, when a visual object is embedded through any of these elements, the control panel or applet occupies a segregated rectangular space on the page and generally confines its activities to that rectangle. But in some browsers and plug-in player types, JavaScript can also interact with the content or the player, allowing your scripts to extend themselves with powers for actions, such as controlling audio playback or the operation of a Java applet. Don't expect universal browser support for controlling plug-ins.

This chapter's primary focus is not on the content and players that you can control, but on the HTML element objects that load the content or players into the page in the first place. Most of the properties represent nothing more than scriptable access to the element HTML attributes. The property descriptions in this chapter are therefore not extensive. Online HTML references (including the W3C HTML 4.01 and the upcoming HTML5 specifications, and the Microsoft Developer Network documentation) should fill in the attribute value information quite well. In practice, scripts have very little interaction with these element objects, but if you ever need to know what's scriptable, you'll find that information here. As for controlling applets and plug-ins, you can find information about that in Chapter 47, "Scripting Java Applets and Plugs-Ins" (on the CD-ROM).

applet Element Object

For HTML element properties, methods, and event handlers, see Chapter 26, "Generic HTML Element Objects."

Properties	Methods	Event Handlers
`align`	(Applet methods)	`oncellchange`
`alt`		`ondataavailable`
`altHTML`		`ondatasetchanged`
`archive`		`ondatasetcomplete`
`code`		`onload`
`codebase`		`onrowenter`
`height`		`onrowexit`
`hspace`		`onrowsdelete`
`name`		`onrowsinserted`

Properties	Methods	Event Handlers
object		onscroll
vspace		
width		
(Applet variables)		

Syntax

Accessing applet element object properties or methods:

```
(NN3+/IE4+)    [window.]document.appletName.property | method([parameters])
(NN3+/IE4+)    [window.]document.applets[index].property | method([parameters])
(IE4+)         [window.]document.all.appletID.property | method([parameters])
(IE5+/W3C)     [window.]document.getElementById("appletID").property |
               method([parameters])
```

Compatibility: WinIE4+, MacIE4+, NN3+, Moz+, Safari+, Opera+, Chrome+

About this object

The fact that the applet itself can expose public instance variables and public methods as properties and methods of the applet object means that the scriptable characteristics of an applet object are highly dependent upon the way the applet was written. You can learn more about how to compose an applet that exposes its innards to JavaScript in Chapter 47.

Perhaps the most important point to remember about accessing applets is that you must have them loaded and running before you can address them as objects. Although you cannot query an applet to find out whether it's loaded (as you can with an image), you can rely on the onload event handler of a window to fire only when all applets in the window are loaded and running. Internet Explorer also features an onload event handler for the applet element directly, but applets tend to be the last things to load on a page. Therefore, you won't be able to use an applet embedded in a document to help you create the HTML content of that page as it loads. But an applet can provide content for new documents or for modifiable elements of a page. With the highly dynamic object models of modern browsers, this can lead to all kinds of possibilities.

Java applets have also been used to maintain contact with a server after the page has loaded by way of a servlet running on the server. A servlet allows the applet to query or be refreshed with instantaneously updated information without having to reload the page. Of course, getting a sophisticated applet to run in a wide range of browsers and operating systems is a challenge unto itself.

A large set of event handlers for this element (all but onload and onscroll) is related to the application of WinIE data binding for param elements nested inside an applet element. These events fire when a variety of actions occur to the data source or recordset associated with the applet. For more about applying data binding, see http://msdn.microsoft.com/en-us/library/ms531385%28VS.85%29.aspx.

Properties

`align`

Value: String Read/Write

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

The `align` property controls either the horizontal or vertical alignment of the element with regard to surrounding content. String values of `left` or `right` cause the applet rectangle to cling to the left or right edges of its immediate positioning context. String values of `absbottom`, `absmiddle`, `baseline`, `bottom`, `middle`, `texttop`, or `top` influence the vertical alignment with respect to adjacent text, with the same kinds of results as corresponding values of the `style.verticalAlign` property.

Keep in mind that CSS alignment properties offer a more standards-compliant way to align applets. Then again, since the `applet` element is itself deprecated, using it at all runs against the grain of web standards (HTML 4+). Chapter 47 shows you how to resolve the deprecation problem and use the `object` element instead of the `applet` element to embed applets in pages.

Related Items: `style.verticalAlign` property

`alt`

Value: String Read/Write

Compatibility: WinIE6+, MacIE6+, NN6+, Moz+, Safari+, Opera+, Chrome+

The `alt` property represents the `alt` attribute, which should contain text that displays in the browser in the event that the applet does not load or the user has Java turned off in the browser preferences. This information should be set as the `applet` element's attribute, because assigning text to the property after the applet attempts to load does not insert the text into the page.

Related Items: `altHTML` property

`altHTML`

Value: String Read/Write

Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-, Opera-, Chrome-

The `altHTML` property is supposed to provide an `applet` element with HTML content to render if the applet doesn't load. In practice, assigning an HTML string to this property has no effect on an `applet` element.

Related Items: `alt` property

`archive`

Value: String Read/Write

Compatibility: WinIE6+, MacIE-, NN6+, Moz+, Safari+, Opera+, Chrome+

The `archive` property represents the `archive` attribute, which points to the URL of a compressed (`.zip`) file containing Java class files needed for the applet. The archive must include the class file that is assigned to the `code` attribute to get the applet loaded and started.

Related Items: `code` property

code

Value: String Read-Only

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

The `code` property is the URL string of the Java class file that is to begin loading the applet (or the property may be the entire applet if it consists of a single class file). You cannot change the code assigned to an applet after the element has loaded (even if the applet code did not load successfully).

Related Items: `codeBase` property

codeBase

Value: String Read-Only

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

The `codeBase` property is the string of the path on the server to the Java class file that is to begin loading the applet (or the property may be the entire applet if it consists of a single class file). The actual Java class filename is not part of the `codeBase` property.

Related Items: `code` property

height
width

Value: Integer Read/Write

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

The `height` and `width` properties represent the `height` and `width` attributes of the `applet` element. Although these values should be set through attributes in the tag, these properties can adjust the size of the applet after the fact in IE5+.

Related Items: `hspace`, `vspace` properties

hspace
vspace

Value: Integer Read/Write

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

The `hspace` and `vspace` properties represent the `hspace` and `vspace` attributes of the `applet` element, which control the number of pixels of transparent padding around the `applet` element

on the page. Although these values should be set through attributes in the tag, these properties can adjust the size of the applet padding after the fact in IE5+.

Related Items: `height`, `width` properties

name

Value: String Read-Only

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

The `name` property represents the `name` attribute, a holdover from the early implementations of the `applet` element before `id` attributes were used to identify elements. The value assigned to the `name` attribute is the name you can use to reference applets in all browsers that support accessing applets: `document.appletName`.

object

Value: String Read-Only

Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-, Opera-, Chrome-

The `object` property represents the `object` attribute, which, according to the W3C HTML standard, points to the URL of a serialized (that is, saved) version of the applet's current state.

Related Items: `code` property

vspace

(See `hspace`)

width

(See `height`)

object Element Object

For HTML element properties, methods, and event handlers, see Chapter 26, "Generic HTML Element Objects."

Properties	Methods	Event Handlers
`align`	(Object methods)	`oncellchange`
`alt`		`ondataavailable`
`altHTML`		`ondatasetchanged`
`archive`		`ondatasetcomplete`
`baseHref`		`onload`
`baseURI`		`onrowenter`

Properties	Methods	Event Handlers
border		onrowexit
classid		onrowsdelete
code		onrowsinserted
codeBase		onscroll
codeType		
contentDocument		
data		
declare		
form		
height		
hspace		
name		
object		
standby		
type		
useMap		
vspace		
width		
(Object variables)		

Syntax

Accessing `object` element object properties or methods:

```
(IE4+)        [window.]document.all.objectID.property | method([parameters])
(IE5+/W3C)    [window.]document.getElementById("objectID").property |
              method([parameters])
```

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

About this object

The `object` element is intended to be the primary way to add external content (that is, content that the browser itself does not render) to a page. For example, Flash animations created with

Macromedia Flash are inserted into pages through the `object` element. The `object` element is also intended to replace usage of the `applet` and `embed` elements.

As with the `applet` element object, scripts can frequently control the programs and plug-ins that get loaded into the browser through the `object` tag. Chapter 47 shows you how to do that for common objects. The property listings here are merely for the properties of the element, most of which mimic the attributes available for the `object` element. Even though the properties are exposed, they are very rarely scripted, except perhaps to adjust the size of the space occupied by a media controller. Most properties are read-only after their values are set by attributes in the element's tag. But if your scripts are creating the `object` element anew, you can set the property values the first time to initialize the object.

Properties

align

Value: String Read/Write

Compatibility: WinIE4+, MacIE4+, NN-, Moz-, FF+, Safari+, Opera+, Chrome+

The `align` property controls either the horizontal or vertical alignment of the element with regard to surrounding content. String values of `left` or `right` cause the object rectangle to cling to the left or right edges of its immediate positioning context. String values of `absbottom`, `absmiddle`, `baseline`, `bottom`, `middle`, `texttop`, or `top` influence the vertical alignment with respect to adjacent text, with the same kinds of results as corresponding values of the `style.verticalAlign` property.

Related Items: `style.verticalAlign` property

alt

Value: String Read/Write

Compatibility: WinIE6+, MacIE-, NN-, Moz-, Safari-, Opera-, Chrome-

The `alt` property represents the `alt` attribute, which should contain text that displays in the browser in the event that the object or its data do not load. This information should be set as the `object` element's attribute, because assigning text to the property after the object attempts to load does not insert the text into the page.

Related Items: `altHTML` property

altHTML

Value: String Read/Write

Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-, Opera-, Chrome-

The `altHTML` property is supposed to provide an `object` element with HTML content to render if the object doesn't load. In practice, assigning an HTML string to this property has no effect on an `object` element.

Related Items: `alt` property

archive

Value: URI list as string Read/Write

Compatibility: WinIE6+, MacIE-, NN6+, Moz+, Safari+, Opera+, Chrome+

The `archive` property is used to specify one or more URIs for archives that contain resources associated with the object. If more than one URI is provided in the `archive` property, just separate them by a space. Although the property is implemented in the preceding browser versions (that is, you won't get script errors if you read or write its value), most browsers do not respond to changes of property values.

Related Items: `code`, `codeBase` properties

baseHref

Value: String Read-Only

Compatibility: WinIE4+, MacIE-, NN-, Moz-, Safari-, Opera-, Chrome-

The `baseHref` property returns the full URL path to the current document.

Related Items: `baseURI` property

baseURI

Value: String Read-Only

Compatibility: WinIE-, MacIE-, NN7+, Moz+, Safari+, Opera+, Chrome+

The `baseURI` property returns the full URL (URI) path to the current document.

Related Items: `baseHref` property

border

Value: Number as string Read/Write

Compatibility: WinIE6+, MacIE-, NN6+, Moz+, Safari+, Opera+, Chrome+

The `border` property controls the thickness of the border around the object, in pixels. You must specify the border thickness as a string, as in `"6"` for six pixels.

classid

Value: String Read-Only

Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-, Opera-, Chrome-

The `classid` property represents the `classid` attribute of the `object` element. Internet Explorer for Windows uses this attribute to assign the Globally Unique ID (GUID) of an ActiveX control. For example, to load a (nearly) invisible Windows Media Player object into a page, the HTML is as follows:

```
<object id="medPlayer" width="1" height="1"
        classid="CLSID:22d6f312-b0f6-11d0-94ab-0080c74c7e95"
        codebase="#Version=1,0,0,0">
```

If your script then accesses the `classid` property of the `medPlayer` object, the value returned is the complete string, as assigned to the attribute:

```
CLSID:22d6f312-b0f6-11d0-94ab-0080c74c7e95
```

Note that the `CLSID:` prefix is also part of the string value. Even if the object does not load (for example, because the object is missing or an error is in the long `classid` string), the property value reports the value as assigned to the attribute.

The HTML 4.0 specification indicates that the `classid` attribute be used for any kind of external class files, including Java applets. But in practice, Internet Explorer wants applet URLs supplied to the `code` attribute (a non-HTML attribute).

Related Items: `code` property

code

Value: String Read-Only

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

The `code` property is the URL string of a Java class file that is to begin loading the applet (or the property may be the entire applet if it consists of a single class file). You cannot change the code assigned to an applet after the element has loaded (even if the applet code did not load successfully).

Related Items: `codeBase` property

codeBase

Value: String Read-Only

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

The `codeBase` property is the string of the path on the server to the source of the applet or ActiveX control referenced by the `classid` or `code` attributes. IE4+ also uses the `codebase` attribute to specify a minimum version of control that is to load, if the attribute is available. This facet is discussed in the coverage of plug-in detection for WinIE in Chapter 42, “The Navigator and Other Environment Objects.”

Related Items: `code` property

codeType

Value: String Read-Only

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

The `codeType` property is a string of the `mime` type of whatever object is pointed to by the `code` attribute value.

Related Items: `type` property

contentDocument

Value: Document node reference Read-Only

Compatibility: WinIE-, MacIE-, NN6+, Moz+, Safari+, Opera+, Chrome+

The `contentDocument` property returns the document node created by the object, if it exists.

data

Value: URL as string Read-Only

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

The `data` property provides access to the URL of a file containing data for the object, as opposed to the object itself.

declare

Value: Boolean Read/Write

Compatibility: WinIE6+, MacIE-, NN6+, Moz+, Safari+, Opera+, Chrome+

The `declare` property is used to indicate that an object is a declaration and not a definition. In other words, a declared object doesn't actually appear on the page — it must be instantiated by an additional object definition. Although the property is implemented in the preceding browser versions (that is, you won't get script errors if you read or write its value), most browsers do not respond to changes of property values.

form

Value: Object reference Read-Only

Compatibility: WinIE4+, MacIE5, NN6+, Moz+, Safari+, Opera+, Chrome+

The `form` property returns a reference to the `form` element that contains the object, if there is one. This property only applies if the object is acting as a control within a form.

height
width

Value: Integer Read/Write

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

The `height` and `width` properties represent the `height` and `width` attributes of the `object` element. Although these values should be set through attributes in the tag, these properties can adjust the size of the embedded element after the fact in IE5+.

Related Items: `hspace`, `vspace` properties

hspace
vspace

Value: Integer — Read/Write

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

The `hspace` and `vspace` properties represent the `hspace` and `vspace` attributes of the `object` element; these attributes control the number of pixels of transparent padding around the `object` element on the page. Although these values should be set through attributes in the tag, these properties can adjust the size of the padding around the element after the fact in IE5+.

Related Items: `height`, `width` properties

name

Value: String — Read-Only

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

The `name` property represents the `name` attribute of the `object` element. It's better form to assign an `id` to the `object` element and use accepted reference syntax for element IDs.

object

Value: External object — Read-Only

Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-, Opera-, Chrome-

The `object` property returns a reference to the object contained by the `object` element. This property is essential if the program running inside the `object` element has the same property or method names as the `object` element itself. For example, consider a Java applet loaded into the `object` element, as follows:

```
<object code="coolApplet" id="myAPPLET" ... >
```

If the applet code contained a public variable called `height`, an attempt to read or write that property through the `object` element will cause the element's properties to be read, and not the applet's properties. Therefore, if you insert the `object` property in the reference, the script reaches into the `applet` object for the property:

```
document.getElementById("myAPPLET").object.height = 40;
```

If there is no ambiguity between element and object property and method names, the browser looks first at the element and then at the object to find a match.

standby

Value: String Read/Write

Compatibility: WinIE6+, MacIE5, NN6+, Moz+, Safari+, Opera+, Chrome+

The `standby` property enables you to set a text message that appears while an object is loading. By setting the `standby` property to a message, such as "Object loading . . . ," you provide the user with a visual cue for why the object hasn't yet sprung to life. Although the property is implemented in the preceding browser versions (that is, you won't get script errors if you read or write its value), most browsers do not respond to changes of property values.

type

Value: String Read-Only

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+, Opera+, Chrome+

The `type` property represents the `type` attribute of the `object` element, which is intended to warn the browser about the `mime` type of data that is to be loaded into the object's process.

Related Items: `codeType` property

useMap

Value: String Read/Write

Compatibility: WinIE6+, MacIE-, NN6+, Moz+, Safari+, Opera+, Chrome+

The `useMap` property identifies the URL of an image map; the image map is described by the `map` element in the same document. The value of the `useMap` property includes a hash mark followed by the name of the image map, as specified in the `usemap` attribute of the `object` element.

vspace

(See `hspace`)

width

(See `height`)

embed Element Object

For HTML element properties, methods, and event handlers, see Chapter 26, "Generic HTML Element Objects."

Properties	Methods	Event Handlers
align	(Object methods)	onload
height		onscroll
hidden		
name		
pluginspage		
src		
units		
width		
(Object variables)		

Syntax

Accessing embed element object properties or methods:

```
(IE4+)        [window.]document.all.objectID.property | method([parameters])
(IE5+/W3C)    [window.]document.getElementById("objectID").property |
              method([parameters])
```

Compatibility: WinIE4+, MacIE4+, NN3+, Moz+, Safari+, Opera+, Chrome+

About this object

The embed element is a carryover from the early browser days, although it will probably be adopted by the W3C HTML standard as part of HTML5. The embed element has been used as a way to embed non-native content (for example, sounds, video clips, and custom mime types for plug-ins, such as Shockwave) into a page. What gets embedded into the page is the controller or viewer for whatever kind of data the embed element points to (through the src attribute).

The embed element is far less sophisticated than the object element, but current browsers continue to support it. If you have been using the embed element in previous applications, it may be a good idea to start gravitating toward the object element. For backward-compatibility purposes, nesting an embed element inside an object element is not uncommon, since both attempt to load the same content and plug-in. Browsers that know about the object element will load the content that way; older browsers will use the embed element and its attributes and parameters.

Because an `embed` element loads a plug-in (including ActiveX control types of plug-ins in WinIE), you could reference the plug-in's properties and methods through the `embed` object's reference, even if the browser has disabled scripting. Be aware that most browsers will alert the visitor if there is an attempt to use these properties and methods, giving the visitor the opportunity to cancel the activity.

Properties

align

Value: String Read/Write

Compatibility: WinIE-, MacIE5+, NN6+, Moz+, Safari+-, Opera+, Chrome+

The `align` property controls either the horizontal or vertical alignment of the element with regard to surrounding content. String values of `left` or `right` cause the object rectangle to cling to the left or right edges of its next outermost positioning context. String values of `absbottom`, `absmiddle`, `baseline`, `bottom`, `middle`, `texttop`, or `top` influence the vertical alignment with respect to adjacent text, with the same kinds of results as corresponding values of the `style.verticalAlign` property.

Related Items: `style.verticalAlign` property

height
width

Value: Integer Read/Write

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari-, Opera+, Chrome+

The `height` and `width` properties represent the `height` and `width` attributes of the `embed` element. Although these values should be set through attributes in the tag, these properties can adjust the size of the element after the fact in IE5+.

hidden

Value: Boolean Read/Write

Compatibility: WinIE4+, MacIE-, NN-, Moz-, Safari-, Opera-, Chrome-

The `hidden` property represents the `hidden` attribute of the `embed` element. When an `embed` element is hidden, neither the controller nor the content is shown. Application of this element in modern browsers should use style sheets to hide and show the element.

Related Items: `style.visibility` property

`name`

Value: String | Read-Only

Compatibility: WinIE4+, MacIE4+, NN6+, Moz+, Safari+-, Opera+, Chrome+

The `name` property represents the `name` attribute of the `embed` element. It's better form to assign an `id` to the `embed` element and use accepted reference syntax for element IDs.

`pluginspage`

Value: String | Read-Only

Compatibility: WinIE4+, MacIE-, NN-, Moz-, Safari-, Opera-, Chrome-

The `pluginspage` property represents the `pluginspage` attribute of the `embed` element. This attribute is a URL that is applied to a link in the browser if the plug-in associated with the external file's `mime` type cannot be found on the client.

`src`

Value: String | Read/Write

Compatibility: WinIE4+, MacIE-, NN-, Moz-, FF+, Safari+, Opera+, Chrome+

The `src` property represents the `src` attribute of the `embed` element. This attribute points to the external file that is to be loaded into the browser through the associated plug-in. Scripts can assign a new URL string to this property to load a different file into the current plug-in.

`units`

Value: String | Read-Only

Compatibility: WinIE4+, MacIE-, NN-, Moz-, Safari-, Opera-, Chrome-

The `units` property returns the unit of measure assigned with the length value of the `height` and `width` properties, in pixels.

Related Items: `height`, `width` properties

The Odd Case of the param Element

HTML pages pass parameters to Java applets, plug-ins, and ActiveX controls by way of `param` elements that are nested inside `applet`, `embed`, and `object` elements. Although a `param` element object is defined by the W3C DOM Level 2 specification, it does not show up on some

browsers' radar when you try to reference the `param` element by itself. Even assigning an `id` to a `param` element or using `document.getElementsByTagName("param")` fail to allow references to access an individual `param` element object.

In practice, this limitation is not particularly important. For one thing, even if you could access the `param` elements of an embedded object or program, attempts to modify the values would be wasted: Those values are read at load time only. Secondly, a well-designed plug-in, applet, or ActiveX control will provide its own properties or methods to retrieve the current settings of whatever properties are initialized through the `param` elements.

CHAPTER

The Regular Expression and RegExp Objects

IN THIS CHAPTER

What regular expressions are

How to use regular expressions for text search-and-replace

How to apply regular expressions to string object methods

Web programmers who have worked in Perl (and other web application programming languages) know the power of regular expressions for processing incoming data, and formatting it for readability in an HTML page or for accurate storage in a server database. Any task that requires extensive search and replacement of text can greatly benefit from the flexibility and conciseness of regular expressions.

Most of the benefit of JavaScript regular expressions accrues to those who script their server-side programs on servers that support a JavaScript version that contains regular expressions. But that's not to exclude the client-side from application of this "language within a language." If your scripts perform client-side data validations or any other extensive text entry parsing, consider using regular expressions rather than cobbling together comparatively complex JavaScript functions to perform the same tasks.

Regular Expressions and Patterns

In several chapters earlier in this book, we describe expressions as any sequence of identifiers, keywords, and/or operators that evaluate to some value. A regular expression follows that description, but has much more power behind it. In essence, a regular expression uses a sequence of characters and symbols to define a pattern of text. Such a pattern is used to locate a chunk of text in a string by matching up the pattern against the characters in the string.

An experienced JavaScript writer may point out the availability of the `string.indexOf()` and `string.lastIndexOf()` methods that can instantly reveal whether a string contains a substring, and even where in the

string that substring begins. These methods work perfectly well when the match is exact, character for character. But if you want to do more sophisticated matching (for example, does the string contain a five-digit ZIP code?), you'd have to cast aside those handy string methods and write some parsing functions. That's the beauty of a regular expression: It lets you define a matching substring that has some intelligence about it and can follow guidelines you set as to what should or should not match.

The simplest kind of regular expression pattern is the same kind you use in the `string.indexOf()` method. Such a pattern is nothing more than the text that you want to match. In JavaScript, one way to create a regular expression is to surround the expression by forward slashes. For example, consider the string

```
Oh, hello, do you want to play Othello in the school play?
```

This string and others may be examined by a script whose job it is to turn formal terms into informal ones. Therefore, one of its tasks is to replace the word "hello" with "hi." A typical brute force search-and-replace function starts with a simple search string pattern. In JavaScript, you define a pattern (a regular expression) by surrounding it with forward slashes. For convenience and readability, we usually assign the regular expression to a variable, as in the following example:

```
var myRegExpression = /hello/;
```

In concert with some regular expression or string object methods, this pattern matches the string "hello" wherever that series of letters appears. The problem is that this simple pattern causes problems during the loop that searches and replaces the strings in the example string: It finds not only the standalone word "hello," but also the "hello" in "Othello."

Trying to write another brute force routine for this search-and-replace operation that looks only for standalone words would be a nightmare. You can't merely extend the simple pattern to include spaces on either or both sides of "hello," because there could be punctuation — a comma, a dash, a colon, or whatever — before or after the letters. Fortunately, regular expressions provide a shortcut way to specify general characteristics, including a feature known as a word boundary. The symbol for a word boundary is `\b` (backslash, lowercase b). If you redefine the pattern to include these specifications on both ends of the text to match, the regular expression creation statement looks like this:

```
var myRegExpression = /\bhello\b/;
```

When JavaScript uses this regular expression as a parameter in a special string object method that performs search-and-replace operations, it changes only the standalone word "hello" to "hi," and passes over "Othello" entirely.

If you are still learning JavaScript and don't have experience with regular expressions in other languages, you have a price to pay for this power: Learning the regular expression lingo, which is filled with so many symbols that expressions sometimes look like cartoon substitutions for swear words. The goal of this chapter is to introduce you to regular expression syntax as implemented in JavaScript rather than engage in lengthy tutorials on it. Of more importance in the long run, is understanding how JavaScript treats regular expressions as objects, and distinctions between

instances of regular expression objects and the `RegExp` static object. We hope the examples in the following sections begin to reveal the powers of regular expressions.

Language Basics

To cover the depth of the regular expression syntax, we divide the subject into three sections. The first covers simple expressions (some of which you've already seen). Then we get into the wide range of special characters used to define specifications for search strings. Last comes an introduction to using parentheses in the language, and how they not only help in grouping expressions for influencing calculation precedence (as they do for regular math expressions), but also how they temporarily store intermediate results of more complex expressions for use in reconstructing strings after their dissection by the regular expression.

Simple patterns

A simple regular expression uses no special characters for defining the string to be used in a search. Therefore, if you want to replace every space in a string with an underscore character, the simple pattern to match the space character is

```
var re = / /;
```

A space appears between the regular expression start and end forward slashes. The problem with this expression, however, is that it knows only how to find a single instance of a space in a long string. Regular expressions can be instructed to apply the matching string on a global basis by appending the `g` modifier:

```
var re = / /g;
```

When this `re` value is supplied as a parameter to the `replace()` method that uses regular expressions (described later in this chapter), the replacement is performed throughout the entire string, rather than just once on the first match found. Notice that the modifier appears after the final forward slash of the regular expression creation statement.

Regular expression matching — like a lot of other aspects of JavaScript — is case-sensitive. But you can override this behavior by using one other modifier (`i`) that lets you specify a case-insensitive match. Therefore, the following expression

```
var re = /web/i;
```

finds a match for "web," "Web," or any combination of uppercase and lowercase letters in the word. You can combine the two modifiers together at the end of a regular expression. For example, the following expression is both case-insensitive and global in scope:

```
var re = /web/gi;
```

In compliance with the ECMA-262 Edition 3 standard, modern browsers also allow a flag to force the regular expression to treat carriage-return-delimited lines of a multiline string as separate substrings with their own start and end boundaries. That modifier is the letter `m`.

Special characters

The regular expression in JavaScript borrows most of its vocabulary from the Perl regular expression. In a few instances, JavaScript offers alternatives to simplify the syntax, but also accepts the Perl version for those with experience in that arena.

Significant programming power comes from the way regular expressions allow you to include terse specifications about such facets as types of characters to accept in a match, how the characters are surrounded within a string, and how often a type of character can appear in the matching string. A series of escaped one-character commands (that is, letters preceded by the backslash) handle most of the character issues; punctuation and grouping symbols help define issues of frequency and range.

You saw an example earlier how `\b` specified a word boundary on one side of a search string. Table 45-1 lists the escaped character specifiers in JavaScript regular expressions. The vocabulary forms part of what are known as metacharacters — characters in expressions that are not matchable characters themselves, but act more as commands or guidelines of the regular expression language.

TABLE 45-1

JavaScript Regular Expression Matching Metacharacters

Character	Matches	Example
`\b`	Word boundary	`/\bor/` matches "origami" and "or" but not "normal" `/or\b/` matches "traitor" and "or" but not "perform" `/\bor\b/` matches full word "or" and nothing else
`\B`	Word non-boundary	`/\Bor/` matches "normal" but not "origami" `/or\B/` matches "normal" and "origami" but not "traitor" `/\Bor\B/` matches "normal" but not "origami" or "traitor"
`\d`	Numeral 0 through 9	`/\d\d\d/` matches "212" and "415" but not "B17"
`\D`	Non-numeral	`/\D\D\D/` matches "ABC" but not "212" or "B17"
`\s`	Single white space	`/over\sbite/` matches "over bite" but not "overbite" or "over bite"
`\S`	Single non-white space	`/over\Sbite/` matches "over-bite" but not "overbite" or "over bite"
`\w`	Letter, numeral, or underscore	`/A\w/` matches "A1" and "AA" but not "A+"
`\W`	Not letter, numeral, or underscore	`/A\W/` matches "A+" but not "A1" and "AA"
`.`	Any character except newline	`/.../` matches "ABC", "1+3", "A 3", or any three characters
`[...]`	Character set	`/[AN]BC/` matches "ABC" and "NBC" but not "BBC"
`[^...]`	Negated character set	`/[^AN]BC/` matches "BBC" and "CBC" but not "ABC" or "NBC"

Not to be confused with the metacharacters listed in Table 45-1, are the escaped string characters for tab (\t), newline (\n), carriage return (\r), formfeed (\f), and vertical tab (\v).

Let us further clarify the use of the [...] and [^...] metacharacters. You can specify either individual characters between the brackets (as shown in Table 45-1) or a contiguous range of characters, or both. For example, the \d metacharacter can also be defined by [0-9], meaning any numeral from zero through nine. If you only want to accept a value of 2 and a range from 6 through 8, the specification would be [26-8]. Similarly, the accommodating \w metacharacter is defined as [A-Za-z0-9_], reminding you of the case-sensitivity of regular expression matches not otherwise modified.

All but the bracketed character set items listed in Table 45-1 apply to a single character in the regular expression. In most cases, however, you cannot predict how incoming data will be formatted — the length of a word or the number of digits in a number. A batch of extra metacharacters lets you set the frequency of the occurrence of either a specific character or a type of character (specified like the ones in Table 45-1). If you have experience in command-line operating systems, you can see that some of the same ideas that apply to wildcards also apply to regular expressions. Table 45-2 lists the counting metacharacters in JavaScript regular expressions.

TABLE 45-2

JavaScript Regular Expression Counting Metacharacters

Character	Matches Last Character	Example
*	Zero or more times	/Ja*vaScript/ matches "JvaScript", "JavaScript", and "JaaavaScript" but not "JovaScript"
?	Zero or one time	/Ja?vaScript/ matches "JvaScript" or "JavaScript" but not "JaaavaScript"
+	One or more times	/Ja+vaScript/ matches "JavaScript" or "JaavaScript" but not "JvaScript"
{n}	Exactly n times	/Ja{2}vaScript/ matches "JaavaScript" but not "JvaScript" or "JavaScript"
{n,}	n or more times	/Ja{2,}vaScript/ matches "JaavaScript" or "JaaavaScript" but not "JavaScript"
{n,m}	At least n, at most m times	/Ja{2,3}vaScript/ matches "JaavaScript" or "JaaavaScript" but not "JavaScript"

Every metacharacter in Table 45-2 applies to the character immediately preceding it in the regular expression. Preceding characters may also be matching metacharacters from Table 45-1. For example, a match occurs for the following expression if the string contains two digits separated by one or more vowels:

```
/\d[aeiouy]+\d/
```

The last major contribution of metacharacters is helping the regular expression search a particular position in a string. By position, we don't mean something such as an offset — the matching functionality of regular expressions can tell us that. But, rather, whether the string to look for should be at the beginning or end of a larger string, or (if the `m` modifier is added to the regular expression assignment) at the beginning or end of a line of that larger string. Table 45-3 shows the positional metacharacters for JavaScript's regular expressions.

TABLE 45-3

JavaScript Regular Expression Positional Metacharacters

Character	Matches Located	Example
^	At beginning of a string or line	`/^Fred/` matches "Fred is OK" but not "I'm with Fred" or "Is Fred here?"
$	At end of a string or line	`/Fred$/` matches "I'm with Fred" but not "Fred is OK" or "Is Fred here?"

For example, you may want to make sure that a match for a Roman numeral is found only when it is at the start of a string, rather than when it is used inline somewhere else. If the document contains Roman numerals in an outline, you can match all the top-level items that are flush left with the document with a regular expression, such as the following:

```
/^[IVXMDCL]+\./m
```

This expression matches any combination of Roman numeral characters that begin at the start of a line and end with a period (the period is a special character in regular expressions, as shown in Table 45-1, so that you have to escape the period to offer it as a character), provided the Roman numeral is at the beginning of a line and has no tabs or spaces before it. There would also not be a match in a line that contains, for example, the phrase "see Part IV" because the Roman numeral is not at the beginning of a line.

Speaking of lines, a line of text is a contiguous string of characters delimited by a newline and/or carriage return (depending on the operating system platform). Word wrapping in `textarea` elements does not affect the starts and ends of true lines of text.

Grouping and backreferencing

Regular expressions obey most of the JavaScript operator precedence laws with regard to grouping by parentheses and the logical OR operator. One difference is that the regular expression OR operator is a single pipe character (`|`) rather than JavaScript's double pipe.

Parentheses have additional powers that go beyond influencing the precedence of calculation. Any set of parentheses (that is, a matched pair of left and right) stores the results of a found

match of the expression within those parentheses. Parentheses can be nested inside one another. Storage is accomplished automatically, with the data stored in an indexed array accessible to your scripts and to your regular expressions (although through different syntax). Access to these storage bins is known as *backreferencing*, because a regular expression can point backward to the result of an expression component earlier in the overall expression. These stored subcomponents come in handy for replace operations, as demonstrated later in this chapter.

Object Relationships

JavaScript has a lot going on behind the scenes when you create a regular expression and perform the simplest operation with it. As important as the regular expression language described earlier in this chapter is to applying regular expressions in your scripts, the JavaScript object interrelationships are perhaps even more important if you want to exploit regular expressions to the fullest.

The first concept to master is that two entities are involved: a regular expression instance object and the `RegExp` static object. Both objects are core objects of JavaScript and are not part of the document object model. Both objects work together, but have entirely different sets of properties that may be useful to your application.

When you create a regular expression (even via the `/.../` syntax), JavaScript invokes the `new RegExp()` constructor, much the way a `new Date()` constructor creates a date object around one specific date. The regular expression instance object returned by the constructor is endowed with several properties containing details of its data. At the same time, the single, static `RegExp` object maintains its own properties that monitor regular expression activity in the current window (or frame).

To help you see the typically unseen operations, we step you through the creation and application of a regular expression. In the process, we show you what happens to all of the related object properties when you use one of the regular expression methods to search for a match.

The starting text that we use to search through is the beginning of Hamlet’s soliloquy (assigned to an arbitrary variable named `mainString`):

```
var mainString = "To be, or not to be: That is the question:";
```

If our ultimate goal is to locate each instance of the word “be,” we must first create a regular expression that matches the word “be.” We set the regular expression up to perform a global search when eventually called upon (assigning the expression to an arbitrary variable named `re`):

```
var re = /\bbe\b/g;
```

To guarantee that only the complete word “be” is matched, we surround the letters with the word boundary metacharacters. The final “g” is the global modifier. The variable to which the

expression is assigned, re, represents a regular expression object whose properties and values are as follows:

Object.PropertyName	Value
re.source	"\bbe\bg"
re.global	true
re.ignoreCase	false
re.lastIndex	0

A regular expression's source property is the string consisting of the regular expression syntax (less the literal forward slashes). Each of the two possible modifiers, g and i, have their own properties, global and ignoreCase, whose values are Booleans indicating whether the modifiers are part of the source expression. The final property, lastIndex, indicates the index value within the main string at which the next search for a match should start. The default value for this property in a newly hatched regular expression is zero so that the search starts with the first character of the string. This property is read/write, so your scripts may want to adjust the value if they must have special control over the search process. As you see in a moment, JavaScript modifies this value over time if a global search is indicated for the object.

The RegExp constructor does more than just create regular expression objects. Like the Math object, the RegExp object is always "around" — one RegExp per window or frame — and tracks regular expression activity in a script. Its properties reveal what, if any, regular expression pattern matching has just taken place in the window. At this stage of the regular expression creation process, the RegExp object has only one of its properties set:

Object.PropertyName	Value
RegExp.input	
RegExp.multiline	false
RegExp.lastMatch	
RegExp.lastParen	
RegExp.leftContext	
RegExp.rightContext	
RegExp.$1	
...	
RegExp.$9	

The last group of properties ($1 through $9) is for storage of backreferences. But because the regular expression we define above doesn't have any parentheses in it, these properties are empty

for the duration of this examination and omitted from future listings in this "walk-through" section.

With the regular expression object ready to go, we invoke the `exec()` regular expression method, which looks through a string for a match defined by the regular expression. If the method is successful in finding a match, it returns a third object whose properties reveal a great deal about the item it found (we arbitrarily assign the variable `foundArray` to this returned object):

```
var foundArray = re.exec(mainString);
```

JavaScript includes a shortcut for the `exec()` method if you turn the regular expression object into a method:

```
var foundArray = re(mainString);
```

Normally, a script would check whether `foundArray` is `null` (meaning that there was no match) before proceeding to inspect the rest of the related objects. Because this is a controlled experiment, we know that at least one match exists, so we first look into some other results. Running this simple method has not only generated the `foundArray` data, but also altered several properties of the `RegExp` and regular expression objects. The following shows you the current stage of the regular expression object:

Object.PropertyName	Value
`re.source`	`"\bbe\bg"`
`re.global`	`true`
`re.ignoreCase`	`false`
`re.lastIndex`	`5`

The only change is an important one: The `lastIndex` value has bumped up to `5`. In other words, this one invocation of the `exec()` method must have found a match whose offset plus length of matching string shifts the starting point of any successive searches with this regular expression to character index `5`. That's exactly where the comma after the first "be" word is in the main string. If the global (`g`) modifier had not been appended to the regular expression, the `lastIndex` value would have remained at zero, because no subsequent search would be anticipated.

As the result of the `exec()` method, the `RegExp` object has had a number of its properties filled with results of the search:

Object.PropertyName	Value
`RegExp.input`	
`RegExp.multiline`	`False`
`RegExp.lastMatch`	`"be"`

Object.PropertyName	Value
`RegExp.lastParen`	
`RegExp.leftContext`	`"To "`
`RegExp.rightContext`	`", or not to be: That is the question:"`

From this object you can extract the string segment that was found to match the regular expression definition. The main string segments before and after the matching text are also available individually (in this example, the `leftContext` property has a space after "To"). Finally, looking into the array returned from the `exec()` method, some additional data is readily accessible:

Object.PropertyName	Value
`foundArray[0]`	`"be"`
`foundArray.index`	`3`
`foundArray.input`	`"To be, or not to be: That is the question:"`

The first element in the array, indexed as the zeroth element, is the string segment found to match the regular expression, which is the same as the `RegExp.lastMatch` value. The complete main string value is available as the input property. A potentially valuable piece of information to a script is the index for the start of the matched string found in the main string. From this last bit of data, you can extract from the found data array the same values as `RegExp.leftContext` (with `foundArray.input.substring(0, foundArray.index)`) and `RegExp.rightContext` (with `foundArray.input.substring(foundArray.index, foundArray[0].length)`).

Because the regular expression suggested a multiple execution sequence to fulfill the global flag, we can run the `exec()` method again without any change. Although the JavaScript statement may not be any different, the search starts from the new `re.lastIndex` value. The effects of this second time through ripple through the resulting values of all three objects associated with this method:

```
var foundArray = re.exec(mainString);
```

Results of this execution are as follows.

Object.PropertyName	Value
`re.source`	`"\bbe\bg"`
`re.global`	`True`
`re.ignoreCase`	`False`

Object.PropertyName	Value
re.lastIndex	19
RegExp.input	
RegExp.multiline	False
RegExp.lastMatch	"be"
RegExp.lastParen	
RegExp.leftContext	", or not to "
RegExp.rightContext	": That is the question:"
foundArray[0]	"be"
foundArray.index	17
foundArray.input	"To be, or not to be: That is the question:"

Because there was a second match, `foundArray` comes back again with data. Its index property now points to the location of the second instance of the string matching the regular expression definition. The regular expression object's `lastIndex` value points to where the next search would begin (after the second "be"). And the `RegExp` properties that store the left and right contexts have adjusted accordingly.

If the regular expression were looking for something less stringent than a hard-coded word, some other properties may also be different. For example, if the regular expression defined a format for a ZIP code, the `RegExp.lastMatch` and `foundArray[0]` values would contain the actual found ZIP codes, which would likely be different from one match to the next.

Running the same `exec()` method once more does not find a third match in our original `mainString` value, but the impact of that lack of a match is worth noting. First of all, the `foundArray` value is `null` — a signal to our script that no more matches are available. The regular expression object's `lastIndex` property reverts to zero, ready to start its search from the beginning of another string. Most importantly, however, the `RegExp` object's properties maintain the same values from the last successful match. Therefore, if you put the `exec()` method invocations in a repeat loop that exits after no more matches are found, the `RegExp` object still has the data from the last successful match, ready for further processing by your scripts.

Using Regular Expressions

Despite the seemingly complex hidden workings of regular expressions, JavaScript provides a series of methods that make common tasks involving regular expressions quite simple to use (assuming you figure out the regular expression syntax to create good specifications). In this section, we present examples of syntax for specific kinds of tasks for which regular expressions can be beneficial in your pages.

Is there a match?

We said earlier that you can use `string.indexOf()` or `string.lastIndexOf()` to look for the presence of simple substrings within larger strings. But if you need the matching power of regular expressions, you have two other methods to choose from:

```
regexObject.test(string)
string.search(regexObject)
```

The first is a regular expression object method, the second a string object method. Both perform the same task and influence the same related objects, but they return different values: a Boolean value for `test()` and a character offset value for `search()` (or `-1` if no match is found). Which method you choose depends on whether you need only a true/false verdict on a match or the location within the main string of the start of the substring.

Listing 45-1 demonstrates the `search()` method on a page that lets you get the Boolean and offset values for a match. Some default text and regular expression is provided (it looks for a five-digit number). You can experiment with other strings and regular expressions. Because this script creates a regular expression object with the `RegExp()` constructor method, you do not include the literal forward slashes around the regular expression.

LISTING 45-1

Looking for a Match

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Got a Match?</title>
      <style type="text/css">
         #instructions
         {
            font-weight:bold;
         }
      </style>
      <script type="text/javascript">
         function findIt(form)
         {
            var re = new RegExp(form.regexp.value);
            var input = form.main.value;
               if (input.search(re) != -1)
               {
                  form.output[0].checked = true;
               }
               else
               {
                  form.output[1].checked = true;
               }
```

```
        }
        function locateIt(form)
        {
           var re = new RegExp(form.regexp.value);
           var input = form.main.value;
           form.offset.value = input.search(re);
        }
     </script>
  </head>
  <body>
     <span id="instructions">Use a regular expression to test for
               the existence of a string:</span>
     <hr />
     <form>
        Enter some text to be searched:<br />
        <textarea name="main" cols="40" rows="4" wrap="virtual">
           The most famous ZIP code on Earth may be 90210.</textarea><br />
        Enter a regular expression to search:<br />
        <input type="text" name="regexp" size="30" value="\b\d\d\d\d\d\b" />
        <p>
           <input type="button" value="Is There a Match?"
                  onclick="findIt(this.form)" />
           <input type="radio"
                  name="output" />Yes
           <input type="radio" name="output" />No
        </p>
        <p>
           <input type="button" value="Where is it?"
                  onclick="locateIt(this.form)" />
           <input type="text" name="offset"
                  size="4" />
        </p>
        <p><input type="reset" /></p>
     </form>
  </body>
</html>
```

Note

The property assignment event handling technique used in the examples in this chapter is a deliberate simplification to make the code more readable. It is generally better to use the more modern approach of binding events using the `addEventListener()` **(NN6+/Moz/W3C) or** `attachEvent()` **(IE5+) methods. A modern cross-browser event handling technique is explained in detail in Chapter 32, "Event Objects."** ■

Getting information about a match

For the next application example, the task is not only to verify that a one-field date entry is in the desired format, but also to extract match components of the entry and use those values to perform further calculations in determining the day of the week. The regular expression in the

example that follows is a fairly complex one, because it performs some rudimentary range checking to make sure the user doesn't enter a month over 12 or a date over 31. What it does not take into account is the variety of lengths of each month. Nor does it take into account that the visitor might enter a date such as `1/30-2013` that would validate successfully, something we would not normally want. Though we would obviously want a much more complex regular expression for real world work, this example is complex enough for our purposes. But the regular expression, and method invoked with it, extract each date object component in such a way that you can perform additional validation on the range to make sure the user doesn't try to give September 31 days. Also be aware that this is not the only way to perform date validations in forms. Chapter 46, "Data-Entry Validation," offers additional thoughts on the matter, that work, for backward compatibility, without regular expressions.

Listing 45-2 contains a page that has a field for date entry, a button to process the date, and an output field for displaying a long version of the date, including the day of the week. At the start of the function that does all the work, we create two arrays to hold the plain language names of the months and days. These arrays are used only if the user enters a valid date.

Next comes the regular expression to be matched against the user entry. If you can decipher all the symbols, you see that three components are separated by potential hyphen or forward slash entries (`[\-\/]`). These symbols must be escaped in the regular expression. Importantly, each of the three component definitions is surrounded by parentheses, which are essential for the various objects created with the regular expression to remember their values for extraction later.

Here is a brief rundown of what the regular expression is looking for:

- A string beginning after a word break
- A string value for the month that contains a `1` plus a `0` through `2`; or an optional `0` plus a `1` through `9`
- A hyphen or forward slash
- A string value for the date that starts with a `0` plus a `1` through `9`; or starts with a `1` or `2` and ends with a `0` through `9`; or starts with a `3` and ends with `0` or `1`
- Another hyphen or forward slash
- A string value for the year that begins with `19` or `20`, followed by two digits

An extra pair of parentheses must surround the `19|20` segment to make sure that either one of the matching values is attached to the two succeeding digits. Without the parentheses, the logic of the expression attaches the digits only to 20. One other thing to notice: we allow the string value for the month to be followed by a hyphen or forward slash, and the same after the string value for the date.

For invoking the regular expression action, we select the `exec()` method, assigning the returned object to the variable `matchArray`. We can also use the `string.match()` method here. Only if the match is successful (that is, all conditions of the regular expression specification are met) does the major processing continue in the script.

The parentheses around the segments of the regular expression instruct JavaScript to assign each found value to a slot in the `matchArray` object. The month segment is assigned

to `matchArray[1]`, the date to `matchArray[2]`, and the year to `matchArray[3]` (`matchArray[0]` contains the entire matched string). Therefore, the script can extract each component to build a plain-language date string with the help of the arrays defined at the start of the function. We even use the values to create a new date object that calculates the day of the week for us. After we have all pieces, we concatenate them and assign the result to the value of the output field. If the regular expression `exec()` method doesn't match the typed entry with the expression, the script provides an error message in the field.

LISTING 45-2

Extracting Data from a Match

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Got a Match?</title>
      <style type="text/css">
         #instructions
         {
            font-weight:bold;
         }
      </style>
      <script type="text/javascript">
         function extractIt(form)
         {
            var months = ["January","February","March","April","May","June",
                          "July","August","September","October",
                          "November","December"];
            var days = ["Sunday","Monday","Tuesday","Wednesday","Thursday",
                        "Friday","Saturday"];
            var re = /\b(1[0-2]|0?[1-9])[\-\/](0?[1-9]|[12][0-9]
                     |3[01])[\-\/]((19|20)\d{2})/;
            var input = form.entry.value;
            var matchArray = re.exec(input);
               if (matchArray)
               {
                  var theMonth = months[matchArray[1] - 1] + " ";
                  var theDate = matchArray[2] + ", ";
                  var theYear = matchArray[3];
                  var dateObj = new Date(matchArray[3],
                                matchArray[1]-1,matchArray[2]);
                  var theDay = days[dateObj.getDay()] + " ";
                  form.output.value = theDay + theMonth + theDate + theYear;
               }
               else
               {
                  form.output.value = "An invalid date.";
               }
         }
      </script>
```

continued

LISTING 45-2 *(continued)*

```
   </head>
   <body>
      <span id="instructions">Use a regular expression to extract
                              data from a string:</span>
      <hr />
      <form>
         Enter a date in the format mm/dd/yyyy or mm-dd-yyyy:<br />
         <input type="text" name="entry" size="12" />
         <p>
            <input type="button" value="Extract Date Components"
                   onclick="extractIt(this.form)" />
         </p>
         <p>The date you entered was:<br />
            <input type="text" name="output" size="40" />
         </p>
         <p><input type="reset" /></p>
      </form>
   </body>
</html>
```

String replacement

To demonstrate using regular expressions for performing search-and-replace operations, we choose an application that may be of value to many page authors who have to display and format large numbers. Databases typically store large integers without commas. After five or six digits, however, such numbers are difficult for users to read. Conversely, if the user needs to enter a large number, commas help ensure accuracy.

Helping the procedure in JavaScript regular expressions is the `string.replace()` method (see Chapter 15, “The String Objects”). The method requires two parameters, a regular expression to search the string and a string to replace any match found in the string. The replacement string can be properties of the `RegExp` object as it stands after the most recent `exec()` method.

Listing 45-3 demonstrates how only a handful of script lines can do a lot of work when regular expressions handle the dirty work. The page contains three fields. Enter any number you want in the first one. A click of the Insert Commas button invokes the `commafy()` function in the page. The result is displayed in the second field. You can also enter a comma-filled number in the second field and click the Remove Commas button to see the inverse operation executed through the `decommafy()` function.

Specifications for the regular expression accept any positive or negative string of numbers. The keys to the action of this script are the parentheses around two segments of the regular expression. One set encompasses all characters not included in the second group: a required set of

three digits. In other words, the regular expression is essentially working from the rear of the string, chomping off three-character segments and inserting a comma each time a set is found.

A `while` repeat loop cycles through the string and modifies the string (in truth, the string object is not being modified, but, rather, a new string is generated and assigned to the old variable name). We use the `test()` method because we don't need the returned value of the `exec()` method. The `test()` method impacts the regular expression and `RegExp` object properties the same way as the `exec()` method, but more efficiently. The first time the `test()` method runs, the part of the string that meets the first segment is assigned to the `RegExp.$1` property; the second segment, if any, is assigned to the `RegExp.$2` property. Notice that we're not assigning the results of the `exec()` method to any variable, because for this application we don't need the array object generated by that method.

Next comes the tricky part. We invoke the `string.replace()` method, using the current value of the string (`num`) as the starting string. The pattern to search for is the regular expression defined at the head of the function. But the replacement string may look strange to you. The replacement string is replacing whatever the regular expression matches with the value of `RegExp.$1`, a comma, and the value of `RegExp.$2`. The `RegExp` object should not be part of the references used in the `replace()` method parameter. Because the regular expression matches the entire `num` string, the `replace()` method is essentially rebuilding the string from its components, plus adding a comma before the second component (the last freestanding three-digit section). Each `replace()` method invocation sets the value of `num` for the next time through the `while` loop and the `test()` method.

Looping continues until no matches occur — meaning that no more freestanding sets of three digits appear in the string. Then the results are written to the second field on the page.

LISTING 45-3

Replacing Strings via Regular Expressions

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Got a Match?</title>
      <style type="text/css">
         #instructions
         {
            font-weight:bold;
         }
      </style>
      <script type="text/javascript">
         function commafy(form)
         {
            var re = /(-?\d+)(\d{3})/;
            var num = form.entry.value;
```

continued

LISTING 45-3 *(continued)*

```
            while (re.test(num))
            {
               num = num.replace(re, "$1,$2");
            }
            form.commaOutput.value = num;
         }
         function decommafy(form)
         {
            var re = /,/g;
            form.plainOutput.value = form.commaOutput.value.replace(re,"");
         }
      </script>
   </head>
   <body>
      <span id="instructions">Use a regular expression to add/delete
                              commas from numbers:</span>
      <hr />
      <form>
         Enter a large number without any commas:<br />
         <input type="text" name="entry" size="15" />
         <p>
            <input type="button" value="Insert commas"
                   onclick="commafy(this.form)" />
         </p>
         <p>The comma version is:<br />
            <input type="text" name="commaOutput" size="20" />
         </p>
         <p>
            <input type="button" value="Remove commas"
                   onclick="decommafy(this.form)" />
         </p>
         <p>The un-comma version is:<br />
            <input type="text" name="plainOutput" size="15" />
         </p>
         <p><input type="reset" /></p>
      </form>
   </body>
</html>
```

Removing the commas is an even easier process. The regular expression is a comma with the global flag set. The `replace()` method reacts to the global flag by repeating the process until all matches are replaced. In this case, the replacement string is an empty string. For further examples of using regular expressions with string objects, see the discussions of the `string.match()`, `string.replace()`, and `string.split()` methods in Chapter 15.

Regular Expression Object

Properties	Methods	Event Handlers
`constructor`	`compile()`	
`global`	`exec()`	
`ignoreCase`	`test()`	
`lastIndex`		
`multiline`		
`source`		

Syntax

Accessing regular expression properties or methods:

```
regularExpressionObject.property | method([parameters])
```

Compatibility: WinIE4+, MacIE4+, NN4+, Moz1+, Safari1+, Opera+, Chrome+

About this object

The regular expression object is created on-the-fly by your scripts. Each regular expression object contains its own pattern and other properties. Deciding which object creation style to use depends on the way the regular expression will be used in your scripts.

When you create a regular expression with the literal notation (that is, with the two forward slashes), the expression is automatically compiled for efficient processing as the assignment statement executes. The same is true when you use the new `RegExp()` constructor and specify a pattern (and optional modifier flags) as a parameter. Whenever the regular expression is fixed in the script, use the literal notation; when some or all of the regular expression is derived from an external source (for example, user input from a text field), assemble the expression as a parameter to the new `RegExp()` constructor. A compiled regular expression should be used at whatever stage the expression is ready to be applied and reused within the script. Compiled regular expressions are not saved to disk or given any more permanence beyond the life of a document's script (that is, they die when the page unloads).

However, there may be times in which the specification for the regular expression changes with each iteration through a loop construction. For example, if statements in a `while` loop modify

the content of a regular expression, compile the expression inside the `while` loop, as shown in the following skeletal script fragment:

```
var srchText = form.search.value;
var re = new RegExp();  // empty constructor
while (someCondition)
{
   re.compile("\\s+" + srchText + "\\s+", "gi");
   statements that change srchText
}
```

Each time through the loop, the regular expression object is both given a new expression (concatenated with metacharacters for one or more white spaces on both sides of some search text whose content changes constantly) and compiled into an efficient object for use with any associated methods.

Properties

constructor

(See `string.constructor` in Chapter 15, "The String Object".)

global
ignoreCase

Value: Booleans Read-Only

Compatibility: WinIE5.5+, MacIE5+, NN4+, Moz1+, Safari1+, Opera+, Chrome+

These two properties reflect the regular expression `g` and `i` modifier flags, if any, associated with a regular expression. Settings are read-only and are determined as the object is created. Each property is independent of the other.

lastIndex

Value: Integer Read/Write

Compatibility: WinIE4+, MacIE4+, NN4+, Moz1+, Safari1+, Opera+, Chrome+

The `lastIndex` property indicates the index counter of the main string to be searched against the current regular expression object. When a regular expression object is created, this value is zero, meaning that there have been no searches with this object, and the default behavior of the first search is to start at the beginning of the string.

If the regular expression has the global modifier specified, the `lastIndex` property value advances to the next higher value, after the object is used in a method to match within a main string. The value is the position in the main string immediately after the previously matched string (and not including any character of the matched string). After locating the final match in a string, the method resets the `lastIndex` property to zero for the next time. You can also

influence the behavior of matches by setting this value on-the-fly. For example, if you want the expression to begin its search at the fourth character of a target string, you change the setting immediately after creating the object, as follows:

```
var re = /somePattern/;
re.lastIndex = 3;  // fourth character in zero-based index system
```

Related Items: Match result object index property

multiline

Value: Boolean Read-Only

Compatibility: WinIE5.5+, MacIE5, NN6+, Moz1+, Safari1+, Opera+, Chrome+

The `multiline` property reveals whether searches extend across multiple lines of a target string, as directed by the optional `m` modifier flag for a regular expression.

Related Items: `RegExp.multiline` property

source

Value: String Read-Only

Compatibility: WinIE4+, MacIE4+, NN4+, Moz1+, Safari1+, Opera+, Chrome+

The `source` property is simply the string representation of the regular expression used to define the object. This property is read-only.

Methods

compile("*pattern*", ["g" | "i" | "m"])

Returns: Regular expression object

Compatibility: WinIE4+, MacIE4+, NN4+, Moz1+, Safari+, Opera+, Chrome+

Use the `compile()` method to compile, on-the-fly, a regular expression whose content changes continually during the execution of a script. See the earlier discussion about this object for an example. Other regular expression creation statements (the literal notation and the `RegExp()` constructor that passes a regular expression) automatically compile their expressions.

exec("*string*")

Returns: Match array object or `null`

Compatibility: WinIE4+, MacIE4+, NN4+, Moz1+, Safari1+, Opera+, Chrome+

The `exec()` method examines the string passed as its parameter for at least one match of the specification defined for the regular expression object. The behavior of this method is similar to that of the `string.match()` method (although the `match()` method is more powerful in

completing global matches). Typically, a call to the `exec()` method is made immediately after the creation of a regular expression object, as in the following example:

```
var re = /somePattern/;
var matchArray = re.exec("someString");
```

Much happens as a result of the `exec()` method. Properties of both the regular expression object and the window's `RegExp` object are updated based on the success of the match. The method also returns an object that conveys additional data about the operation. Table 45-4 shows the properties of this returned object.

TABLE 45-4

Match Found Array Object Properties

Property	Description
`index`	Zero-based index counter of the start of the match inside the string
`input`	Entire text of original string
`[0]`	String of most recent matched characters
`[1],...[n]`	Parenthesized component matches

Some of the properties in this returned object echo properties in the `RegExp` object. The value of having them in the regular expression object is that their contents are safely stowed in the object, whereas the `RegExp` object and its properties may be modified soon by another call to a regular expression method. Items the two objects have in common are the `[0]` property (mapped to the `RegExp.lastMatch` property) and the `[1], ... [n]` properties (the first nine of which map to `RegExp.$1 ... RegExp.$9`). Although the `RegExp` object stores only nine parenthesized subcomponents, the returned array object stores as many as are needed to accommodate parenthesis pairs in the regular expression.

If no match turns up between the regular expression specification and the string, the returned value is `null`. See Listing 45-2 for an example of how this method can be applied. An alternate shortcut syntax may be used for the `exec()` method. Turn the regular expression into a function, as in

```
var re = /somePattern/;
var matchArray = re("someString");
```

Related Items: `string.match()` method

test("*string*")

Returns: Boolean

Compatibility: WinIE4+, MacIE4+, NN4+, Moz1+, Safari1+, Opera+, Chrome+

The most efficient way to find out if a regular expression has a match in a string is to use the `test()` method. Returned values are `true` if a match exists and `false` if not. In case you need more information, a companion method, `string.search()`, returns the starting index value of the matching string. See Listing 45-1 for an example of this method in action.

Related Items: `string.search()` method

RegExp Object

Properties	Methods	Event Handlers
`input`	(None)	(None)
`lastMatch`		
`lastParen`		
`leftContext`		
`multilane`		
`prototype`		
`rightContext`		
`$1, ... $9`		

Syntax

Accessing `RegExp` properties:

```
RegExp.property
```

Compatibility: WinIE4+, MacIE4+, NN4+, Moz1+, Safari+, Opera+, Chrome+

About this object

Beginning with version 4 browsers, the browser maintains a single instance of a `RegExp` object for each window or frame. The object oversees the action of all methods that involve regular expressions (including the few related string object methods). Properties of this object are exposed not only to JavaScript in the traditional manner, but also to a parameter of the method `string.replace()` for some shortcut access (see Listing 45-3).

With one `RegExp` object serving all regular expression-related methods in your document's scripts, you must exercise care in accessing or modifying this object's properties. You must make sure that the `RegExp` object has not been affected by another method. Most properties are subject to change as the result of any method involving a regular expression. This may be reason

enough to use the properties of the array object returned by most regular expression methods instead of the `RegExp` properties. The former stick with a specific regular expression object even after other regular expression objects are used in the same script. The `RegExp` properties reflect the most recent activity, irrespective of the regular expression object involved.

In the following listings, we supply the long, JavaScript-like property names. But each property also has an abbreviated, Perl-like manner to refer to the same properties. You can use these shortcut property names in the `string.replace()` method if you need the values. Also be aware that the `input`, `lastMatch`, `lastParen`, `leftContext`, `rightContext`, and `$1` through `$9` properties are deprecated in JavaScript 1.5.

Properties

input

Value: String Read/Write

Compatibility: WinIE4+, MacIE4+, NN4+, Moz-, Safari+, Opera-, Chrome+

The `RegExp.input` property is the main string against which a regular expression is compared in search of a match. In all of the example listings earlier in this chapter, the property was `null`. Such is the case when the main string is supplied as a parameter to the regular expression-related method.

But many text-related document objects have an unseen relationship with the `RegExp` object. If a `text`, `textarea`, `select`, or `link` object contains an event handler that invokes a function containing a regular expression, the `RegExp.input` property is set to the relevant textual data from the object. You don't have to specify any parameters for the event handler call or in the function called by the event handler. For `text` and `textarea` objects, the `input` property value becomes the content of the object; for the `select` object, it is the text (not the value) of the selected option; and for a link, it is the text highlighted in the browser associated with the link (and reflected in the link's `text` property).

Having JavaScript set the `RegExp.input` property for you may simplify your script. You can invoke either of the regular expression methods without having to specify the main string parameter. When that parameter is empty, JavaScript applies the `RegExp.input` property to the task. You can also set this property on-the-fly if you want. The short version of this property is `$_` (dollar sign underscore).

Related Items: Matching array object `input` property

multiline

Value: Boolean Read/Write

Compatibility: WinIE5.5+, MacIE5+, NN4+, Moz1+, Safari+, Opera-, Chrome+

The `RegExp.multiline` property determines whether searches extend across multiple lines of a target string. This property is automatically set to `true` when an event handler of a `textarea` triggers a function containing a regular expression. You can also set this property on-the-fly if

you want. The short version of this property is `$*`. This property (as distinct from the `multiline` property of an instance of a regular expression) is not defined in the ECMA-262 specification and is based on an early, incorrect implementation.

Related Items: Regular expression instance object `multiline` property

lastMatch

Value: String Read-Only

Compatibility: WinIE5.5+, MacIE5+, NN4+, Moz1+, Safari+, Opera-, Chrome+

After execution of a regular expression—related method, any text in the main string that matches the regular expression specification is automatically assigned to the `RegExp.lastMatch` property. This value is also assigned to the `[0]` property of the object array returned after the `exec()` and `string.match()` methods find a match. The short version of this property is `$&`.

Related Items: Matching array object `[0]` property

lastParen

Value: String Read-Only

Compatibility: WinIE5.5+, MacIE5+, NN4+, Moz1+, Safari+, Opera-, Chrome+

When a regular expression contains many parenthesized subcomponents, the `RegExp` object maintains a list of the resulting strings in the `$1,...$9` properties. You can also extract the value of the last matching parenthesized subcomponent with the `RegExp.lastParen` property, which is a read-only property. The short version of this property is `$+`.

Related Items: `RegExp.$1,...$9` properties

leftContext
rightContext

Value: String Read-Only

Compatibility: WinIE5.5+, MacIE5+, NN4+, Moz1+, Safari+, Opera+, Chrome+

After a match is found in the course of one of the regular expression methods, the `RegExp` object is informed of some key contextual information about the match. The `leftContext` property contains the part of the main string to the left of (up to but not including) the matched string. Be aware that the `leftContext` starts its string from the point at which the most recent search began. Therefore, for second or subsequent runs through the same string with the same regular expression, the `leftContext` substring varies widely from the first time through.

The `rightContext` consists of a string starting immediately after the current match and extending to the end of the main string. As subsequent method calls work on the same string and regular expression, this value obviously shrinks in length until no more matches are found. At this point, both properties revert to `null`. Note that Firefox in Windows returns a `null` for the `rightContext` property. The short versions of these properties are `` $` `` and `$'` for `leftContext` and `rightContext`, respectively.

prototype

(See `String.prototype` in Chapter 15, "The String Object".)

$1 ... $9

Value: String Read-Only

Compatibility: WinIE4+, MacIE4+, NN4+, Moz1+, Safari+, Opera+, Chrome+

As a regular expression method executes, any parenthesized results are stored in `RegExp`'s nine properties reserved for just that purpose (called backreferences). The same values (and any beyond the nine that `RegExp` has space for) are stored in the array object returned with the `exec()` and `string.match()` methods. Values are stored in the order in which the left parenthesis of a pair appears in the regular expression, regardless of nesting of other components. (Firefox in Windows returns a `null` for these properties.)

You can use these backreferences directly in the second parameter of the `string.replace()` method, without using the `RegExp` part of their address. The ideal situation is to encapsulate components that need to be rearranged or recombined with replacement characters. For example, the following script function turns a name that is last-name-first into first-name-last:

```
function swapEm()
{
   var re = /(\w+),\s*(\w+)/;
   var input = "Lincoln, Abraham";
   return input.replace(re,"$2 $1");
}
```

In the `replace()` method, the second parenthesized component (just the first name) is placed first, followed by a space and the first component. The original comma is discarded. You are free to combine these shortcut references as you like, including multiple times per replacement, if it makes sense to your application.

Related Items: Matching array object `[1] ... [n]` properties

Part VII

More JavaScript Programming

IN THIS PART

Chapter 46
Data-Entry Validation

Chapter 47
Scripting Java Applets and Plug-Ins

Chapter 48
Debugging Scripts

Chapter 49
Security and Netscape Signed Scripts

Chapter 50
Cross-Browser Dynamic HTML Issues

Chapter 51
Internet Explorer Behaviors

CHAPTER

Data-Entry Validation

IN THIS CHAPTER

Validating data as it is being entered

Validating data immediately prior to submission

Organizing complex data-validation tasks

Give users a field in which to enter data and you can be sure that some users will enter the wrong kind of data. Often the mistake is accidental — a slip of the pinkie on the keyboard; other times, users intentionally type an incorrect entry to test the robustness of your application. Whether you solicit a user's entry for client-side scripting purposes or for input into a server-based CGI or database, you should use JavaScript on the client to handle validation of the user's entry. Even for a form connected to a server application, it's far more efficient (from the perspective of bandwidth, server load, and execution speed) to let client-side JavaScript get the data straight before your server program deals with it.

Real-Time Versus Batch Validation

You have two opportunities to perform data-entry validation in a form: as the user enters data into the form and just before the form is submitted. We recommend you take advantage of both of these opportunities.

Real-time validation triggers

The most convenient time to catch an error is immediately after the user makes it — especially for a long form that requests a wide variety of information. You can make the user's experience less frustrating if you catch an entry mistake just after the user enters the information: his or her attention is already focused on the nature of the content (or some paper source material may already be in front of the user). It is much easier for the user to address the same information right away.

A reasonable question for the page author is how to trigger the real-time validation. Your initial thought is probably to tie the validation to the `onchange` handler. The problem with using `onchange` as the validation trigger is that a user can defeat the validation. A change event occurs only when the text of a field indeed changes as the user tabs or clicks out of the field. If the user is alerted about some bad entry in a field and doesn't fix the error, the change event doesn't fire again. In some respects, this is good because a user may have a legitimate reason for passing by a particular form field initially, with the intention of returning to the entry later. Because a user can defeat the `onchange` event handler trigger, a reasonable solution is to perform batch validation prior to submission.

An even better approach, and a more effective real-time solution, is available to modern browsers. We're referring to the strategy of letting keyboard events trigger validations. This is most helpful when you want to prevent some character(s) from being entered into a field. For example, if a field is supposed to contain only a positive integer value, you can use the `onkeypress` event handler of the text box to verify that the character just typed is a number. If the character is not a number, the event is trapped and no character reaches the text box. You should also alert the user in some way about what's going on.

Listing 46-1 demonstrates a simplified version of this kind of keyboard trapping. The message to the user is displayed in the status bar. Displaying the message there has the advantage of being less intrusive than an alert dialog box (and keeps the text insertion cursor in the text box), but it also means that users might not see the message. (This is especially true in Chrome because that browser has a transient status bar.) Also, recall from Chapter 27, "Window and Frame Objects," that even browsers with a status bar might not support scripting a message to it. The `onsubmit` event handler in the listing prevents a press of the Enter key in this one-field form from reloading this sample page.

LISTING 46-1

Allowing Only Numbers into a Text Box

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Letting Only Numbers Pass to a Form Field</title>
      <script type="text/javascript">
         function checkIt(evt)
         {
            evt = (evt) ? evt : window.event;
            var charCode = (evt.charCode) ? evt.charCode :
                        ((evt.which) ? evt.which : evt.keyCode);
               if (charCode > 31 && (charCode < 48 || charCode > 57))
               {
                  window.status = "This field accepts numbers only.";
                  return false;
               }
            status = "";
```

continued

LISTING 46-1 *(continued)*

```
            return true;
         }
      </script>
   </head>
   <body>
      <h1>Letting Only Numbers Pass to a Form Field</h1>
      <hr />
      <form onsubmit="return false">
         Enter any positive integer:
         <input type="text" name="numeric"
                onkeypress="return checkIt(event)" />
      </form>
   </body>
</html>
```

Note

The property assignment event handling technique used in this example and throughout this chapter is a deliberate simplification to make the code more readable. It is generally better to use the more modern approach of binding events using the `addEventListener()` **(NN6+/Moz/W3C) or** `attachEvent()` **(IE5+) methods. A modern cross-browser event handling technique is explained in detail in Chapter 32, "Event Objects."** ■

Aggressive keyboard event monitoring isn't practical for most validation actions, however. For example, if the user is supposed to enter an email address, you need to validate the complete entry for the presence of an @ symbol (through the `onchange` event handler). On the other hand, you can be granular about your validations and use both the `onchange` and `onkeypress` event handlers; you employ the latter for blocking invalid characters in email addresses (such as spaces).

Another interesting usage of keyboard event monitoring for validation is to offer helper messages in response to errant input, as opposed to preventing or rejecting the input outright. In this scenario, you typically place, beside the input control, a text label that will serve to display messages related to the validity of the input data. As the user enters information, the helper message dynamically updates in real time to provide input cues. You see this technique in action a bit later in the chapter, in Listing 46-8.

In the helper message approach to real-time validation, you aren't actually preventing any characters from being entered. For this reason, you may find the `onkeyup` event handler to be more direct than `onkeypress`; unlike `onkeypress`, `onkeyup` doesn't enable you to return `false` from the handler to prevent the entry of a character.

Batch mode validation

In all scriptable browsers, a form's `onsubmit` event handler cancels the submission if the handler evaluates to `return false`. Additional submission event cancellers include setting the IE4+

`event.returnValue` property to `false` and invoking the `evt.preventDefault()` method in Moz/W3C (see Chapter 32). You can see an example of the basic `return false` behavior in Listing 36-4 of Chapter 36. That example uses the results of a `window.confirm()` dialog box to determine the return value of the event handler. But you can also use a return value from a series of individual text box validation functions. If any one of the validations fails, the user is alerted and the submission is canceled.

Before you worry about two versions of validation routines loading down the scripts in your page, you'll be happy to know that you can reuse the same validation routines in both the real-time and batch validations. Later in this chapter, we demonstrate what we call industrial-strength data-entry validation, adapted from a real intranet application. But before you get there, you should learn about general validation techniques that you can apply to both types of validations.

Designing Filters

The job of writing data-validation routines essentially involves designing filters that weed out characters or entries that don't fit your programming scheme. Whenever your filter detects an incorrect entry, it should alert the user about the nature of the problem and enable the user to correct the entry.

Before you put a `text` or `textarea` object into your document that invites users to enter data, you must decide if any possible entry can disturb the execution of the rest of your scripts. For example, if your script must have a number from that field to perform calculations, you will need to filter out any entry that contains letters or punctuation — except for periods, if the program can accept floating-point numbers. Your task is to anticipate every possible entry users can make and allow only those entries your scripts can use.

Not every entry field needs a data-validation filter. For example, you may prompt a user for information that is eventually stored as a `document.cookie` or in a string database field on the server for future retrieval. If no further processing takes place on that information, you may not have to worry about the specific contents of the field.

One other design consideration is whether a text field is even the proper user interface element for the data required of the user. If the range of choices for a user entry is small (a dozen or fewer), a more sensible method is to avoid the data-entry problem altogether by turning that field into a `select` element. Your HTML attributes for the object ensure that you control the kind of entry made to that object. As long as your script knows how to deal with each of the options defined for that object, you're in the clear.

Building a Library of Filter Functions

A number of basic data-validation processes function repeatedly in form-intensive HTML pages. Filters for integers only, numbers only, empty entries, alphabet letters only, and the like are put to use every day. If you maintain a library of generalizable functions for each of your

data-validation tasks, you can drop these functions into your scripts at a moment's notice and be assured that they will work. You can also create the library of validation functions as a separate `.js` library file and link the scripts into any HTML file that needs them.

Making validation functions generalizable requires careful choice of wording and logic so that they return Boolean values that make syntactical sense when called from elsewhere in your scripts. As you see later in this chapter, when you build a larger framework around smaller functions, each function is usually called as part of an `if...else` conditional statement. Therefore, assign a name that fits logically as part of an `if` clause in plain language. For example, you can name a function that checks whether an entry is empty, `isEmpty()`. The calling statement for this function is

```
if (isEmpty(value)) { ... }
```

From a plain-language perspective, the expectation is that the function returns `true` if the passed value is empty. With this design, the statements nested in the `if` construction handle empty entry fields. We revisit this design later in this chapter when we start stacking multiple-function calls together in a larger validation routine.

To get you started with your library of validation functions, this chapter provides some building blocks that you can learn from and use as starting points for more specific filters of your own design. These functions utilize regular expressions, which are built into all modern JavaScript-powered browsers. Some of these functions are put to use in the JavaScript application in Chapter 53, "Application: A Lookup Table."

isEmpty()

The `isEmpty()` function, shown in Listing 46-2, checks to see whether an input value has been entered. Although you could design the function so that it checks for a null value, which would indicate emptiness, regular expressions provide an even cleaner solution. The regular expression approach examines whether the input string has one or more characters of any kind in it. This function returns `true` if the input is indeed empty, and `false` otherwise.

LISTING 46-2

Is an Entry Empty or Null?

```
// function to see if an input value has been
// entered at all
function isEmpty(inputStr)
{
   var re = /.+/;
      if (!inputStr.match(re))
      {
         return true;
      }
   return false;
}
```

isPosInteger()

The `isPosInteger()` function examines an input value to see if it is a positive integer, which includes only numbers from zero through nine, with no punctuation or other symbols. As Listing 46-3 reveals, regular expressions dramatically reduce the code in what would otherwise be a function that analyzes each character one at a time. The regular expression code simply checks to make sure that the string consists entirely of any number of numerals.

LISTING 46-3

Test for Positive Integers

```
// function to see if a suspected numeric input
// is a positive integer
function isPosInteger(inputStr)
{
   var re = /^\d*$/;
   inputStr = inputStr.toString();
      if (!inputStr.match(re))
      {
         return false;
      }
   return true;
}
```

isInteger()

The `isInteger()` function includes the entry of a negative integer value. Listing 46-4 shows that this function is identical to `isPosInteger()` except that the regular expression allows for one leading hyphen.

LISTING 46-4

Checking for Leading Minus Sign

```
// function to see if a suspected numeric input
// is a positive or negative integer
function isInteger(inputStr)
{
   var re = /^[-]?\d*$/;
   inputStr = inputStr.toString();
      if (!inputStr.match(re))
      {
         return false;
      }
   return true;
}
```

isNumber()

The isNumber() numeric filter function enables any integer or floating-point number to pass, while filtering out all others. All that distinguishes an integer from a floating-point number, for data-validation purposes, is the decimal point. A slight modification to the regular expression from Listing 46-4 — allowing a single decimal point between any number of numerals — turns the validation function into one that tests for any positive or negative number, with or without numbers to the right of the decimal, as shown in Listing 46-5.

LISTING 46-5

Testing for a Decimal Point

```
// function to see if a suspected numeric input
// is a positive or negative number
function isNumber(inputStr)
{
   var re = /^[-]?\d*\.?\d*$/;
   inputStr = inputStr.toString();
      if (!inputStr.match(re))
      {
         return false;
      }
   return true;
}
```

Custom validation functions

The listings shown so far in this chapter should give you plenty of source material to use in writing customized validation functions for your applications. Listing 46-6 shows an example of such an application-specific variation (extracted from the application in Chapter 53, "Application: A Lookup Table").

LISTING 46-6

A Custom Validation Function

```
// function to determine if value is in acceptable range
// for this application
function inRange(inputStr)
{
   num = parseInt(inputStr);
      if (num < 1 || num > 586 && num < 596 || num > 599 && num < 700 ||
          num > 728)
      {
         return false;
      }
   return true;
}
```

For this application, you need to verify if an entry falls within multiple ranges of acceptable numbers. The first statement of the `inRange()` function converts the incoming value to a number (through the `parseInt()` function) so that the value can be compared numerically against maximum and minimum values of several ranges within the database. The `if` condition then looks for values outside the acceptable range, so it can alert the user and return a `false` value.

The `if` condition is quite a long sequence of operators. As you noticed in the list of operator precedence (see Chapter 22, "JavaScript Operators"), the Boolean `and` operator (`&&`) has precedence over the Boolean `or` operator (`||`). Therefore, the `and` expressions evaluate first, followed by the `or` expressions. Using a "best practices" approach, otherwise unneeded parentheses may help you better visualize what's going on in that monster condition:

```
if (num < 1 || (num > 586 && num < 596) ||(num > 599 && num < 700) ||
    num > 728)
```

In other words, you exclude four possible ranges from consideration:

- Values less than 1
- Values between 586 and 596
- Values between 599 and 700
- Values greater than 728

Any value for which any one of these tests is true yields a Boolean `false` from this function. Combining all these tests into a single condition statement eliminates the need to construct an otherwise complex series of nested `if` conditions.

Combining Validation Functions

When you design a page that requests a particular kind of text input from a user, you often need to call more than one data-validation function to handle the entire job. For example, if you merely want to test for a positive integer entry, your validation should test for the presence of any entry, as well as the validation as an integer.

After you know the kind of permissible data that your script will use after validation, you're ready to plot the sequence of data validation. Because each page's validation task is different, we supply some guidelines to follow in your planning rather than prescribe a fixed route for all to take.

Our preferred sequence is to start with examinations that require less work, and increase the intensity of validation detective work with succeeding functions. We borrow this tactic from real life: When a lamp fails to turn on, we look for a pulled plug or a burned-out lightbulb before tearing the lamp's wiring apart to look for a short.

Using the data-validation sequence from the data-entry field (which must be a three-digit number within a specified range) in Chapter 53, we start with the test that requires the least amount of work: Is there an entry at all? After our script has ensured that an entry of some kind exists, it then checks whether that entry is "all numbers as requested of the user." If so, the script compares the number against the ranges of numbers in the database.

To make this sequence work efficiently, we create a master validation function consisting of nested `if...else` statements. Each `if` condition calls one of the generalized data-validation functions. Listing 46-7 shows the master validation function.

LISTING 46-7

Master Validation Function

```
// Master value validator routine
function isValid(inputStr)
{
  if (isEmpty(inputStr))
  {
     alert("Please enter a number into the field before clicking the button.");
     return false;
  }
  else
  {
     if (!isNumber(inputStr))
     {
       alert("Please make sure entries are numbers only.");
       return false;
     }
     else
     {
       if (!inRange(inputStr))
       {
         var msg = "Sorry, the number you entered is not part of our database.";
         msg += "Try another three-digit number.";
         alert(msg);
         return false;
       }
     }
   }
   return true;
}
```

This function, in turn, is called by the function that controls most of the work in this application. All that the main function wants to know is whether the entered number is valid. The details of validation are handed off to the `isValid()` function and its special-purpose validation testers.

We construct the logic in Listing 46-7 so that if the input value fails to be valid, the `isValid()` function alerts the user to the problem and returns `false`. That means we have to watch our `true` and `false` returns very carefully.

In the first validation test, an empty value is a bad thing; thus, when the `isEmpty()` function returns `true`, the `isValid()` function returns `false` because an empty string is not a valid entry. In the second test, a number value is good, so the logic has to flip 180 degrees. The `isValid()` function returns `false` only if the `isNumber()` function returns `false`. But

because `isNumber()` returns `true` when the value is a number, we switch the condition to test for the opposite results of the `isNumber()` function by negating the function name (preceding the function with the Boolean `not` (`!`) operator). This operator works only with a value that evaluates to a Boolean expression — which the `isNumber()` function always does. The final test for being within the desired range works on the same basis as `isNumber()`, using the Boolean `not` operator to turn the results of the `inRange()` function into the method that works best for this sequence.

Finally, if all validation tests fail to find bad or missing data, the entire `isValid()` function returns `true`. The statement that calls this function can now proceed, assured that the value entered by the user will work.

There is one additional point worth reinforcing, especially for newcomers. Although all these functions seem to be passing around the same input string as a parameter, notice that any changes made to the value (such as converting it to a string or number) are kept private to each function. These subfunctions never touch the original value in the calling function — they work only with copies of the original value. Therefore, even after the data validation takes place, the original value is in its original form and ready to go.

Date and Time Validation

You can scarcely open a bigger can of cultural worms than trying to program around the various date and time formats in use around the world. If you have ever looked through the possible settings in your computer's operating system, you can begin to understand the difficulty of this issue.

Trying to write JavaScript that accommodates all of the world's date and time formats for validation is an enormous, if not wasteful, challenge. It's one thing to check that a text box contains data in the form `xx/xx/xxxx`, but there are also valid value concerns that can get very messy on an international basis. For example, whereas North America typically uses the `mm/dd/yyyy` format, a large portion of the rest of the world uses `dd/mm/yyyy` (with different delimiter characters, as well). Therefore, how should your validation routine treat the entry `20/03/2002`? Is it incorrect because there are not 20 months in a year? Or is it correct as March 20th? To query a user for this kind of information, we suggest you divide the components into individually validated fields (separate text objects for hours and minutes), or make `select` element entries whose individual values can be assembled at submit time into a hidden date field, for processing by the database that needs the date information. (Alternately, you can let your server application handle the conversion.) Or you can provide helper text to clarify what format you're expecting (more on this in a moment).

Despite our encouragement to "divide and conquer" date entries, there may be situations in which you feel it's safe to provide a single text box for date entry (perhaps for a form that is used on a corporate intranet, strictly by users in one country). You see some more sophisticated code later in this chapter, but a quick-and-dirty solution runs along these lines:

1. Use the entered data (for example, in `mm/dd/yyyy` format) as a value passed to the `new Date()` constructor function.

2. From the newly created date object, extract each of the three components (month, day, and year) into separate numeric values (with the help of `parseInt()`).
3. Compare each of the extracted values against the corresponding date, month, and year values returned by the date object's `getDate()`, `getMonth()`, and `getFullYear()` methods (adjusting for zero-based values of `getMonth()`).
4. If all three pairs of values match, the entry is apparently valid.

Listing 46-8 puts this action sequence to work in such a way as to make the user interface highly intuitive. The `validDate()` function receives a reference to the field being checked. A copy of the field's value is made into a date object, and its components are read. If any part of the date conversion or component extraction fails (because of improperly formatted data or unexpected characters), a helper message is displayed next to the input box. This code assumes that the user enters a date in the `mm/dd/yyyy` format, which is the sequence in which the `Date` object constructor expects its data. If the user enters `dd/mm/yyyy`, the validation fails for any day beyond the twelfth.

LISTING 46-8

Simple Date Validation

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Simple Date Validation</title>
      <style type="text/css">
         .vm_invalid
         {
            font-style:italic;
            color:red;
         }
         .vm_valid
         {
            font-style:italic;
            color:green;
         }
      </style>
      <script type="text/javascript">
         function validDate(fld)
         {
            var testMo, testDay, testYr, inpMo, inpDay, inpYr, msg;
            var inp = fld.value;
            status = "";
            var re = /\b\d{1,2}[\/-]\d{1,2}[\/-]\d{4}\b/;
               if (re.test(inp))
               {
```

```
         var delimChar = (inp.indexOf("/") != -1) ? "/" : "-";
         var delim1 = inp.indexOf(delimChar);
         var delim2 = inp.lastIndexOf(delimChar);
         mo = parseInt(inp.substring(0, delim1), 10);
         day = parseInt(inp.substring(delim1+1, delim2), 10);
         yr = parseInt(inp.substring(delim2+1), 10);
         var testDate = new Date(yr, mo-1, day);
         if (testDate.getDate() == day)
         {
             if (testDate.getMonth() + 1 == mo)
             {
                 if (testDate.getFullYear() == yr)
                 {
                     msg = "";
                 }
                 else
                 {
                     msg = "Invalid year.";
                 }
             }
             else
             {
                 msg = "Invalid month.";
             }
         }
         else
         {
             msg = "Invalid date.";
         }
      }
      else
      {
         msg = "Enter as mm/dd/yyyy.";
      }
   var validateMsg = document.getElementById("validateMsg");
      if (msg)
      {
         // there's a message, so something failed
         // change the validation message style and content
         validateMsg.className = "vm_invalid";
         validateMsg.innerHTML = msg;
         while(validateMsg.firstChild)
         {
            validateMsg.removeChild(validateMsg.firstChild);
         }
         validateMsg.appendChild(document.createTextNode(msg));
         // make sure focus stays with the input control
         fld.focus();
      }
      else
```

continued

LISTING 46-8 *(continued)*

```
            {
               // everything's OK
               // change the validation message style and content
               validateMsg.className = "vm_valid";
               while(validateMsg.firstChild)
               {
                  validateMsg.removeChild(validateMsg.firstChild);
               }
               validateMsg.appendChild(document.createTextNode("Date OK."));
               // show date string in status bar
               window.status = testDate.toLocaleDateString();
            }
         }
      </script>
   </head>
   <body>
    <h1>Simple Date Validation</h1>
    <hr />
    <form name="entryForm" onsubmit="return false">
      Enter any date (mm/dd/yyyy):
      <input type="text" id="test" name="startDate" onkeyup="validDate(this)" />
       <span id="validateMsg"></span>
    </form>
   </body>
</html>
```

Note

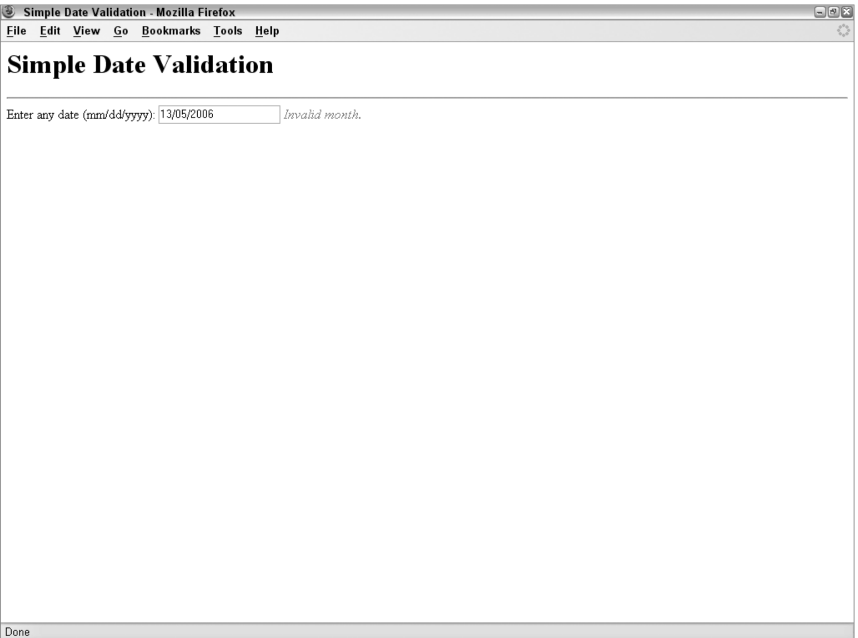
As handy as it may be, in a strict W3C approach to JavaScript, you wouldn't use the `innerHTML` property since it isn't officially part of the W3C standard. However, it is often too powerful a convenience property to ignore, as much of the code throughout this book is a testament. The book shows the W3C node manipulation alternative to `innerHTML` in some examples. Refer to Chapter 29, "Document and Body Objects," for a thorough explanation, and examples of the W3C alternative to `innerHTML`. ■

Although Figure 46.1 reveals the helper text next to the text input box, it doesn't show the effectiveness of real-time data validation. Note in the code how the `validDate()` function is tied to the `onkeyup` event, which causes the function to be called in response to every key press (key release, actually). The net effect is that the input is checked continuously as the date is entered, with helper text reflecting any problems along the way.

If this real-time validation approach has a familiar feel to it, then you're probably thinking about any of several modern applications that use Ajax, and similar validation techniques, to provide a richer data entry experience. In the case of Ajax, the real-time validation involves checking user input against data obtained from a server in an XML format, as opposed to simply performing client-side validity checks, as in this example.

FIGURE 46-1

Real-time data validation can dramatically improve any user interface.



Cross-Reference

For more on Ajax, check out Chapter 39, "Ajax, E4X, and XML Objects." ■

During batch validations, it is especially helpful to the user if your code — upon discovering an invalid entry — not only brings focus to the subject text field, but also selects the content for the user. By preselecting the entire field, you make it easy for the user to just retype the data into the field for another attempt (or to begin using the left and right arrow keys to move the insertion cursor for editing). The reverse type on the field text also helps bring attention to the field. (Not all operating systems display a special rectangle around a focused text field.) Form fields have both `focus()` and `select()` methods, which you should invoke for the subject field in that order.

In the case of real-time validation — which means you're validating user input literally as it is being entered — it is only necessary to keep the input control focused. Forcing the selection of the control wouldn't work because each subsequent key press would overwrite the previously entered text. The example in Listing 46-8 uses the focus-only technique to make sure the text input control retains focus as the text within it is validated in real time.

An Industrial-Strength Validation Solution

I had the privilege of working on a substantial intranet project that included dozens of forms, often with two or three different kinds of forms displayed simultaneously within a frameset. Data-entry accuracy was essential to the validity of the entire application. My task was to devise a data-entry validation strategy that not only ensured accurate entry of data types for the underlying (SQL) database, but also intelligently prompted users who made mistakes in their data entry.

Structure

From the start, the validation routines were to be in a client-side library linked in from an external `.js` file that would allow all forms to share the validation functions. Because there were multiple forms displayed in a frameset, it would prove too costly in download time and memory requirements to include the `validations.js` file in every frame's document. Therefore, the library was moved, to load in with the frameset. The `<script src="validations.js"></script>` tag set went in the Head portion of the framesetting document.

This logical placement presented a small challenge for the workings of the validations because there had to be two-way conversations between a validation function (in the frameset) and a form element (nested in a frame). The mechanism required that a reference to the frame containing the form element be passed as part of the validation routine so that the validation script could make corrections, automatic formatting, and erroneous field selections from the frameset document's script. (In other words, the frameset script needed a path back to the form element making the validation call.)

Dispatch mechanism

From the specification drawn up for the application, it is clear that there are more than two dozen specific types of validations across all the forms. Moreover, multiple programmers work on different forms. It is helpful to standardize the way validations are called, regardless of the validation type (number, string, date, phone number, and so on).

My idea was to create one `validate()` function that contained parameters for the current frame, the current form element, and the type of validation to perform. This would make it clear to anyone reading the code later that an event handler calling `validate()` performed validation, and that the details of the code were in the `validations.js` library file.

In `validations.js`, I converted the string name of a validation type into the name of the function that performs the validation, in order to make this idea work. As a bridge between the two, I created what I called a *dispatch lookup table* for all the primary validation routines that would be called from the forms. Each entry of the lookup table had a label consisting of the name of the validation and a method that invoked the function. Listing 46-9 shows an excerpt of the entire lookup table creation mechanism.

LISTING 46-9

Creating the Dispatch Lookup Table

```
/*
   Begin validation dispatching mechanism
*/
function dispatcher(validationFunc)
{
   this.doValidate = validationFunc;
}
var dispatchLookup = new Array();
dispatchLookup["isNotEmpty"] = new dispatcher(isNotEmpty);
dispatchLookup["isPositiveInteger"] = new dispatcher(isPositiveInteger);
dispatchLookup["isDollarsOnly8"] = new dispatcher(isDollarsOnly8);
dispatchLookup["isUSState"] = new dispatcher(isUSState);
dispatchLookup["isZip"] = new dispatcher(isZip);
dispatchLookup["isExpandedZip"] = new dispatcher(isExpandedZip);
dispatchLookup["isPhone"] = new dispatcher(isPhone);
dispatchLookup["isConfirmed"] = new dispatcher(isConfirmed);
dispatchLookup["isNY"] = new dispatcher(isNY);
dispatchLookup["isNum16"] = new dispatcher(isNum16);
dispatchLookup["isM90_M20Date"] = new dispatcher(isM90_M20Date);
dispatchLookup["isM70_0Date"] = new dispatcher(isM70_0Date);
dispatchLookup["isM5_P10Date"] = new dispatcher(isM5_P10Date);
dispatchLookup["isDateFormat"] = new dispatcher(isDateFormat);
```

Each entry of the array is assigned a `dispatcher` object, whose custom object constructor assigns a function reference to the object's `doValidate()` method. For these assignment statements to work, their corresponding functions must be defined earlier in the document. You can see some of these functions later in this section.

The link between the form elements and the dispatch lookup table is the `validate()` function, shown in Listing 46-10. A call to `validate()` requires a minimum of three parameters, as shown in the following example:

```
<input type="text" name="phone" size="10"
       onchange="parent.validate(window, this, 'isPhone')" />
```

The first parameter is a reference to the frame containing the document that is calling the function (passed as a reference to the current window). The second is a reference to the form control element itself (using the `this` operator). After that, you see one or more individual validation function names as strings. This design allows more than one type of validation to take place with each call to `validate()`. (For example, in case a field must check both for a data type and that the field is not empty.)

LISTING 46-10

Main Validation Function

```
// main validation function called by form event handlers
function validate(frame, field, method)
{
   gFrame = frame;
   gField =  window.frames[frame.name].document.forms[0].elements[field.name];
   var args = validate.arguments;
   for (i = 2; i < args.length; i++)
   {
      if (!dispatchLookup[args[i]].doValidate())
      {
         return false;
      }
   }
   return true;
}
```

In the `validate()` function, the frame reference is assigned to a global variable that is declared at the top of the `validations.js` file. Validation functions in this library need this information to build a reference back to a companion field required of some validations (explained later in this section). A second global variable contains a reference to the calling form element. Because the form element reference by itself does not contain information about the frame in which it lives, the script must build a reference out of the information passed as parameters. The reference must work from the framesetting document down to the frame, its form, and form element name. Therefore, I use the `frame` and `field` object references to get their respective names (within the `frames` and `elements` arrays) to assemble the text field's object reference; the resulting value is assigned to the `gField` global variable. I choose to use global variables in this case because passing these two values to numerous nested validation functions could be difficult to track reliably. Instead, the only parameter passed to specific validation functions is the value under test.

Next, the script creates an array of all arguments passed to the `validate()` function. A `for` loop starts with an index value of `2`, the third parameter containing the first validation function name. For each one, the named item's `doValidate()` method is called. If the validation fails, this function returns `false`; but if all validations succeed, this function returns `true`. Later you see that this function's returned value is the one that allows or disallows a form submission.

Sample validations

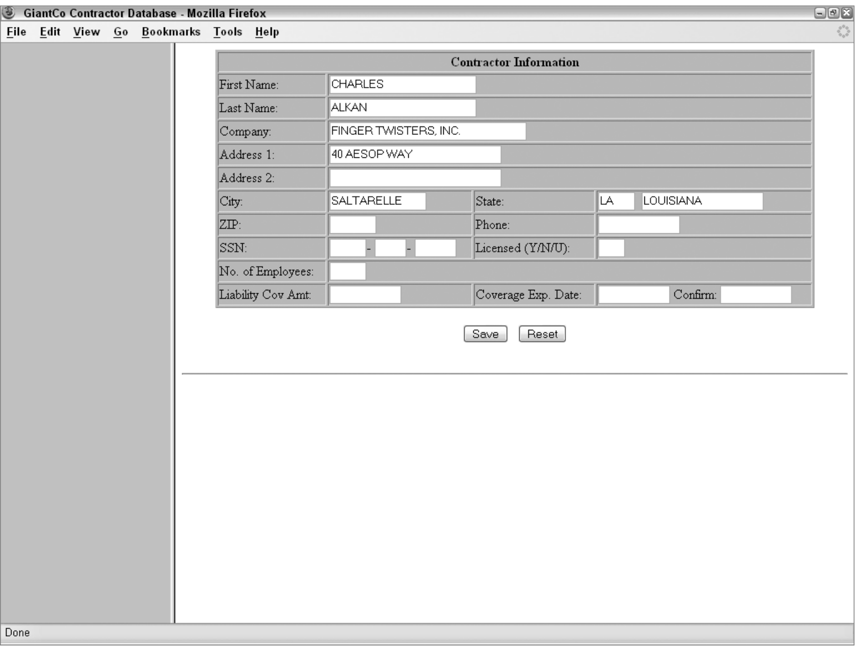
Above the dispatching mechanism in the `validations.js` are the validation functions themselves. Many of the named validation functions have supporting utility functions that other named validation functions often use. Because of the eventual large size of this library file (the production version was about 40KB), I organized the functions into two groups: the named

functions first, and the utility functions below them (but still before the dispatching mechanism at the bottom of the document).

To demonstrate how some of the more common data types are validated for this application, I show several validation functions and, where necessary, their supporting utility functions. Figure 46.2 shows a sample form that takes advantage of these validations. (You have a chance to try it later in this chapter.) When you are dealing with critical corporate data, you must go to extreme lengths to ensure valid data. And to help users see their mistakes quickly, you need to build some intelligence into validations where possible.

FIGURE 46-2

Sample form for industrial-strength validations.



U.S. state name

The design specification for forms that require entry of a U.S. state calls for entry of the state's two-character abbreviation. A companion field to the right displays the entire state name as user feedback verification. The `onchange` event handler not only calls the validation, but also feeds the focus to the field following the expanded state field, so users are less likely to type into it.

Before the validation can even get to the expansion part, it must first check that the entry is a valid, two-letter abbreviation. Because we need both the abbreviation and the full state name for this validation, we create an array of all the states, using each state abbreviation as the index label for each entry. Listing 46-11 shows that array creation.

LISTING 46-11

Creating a U.S. States Array

```
// States array
var USStates = new Array(51);
USStates["AL"] = "ALABAMA";
USStates["AK"] = "ALASKA";
USStates["AZ"] = "ARIZONA";
USStates["AR"] = "ARKANSAS";
USStates["CA"] = "CALIFORNIA";
USStates["CO"] = "COLORADO";
USStates["CT"] = "CONNECTICUT";
USStates["DE"] = "DELAWARE";
USStates["DC"] = "DISTRICT OF COLUMBIA";
USStates["FL"] = "FLORIDA";
USStates["GA"] = "GEORGIA";
USStates["HI"] = "HAWAII";
USStates["ID"] = "IDAHO";
USStates["IL"] = "ILLINOIS";
USStates["IN"] = "INDIANA";
USStates["IA"] = "IOWA";
USStates["KS"] = "KANSAS";
USStates["KY"] = "KENTUCKY";
USStates["LA"] = "LOUISIANA";
USStates["ME"] = "MAINE";
USStates["MD"] = "MARYLAND";
USStates["MA"] = "MASSACHUSETTS";
USStates["MI"] = "MICHIGAN";
USStates["MN"] = "MINNESOTA";
USStates["MS"] = "MISSISSIPPI";
USStates["MO"] = "MISSOURI";
USStates["MT"] = "MONTANA";
USStates["NE"] = "NEBRASKA";
USStates["NV"] = "NEVADA";
USStates["NH"] = "NEW HAMPSHIRE";
USStates["NJ"] = "NEW JERSEY";
USStates["NM"] = "NEW MEXICO";
USStates["NY"] = "NEW YORK";
USStates["NC"] = "NORTH CAROLINA";
USStates["ND"] = "NORTH DAKOTA";
USStates["OH"] = "OHIO";
USStates["OK"] = "OKLAHOMA";
USStates["OR"] = "OREGON";
USStates["PA"] = "PENNSYLVANIA";
USStates["RI"] = "RHODE ISLAND";
USStates["SC"] = "SOUTH CAROLINA";
USStates["SD"] = "SOUTH DAKOTA";
USStates["TN"] = "TENNESSEE";
```

```
USStates["TX"] = "TEXAS";
USStates["UT"] = "UTAH";
USStates["VT"] = "VERMONT";
USStates["VA"] = "VIRGINIA";
USStates["WA"] = "WASHINGTON";
USStates["WV"] = "WEST VIRGINIA";
USStates["WI"] = "WISCONSIN";
USStates["WY"] = "WYOMING";
```

The existence of this array comes in handy in determining if the user enters a valid, two-letter abbreviation. Listing 46-12 shows the actual `isUSState()` validation function that puts this array to work.

The function's first task is to assign an uppercase version of the entered value to a local variable (`inputStr`), which is the value being analyzed throughout the rest of the function. If the user enters something in the field (`length > 0`), but no entry in the `USStates` array exists for that value, the entry is not a valid state abbreviation. Time to go to work to help out the user.

LISTING 46-12

Validation Function for U.S. States

```
// input value is a U.S. state abbreviation; set entered value to all uppercase
// also set companion field (NAME="<xxx>_expand") to full state name
function isUSState()
{
   var inputStr = gField.value.toUpperCase();
     if (inputStr.length > 0 && USStates[inputStr] == null)
     {
       var msg = "";
       var firstChar = inputStr.charAt(0);
         if (firstChar == "A")
         {
            msg += "\n(Alabama = AL; Alaska = AK; Arizona = AZ; Arkansas = AR)";
         }
         if (firstChar == "D")
         {
            msg += "\n(Delaware = DE; District of Columbia = DC)";
         }
         if (firstChar == "I")
         {
            msg += "\n(Idaho = ID; Illinois = IL; Indiana = IN; Iowa = IA)";
         }
         if (firstChar == "M")
         {
            msg += "\n(Maine = ME; Maryland = MD; Massachusetts = MA;  " +
                   "Michigan = MI; Minnesota = MN; Mississippi = MS; " +
                   "Missouri = MO; Montana = MT)";
```

continued

LISTING 46-12 *(continued)*

```
         }
         if (firstChar == "N")
         {
            msg += "\n(Nebraska = NE; Nevada = NV)";
         }
      alert("Check the spelling of the state abbreviation." + msg);
      gField.focus();
      gField.select();
      return false;
   }
   gField.value = inputStr;
   var expandField =
      window.frames[gFrame.name].document.forms[0].elements[gField.name +
      "_expand"];
   expandField.value = USStates[inputStr];
   return true;
}
```

The function assumes that the user tried to enter a valid state abbreviation but either had incorrect source material or momentarily forgot a particular state's abbreviation. Therefore, the function examines the first letter of the entry. If that first letter is any one of the five identified as causing the most difficulty, a legend for all states beginning with that letter is assigned to the `msg` variable (for running on newer browsers only, a `switch` construction is preferred). An alert message displays the generic alert, plus any special legend, if one is assigned to the `msg` variable. When the user closes the alert, the field has focus and its text is selected. The function returns `false` at this point.

If, on the other hand, the abbreviation entry is a valid one, the field is handed the uppercase version of the entry. The script then uses the two global variables set in `validate()` to create a reference to the expanded display field (whose name must be the same as the entry field, plus `"_expand"`). That expanded display field is then supplied the `USStates` array entry value corresponding to the abbreviation label. All is well with this validation, so it returns `true`.

You can see here that the so-called validation routine is doing far more than simply checking the validity of the data. By communicating with the field, converting its contents to uppercase, and talking to another field in the form, a simple call to the validation function yields a lot of mileage.

Date validation

Many of the forms in this application have date fields. In fact, dates are an important part of the data maintained in the database behind the forms. All users of this application are familiar with standard date formats in use in the United States, so I don't have to worry about the possibility of cultural variations in date formats. Even so, I want the date entry to accommodate the common date formats, such as `mmddyyyy`, `mm/dd/yyyy`, and `mm-dd-yyyy` (as well as accommodate two-digit year entries spanning 1930 to 2029).

The plan also calls for going further in helping users enter dates within certain ranges. For example, a field used for a birth date (the listings are for medical professionals) should recommend dates starting no more than 90 years, and no less than 20 years, from the current date. And to keep this application running well into the future, the ranges should be on a sliding scale from the current year, no matter when it might be. Whatever the case, the date range validation is only a recommendation and not a transaction stopper.

Rather than create separate validation functions for each date field, I create a system of reusable validation functions for each date range (several fields on different forms require the same date ranges). Each one of these individual functions calls a single, generic date-validation function that handles the date-range checking. Listing 46-13 shows a few examples of these individual range-checking functions.

LISTING 46-13

Date Range Validations

```
// Date Minus 90/Minus 20
function isM90_M20Date()
{
   if (gField.value.length == 0)
   {
      return true;
   }
   var thisYear = getTheYear();
   return isDate((thisYear - 90),(thisYear - 20));
}

// Date Minus 70/Minus 0
function isM70_0Date()
{
   if (gField.value.length == 0)
   {
      return true;
   }
   var thisYear = getTheYear();
   return isDate((thisYear - 70),(thisYear));
}

// Date Minus 5/Plus 10
function isM5_P10Date()
{
   if (gField.value.length == 0)
   {
      return true;
   }
   var thisYear = getTheYear();
   return isDate((thisYear - 5),(thisYear + 10));
}
```

The naming convention I create for the functions includes the two range components relative to the current date. A letter “M” means the range boundary is minus a number of years from the current date; “P” means the range is plus a number of years. If the boundary should be the current year, a zero is used. Therefore, the `isM5_P10Date()` function performs range checking for boundaries between five years before and 10 years after the current year.

Before performing any range checking, each function makes sure there is some value to validate. If the field entry is empty, the function returns `true`. This is fine here because dates are not required when the data is unknown.

Next, the functions get the current four-digit year. The code here had to work originally with browsers that did not have the `getFullYear()` method available yet. Therefore, the Y2K fix described in Chapter 17, “The Date Object,” was built into the application:

```
function getTheYear()
{
   var thisYear = (new Date()).getYear();
   thisYear = (thisYear < 100) ? thisYear + 1900 : thisYear;
   return thisYear;
}
```

The final call from the range validations is to a common `isDate()` function, which handles not only the date range validation but also the validation for valid dates (for example, making sure that September has only 30 days). Listing 46-14 shows this monster-sized function. Because of the length of this function, I interlace commentary within the code listing.

LISTING 46-14

Primary Date Validation Function

```
// date field validation (called by other validation functions that specify
// minYear/maxYear)
function isDate(minYear,maxYear,minDays,maxDays)
{
   var inputStr = gField.value;
```

To make it easier to work with dates supplied with delimiters, I first convert hyphen delimiters to slash delimiters. The pre-regular expression `replaceString()` function is the same one described in Chapter 15; it is located in the utility functions part of the `validations.js` file.

```
   // convert hyphen delimiters to slashes
   while (inputStr.indexOf("-") != -1)
   {
      inputStr = replaceString(inputStr,"-","/");
   }
```

For validating whether the gross format is OK, I check whether zero or two delimiters appear. If the value contains only one delimiter, the overall formatting is not acceptable. The error alert shows models for acceptable date-entry formats.

```
    var delim1 = inputStr.indexOf("/");
    var delim2 = inputStr.lastIndexOf("/");
       if (delim1 != -1 && delim1 == delim2)
       {
         // there is only one delimiter in the string
         alert("The date entry is not in an acceptable format.\n\nYou can
  enter dates in the following formats: mmddyyyy, mm/dd/yyyy, or mm-dd-yyyy.
  (If the month or date data is not available, enter \'01\' in the
  appropriate location.)");
         gField.focus();
         gField.select();
         return false;
       }
```

If there are two delimiters, I tear apart the string into components for month, day, and year. Because two-digit entries can begin with zeros, I make sure the `parseInt()` functions specify base-10 conversions.

```
       if (delim1 != -1)
       {
         // there are delimiters; extract component values
         var mm = parseInt(inputStr.substring(0,delim1),10);
         var dd = parseInt(inputStr.substring(delim1 + 1,delim2),10);
         var yyyy = parseInt(inputStr.substring(delim2 + 1, inputStr.length),10);
```

For no delimiters, I tear apart the string and assume two-digit entries for the month and day, and two or four digits for the year.

```
       }
       else
       {
          // there are no delimiters; extract component values
          var mm = parseInt(inputStr.substring(0,2),10);
          var dd = parseInt(inputStr.substring(2,4),10);
          var yyyy = parseInt(inputStr.substring(4,inputStr.length),10);
       }
```

The `parseInt()` functions reveal whether any entry is not a number by returning `NaN`, so I check whether any of the three values is not a number. If so, an alert signals the formatting problem and supplies acceptable models.

```
          if (isNaN(mm) || isNaN(dd) || isNaN(yyyy))
          {
            // there is a non-numeric character in one of the component values
            alert("The date entry is not in an acceptable format.\n\nYou can
                   enter dates in the following formats: mmddyyyy, mm/dd/yyyy, or

                   mm-dd-yyyy.");
            gField.focus();
            gField.select();
            return false;
          }
```

Next, I perform some gross range validation on the month and date to make sure that months are entered from 1 to 12 and dates from 1 to 31. I take care of aligning exact month lengths later.

```
        if (mm < 1 || mm > 12)
        {
           // month value is not 1 thru 12
           alert("Months must be entered between the range
                  of 01 (January) and 12 (December).");
           gField.focus();
           gField.select();
           return false;
        }
        if (dd < 1 || dd > 31)
        {
           // date value is not 1 thru 31
           alert("Days must be entered between the range of 01 and a
                  maximum of 31 (depending on the month and year).");
           gField.focus();
           gField.select();
           return false;
        }
```

Before getting too deep into the year validation, I convert any two-digit year within the specified range to its four-digit equivalent.

```
        // validate year, allowing for checks between year ranges
        // passed as parameters from other validation functions
        if (yyyy < 100)
        {
           // entered value is two digits, which we allow for 1930-2029
           if (yyyy >= 30)
           {
              yyyy += 1900;
           }
           else
           {
              yyyy += 2000;
           }
        }

  var today = new Date();
```

I designed this function to work with a pair of year ranges or date ranges (so many days before and/or after today). If the function is passed date ranges, the first two parameters must be passed as `null`. This first batch of code works with the date ranges (because the `minYear` parameter is `null`).

```
  if (!minYear)
  {
     // function called with specific day range parameters
     var dateStr = new String(monthDayFormat(mm) +
                              "/" + monthDayFormat(dd) +
```

```
                            "/" + yyyy);
      var testDate = new Date(dateStr);
         if (testDate.getTime() < (today.getTime() +
            (minDays * 24 * 60 * 60 * 1000)))
         {
            alert("The most likely range for this entry begins " + minDays +
                  " days from today.");
         }
         if (testDate.getTime() > today.getTime() +
             (maxDays * 24 * 60 * 60 * 1000))
         {
            alert("The most likely range for this entry ends " + maxDays +
                  " days from today.");
         }
   }
```

You can also pass hard-wired, four-digit years as parameters. The following branch compares the entered year against the range specified by those passed year values.

```
   else if (minYear && maxYear)
   {
      // function called with specific year range parameters
      if (yyyy < minYear || yyyy > maxYear)
      {
         // entered year is outside of range passed from calling function
         alert("The most likely range for this entry is between the years " +
               minYear + " and " + maxYear +
               ".  If your source data indicates a date outside
                   this range, then enter that date.");
      }
   }
   else
```

For year parameters passed as positive or negative year differences, I begin processing by getting the four-digit year for today's date. Then I compare the entered year against the passed range values. If the entry is outside the desired range, an alert reveals the preferred year range within which the entry should fall. But the function does not return any value here because an out-of-range value is not critical for this application.

```
   {
      // default year range (now set to (this year - 100) and (this year + 25))
      var thisYear = today.getYear();
         if (thisYear < 100)
         {
            thisYear += 1900;
         }
      if (yyyy < minYear || yyyy > maxYear)
      {
         alert("It is unusual for a date entry to be before " + minYear +
               " or after " + maxYear + ". Please verify this entry.");
      }
   }
```

One more important validation is to make sure that the entered date is valid for the month and year. Therefore, the various date components are passed to functions to check against month lengths, including the special calculations for the varying length of February. Listing 46-15 shows these functions. The alert messages they display are smart enough to inform the user what the maximum date is for the entered month and year.

```
if (!checkMonthLength(mm,dd))
{
   gField.focus();
   gField.select();
   return false;
}
if (mm == 2)
{
   if (!checkLeapMonth(mm,dd,yyyy))
   {
      gField.focus();
      gField.select();
      return false;
   }
}
```

The final task is to reassemble the date components into a format that the database wants for its date storage, and stuff it into the form field. If the user enters an all-number or hyphen-delimited date, it is automatically reformatted and displayed as a slash-delimited, four-digit-year date.

```
   // put the Informix-friendly format back into the field
   gField.value = monthDayFormat(mm) + "/" + monthDayFormat(dd) + "/" + yyyy;
   return true;
}
```

A utility function, invoked multiple times in the previous function, converts a single-digit month or day number to a string that might have a leading zero:

```
// convert month or day number to string,
// padding with leading zero if needed
function monthDayFormat(val)
{
   if (isNaN(val) || val == 0)
   {
      return "01";
   }
   else if (val < 10)
   {
      return "0" + val;
   }
   return "" + val;
}
```

LISTING 46-15

Functions to Check Month Lengths

```
// check the entered month for too high a value
function checkMonthLength(mm,dd)
{
   var months = new
Array("","January","February","March","April","May","June","July",
         "August","September","October","November","December");
   if ((mm == 4 || mm == 6 || mm == 9 || mm == 11) && dd > 30)
   {
      alert(months[mm] + " has only 30 days.");
      return false;
   }
   else if (dd > 31)
   {
      alert(months[mm] + " has only 31 days.");
      return false;
   }
   return true;
}

// check the entered February date for too high a value
function checkLeapMonth(mm,dd,yyyy)
{
   if (yyyy % 4 > 0 && dd > 28)
   {
      alert("February of " + yyyy + " has only 28 days.");
      return false;
   }
   else if (dd > 29)
   {
      alert("February of " + yyyy + " has only 29 days.");
      return false;
   }
   return true;
}
```

This is a rather extensive date-validation routine, but it demonstrates how thorough you must be when a database relies on accurate entries. The more prompting and assistance you can give to users to ferret out problems with invalid entries, the happier those users will be.

Cross-confirmation fields

The final validation type covered here is probably not a common request, but it demonstrates how the dispatch mechanism, created at the outset, expands so easily to accommodate this enhanced client request. The situation is that some fields (mostly dates in this application) are deemed critical pieces of data because this data triggers other processes from the database. As

a further check to ensure entry of accurate data, a number of values are set up for entry twice in separate fields — and the fields have to match exactly. In many ways, this mirrors the two passes you are often requested to make when you set a password: enter two copies and let the computer compare them to make sure you typed what you intended to type.

I established a system that places only one burden on the many programmers working on the forms: although you can name the primary field anything you want (to help alignment with database column names, for example), you must name the secondary field the same plus `"_xcfm"` — which stands for *cross-confirm*. Then, pass the `isConfirmed` validation name to the `validate()` function after the date range validation name, as follows:

```
onchange="parent.validate(window, this, 'isM5_P10Date','isConfirmed')"
```

In other words, after the entered value is initially checked against a required date range, the `isConfirmed()` validation function compares the fully vetted, properly formatted date in the current field against its parallel entry.

Listing 46-16 shows the one function in `validations.js` that handles the confirmation in both directions. After assigning a copy of the entry field value to the `inputStr` variable, the function next sets a Boolean flag (`primary`) that lets the rest of the script know if the entry field is the primary or secondary field. If the string `"_xcfm"` is missing from the field name, the entry field is the primary field.

For the primary field branch, the script assembles the name of the secondary field and compares the content of the secondary field's value against the `inputStr` value. If they are not the same, the user is entering a new value into the primary field, and the script empties the secondary field to force reentry to verify that the user enters the proper data.

For the secondary field entry branch, the script assembles a reference to the primary field by stripping away the final five characters of the secondary field's name. (I could use the `lastIndexOf()` string method, instead of the longer way involving the string's length; but after experiencing so many platform-specific problems with `lastIndexOf()` in earlier browsers, I decided to play it safe.) Finally, the two values are compared, with an appropriate alert displayed if they don't match.

LISTING 46-16

Cross-Confirmation Validation

```
// checks an entry against a parallel, duplicate entry to
// confirm that correct data has been entered
// Parallel field name must be the main field name plus "_xcfm"
function isConfirmed()
{
   var inputStr = gField.value;
   // flag for whether field under test is primary (true) or confirmation field
   var primary = (gField.name.indexOf("_xcfm") == -1);
      if (primary)
```

```
      {
         // clear the confirmation field if primary field is changed
         var xcfmField =
 window.frames[gFrame.name].document.forms[0].elements[gField.name + "_ xcfm"];
         var xcfmValue = xcfmField.value;
            if (inputStr != xcfmValue)
            {
               xcfmField.value = "";
               return true;
            }
      }
      else
      {
         var xcfmField =
 window.frames[gFrame.name].document.forms[0].elements[gField.name.substring(0,
(gField.name.length-5))];
         var xcfmValue = xcfmField.value;
            if (inputStr != xcfmValue)
            {
               alert("The main and confirmation entry field contents do not
                     match. Both fields must have EXACTLY the same content to
                     be accepted by the database.");
               gField.focus();
               gField.select();
               return false;
            }
      }
   return true;
}
```

Last-minute check

Every validation event handler is designed to return `true` if the validation succeeds. This comes in handy for the batch validation that performs one final check of the entries triggered by the form's `onsubmit` event handler. This event handler calls a `checkForm()` function and passes the form control object as a parameter. That parameter helps create a reference to the form element that is passed to each validation function.

Because successful validations return `true`, you can nest consecutive validation tests so that the most nested statement of the construction is `return true` after all validations have succeeded. The form's `onsubmit` event handler is as follows:

```
onsubmit="return checkForm(this)"
```

The following code fragment is an example of a `checkForm()` function. A separate `isDateFormat()` validation function, called here, checks whether the field contains an entry in the proper format — meaning that it has likely survived the range checking and format shifting of the real-time validation check.

```
function checkForm(form)
{
   if (parent.validate(window, form.birthdate, "isDateFormat"))
   {
      if (parent.validate(window, form.phone, "isPhone"))
      {
         if (parent.validate(window, form.name, "isNotEmpty"))
         {
            return true;
         }
      }
   }
   return false;
}
```

If any one validation fails, the field is given focus, and its content is selected (controlled by the individual validation function). In addition, the `checkForm()` function returns `false`. This, in turn, cancels the form submission.

Try it out

Listing 46-17 is a definition for a frameset that not only loads the validation routines described in this section, but also loads a page with a form that exercises the validations in real-time and batch mode, just prior to submission. The form appears earlier in this chapter in Figure 46.2.

LISTING 46-17

Frameset for Trying validation.js

```
<html>
   <head>
      <title>GiantCo Contractor Database</title>
      <script type="text/javascript" src="validation.js">
      </script>
      <script type="text/javascript">
         function blank()
         {
            return "<html><body bgcolor='lightsteelblue'><\/body><\/html>";
         }
      </script>
   </head>
   <frameset frameborder="" cols="20%,80%">
      <frame name="toc" src="javascript:parent.blank()" />
      <frame name="entries" src="jsb46-18.html" />
   </frameset>
</html>
```

The application scenario for the form is the entry of data into a company's contractor database. Some fields are required, and the date field must be cross-confirmed with a second entry of the same data. If the form passes its final validation prior to submission, the form reloads, and you see a readout of the form data that would have been submitted from the previous form, had the `action` been set to a server application URI.

Plan for Data Validation

We devoted this entire chapter to the subject of data validation because it represents the one area of error-checking that almost all JavaScript authors should be concerned with. If your scripts (client-side or server-side) perform processing on user entries, you want to prevent script errors at all costs.

CHAPTER

Scripting Java Applets and Plug-Ins

IN THIS CHAPTER

Communicating with Java applets from scripts

Accessing scripts and objects from Java applets

Controlling scriptable plug-ins

Netscape was the first to implement the facility enabling JavaScript scripts, Java applets, and plug-ins to communicate with each other under one technology umbrella, called the NPAPI (Netscape Plugin Application Programming Interface). Microsoft met the challenge and implemented a large part of that technology for WinIE4, but of course without using the Netscape-trademarked name for the technology. The name is a convenient way to refer to the capability, so you find it used throughout this chapter, applying to any browser that supports such facilities. This chapter focuses on the scripting side of the NPAPI: approaching applets and plug-ins from scripts and accessing scripts from Java applets.

Except for the part about talking to scripts from inside a Java applet, we don't assume you have any knowledge of Java programming. The primary goal here is to help you understand how to control applets and plug-ins (including ActiveX controls in WinIE) from your scripts. If you're in a position to develop specifications for applets, you learn what to ask of your Java programmers. But if you are also a Java applet programmer, you learn the necessary skills to get your applets in touch with HTML pages and scripts.

NPAPI Overview

Before you delve too deeply into the subject, you should be aware that NPAPI features are not available in Internet Explorer for Macintosh and early versions of Safari. Although unfortunate, this shouldn't prove to be an enormous hindrance as you contemplate a cross-browser application that utilizes the NPAPI. The bigger issue that emerges is between Internet Explorer and Mozilla, as the former prefers ActiveX controls, whereas the latter leans toward Java applets for similar functionality.

Whether you use the NPAPI to communicate with a Java applet or an ActiveX control, the advantage to you as an author is that the NPAPI extends the document object model to include objects and data types that are not a part of the HTML world. HTML, for example, does not have a form control element that displays real-time stock ticker data; nor does HTML have the capability to treat a sound file like anything more than a URL to be handed off to a helper application. With the NPAPI, however, your scripts can treat the applet that displays the stock ticker as an object whose properties and methods can be modified after the applet loads; scripts can also tell an audio clip when to play or pause by controlling the plug-in that manages the incoming sound file.

Why Control Java Applets?

A question we often hear from experienced Java programmers is, "Why bother controlling an applet via a script when you can build all the interactivity you want into the applet itself?" This question is valid if you come from the Java world, but it takes a viewpoint from the HTML and scripting world to fully answer it.

Java applets exist in their own private rectangles, remaining largely oblivious to the HTML surroundings on the page. Applet designers who don't have extensive web page experience tend to regard their applets as the center of the universe rather than as components of HTML pages.

As a scripter, on the other hand, you may want to use those applets as powerful components to spiff up the overall presentation. Using applets as prewritten objects enables you to make simple changes to the HTML pages — including the geographic layout of elements and images — at the last minute, without having to rewrite and recompile Java program code. If you want to update the look with an entirely new graphical navigation or control bar, you can do it directly via HTML and scripting.

When it comes to designing or selecting applets for inclusion into our scripted page, we prefer using applet interfaces that confine themselves to data display, putting any control of the data into the hands of the script, rather than using onscreen buttons in the applet rectangle. We believe this setup enables much greater last-minute flexibility in the page design — not to mention consistency with HTML form element interfaces — than putting everything inside the applet rectangle.

A Little Java

If you plan to look at a Java applet's scripted capabilities, you can't escape the need to know a little about Java applets and some terminology. The discussion goes more deeply into object orientation than you have seen with JavaScript, but we'll try to be gentle.

Java building blocks classes

One part of Java that closely resembles JavaScript is that Java programming deals with objects, much the way JavaScript deals with a page's objects. Java objects, however, are not the familiar HTML objects, but rather more basic building blocks, such as tools that draw to the screen and data streams. But both languages also have some non-HTML kinds of objects in common: strings, arrays, numbers, and so on.

While not technically correct, we can say that every Java object is known as a *class* — a term from the object-oriented world. When you use a Java compiler to generate an applet from its source code, a `.class` file is generated, which happens to incorporate many Java classes, such as strings, image areas, font objects, and the like. This file is the one you specify for the `code` attribute of the `<object>` tag (the `<applet>` tag was used in older browsers but is now deprecated).

Java methods

Most JavaScript objects have methods attached to them that define what actions the objects are capable of performing. A string object, for instance, has the `toUpperCase()` method that converts the string to all uppercase letters. Java classes also have methods. Many methods are predefined in the base Java classes embedded inside the virtual machine. But inside a Java applet, the author can write methods that either override the base method or are used exclusively in the applet's class. These methods are, in a way, like the functions you write in JavaScript for a page.

Not all methods, however, are created the same. Java lets authors determine how visible a method is to outsiders. The types of methods that you, as a scripter, are interested in are the ones declared as public methods. You can access such methods from JavaScript via a syntax that falls very much in line with what you already know. For example, a common public method in applets stops an applet's main process. Such a Java method may look like this:

```
@Override
public void stop() {
   if(thread != null) {
      thread = null;
   }
}
```

The `void` keyword simply means that this method does not return any values (compilers need to know this stuff). Assuming that you have one applet loaded in your page, the JavaScript call to this applet method is

```
document.applets[0].stop();
```

Listing 47-1 shows how all this works with the `<object>` tag for a scriptable digital clock applet example. The script includes calls to two of the applet's methods: to stop and to start the clock.

The syntax for accessing the method (in the two functions) is just like JavaScript in that the references to the applet's methods include the applet object (`clock1` in the example), which is contained by the `document` object.

LISTING 47-1

Stopping and Starting an Applet (XHTML)

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>A Script That Could Stop a Clock</title>
      <script type="text/javascript">
         function pauseClock()
         {
            // get ref to second object for non-WinIE
            var applet = (document.clock1.length) ? document.clock1[1] :
                           document.clock1;
            applet.stop();
         }
         function restartClock()
         {
            var applet = (document.clock1.length) ? document.clock1[1] :
                           document.clock1;
            applet.start();
         }
      </script>
   </head>
   <body>
      <h1>Simple control over an applet</h1>
      <hr />
      <object name="clock1" classid="clsid:8AD9C840-044E-11D1-B3E9-00805F499D93"
              codebase=
      "http://java.sun.com/update/1.5.0/jinstall-1_5_0-windows-i586.cab"
              width="500" height="45">
         <param name="code" value="ScriptableClock.class" />
         <param name="bgColor" value="Green" />
         <param name="fgColor" value="Blue" />
         <!--[if !IE]> Non-WinIE Browsers -->
            <object name="clock1" classid="java:ScriptableClock.class"
                    type="application/x-java-applet"
                    width="500" height="45">
               <param name="bgColor" value="Green" />
               <param name="fgColor" value="Blue" />
            </object>
         <!-- <![endif]-->
      </object>

      <form name="widgets1">
         <input type="button" value="Pause Clock" onclick="pauseClock()" />
         <input type="button" value="Restart Clock" onclick="restartClock()" />
```

continued

LISTING 47-1 *(continued)*

```
      </form>
   </body>
</html>
```

Note

The property assignment event handling technique employed throughout the code in this chapter and much of the book is a deliberate simplification to make the code more readable. It is generally better to use the more modern approach of binding events using the `addEventListener()` **(NN6+/Moz/W3C) or** `attachEvent()` **(IE5+) methods. A modern cross-browser event handling technique is explained in detail in Chapter 32, "Event Objects."** ■

Even though the `<object>` tag is supported by all modern browsers, its specific usage across browsers isn't exactly standard. It's necessary to work a little markup magic to use modern `<object>` tags that operate in IE, Mozilla-based, and WebKit-based browsers. The difficulty arises in the way the browsers specify some attribute or parameter values for Java applets. Listing 47-1 reveals how to embed an applet into a document with the `<object>` tag that works simultaneously in the different browsers.

An important feature of the listing is a Microsoft proprietary markup feature called *conditional comments* (`http://msdn.microsoft.com/en-us/library/ms537512%28VS.85%29.aspx`). These HTML comment tags, with their special comment text, allow IE to skip over HTML markup (although if the Microsoft conditional comments are not used, IE will still render correctly; this is just a cleaner, best practices, approach). In the listing, only the first `<object>` tag is rendered in WinIE; other browsers load both, but do not load the applet in the first tag because the attribute and parameter values are not in the format that they require for loading a Java applet. In addition to the tag markup differences, note that the functions controlling the applet create references to the applet object differently. Browsers other than IE have two `object` elements with the same name to deal with, meaning that there is an array of objects with that name; thus the script pulls a reference to the second one when two are detected.

Java applet "properties"

The Java equivalents of JavaScript object properties can be called properties in Java as well, but in reality are public accessor and mutator methods, commonly called *getters* and *setters*. These methods allow for reading and writing the corresponding private attribute of the class. If you have access to some Java source code, you can recognize these getters and setters by their names:

```
private String fontName;
...
public String getFontName() {
   return fontName;
}

public void setFontName(String fontName) {
   this.fontName = fontName;
}
```

In the preceding code, you'll notice that the private attribute `fontName` is associated with its corresponding getters and setters via a standard JavaBean naming convention: camel case. As a JavaScript programmer, you would be using these methods to indirectly access the corresponding private attribute. Getters and setters are used because the use of public instance attributes is considered a violation of object encapsulation and is seriously frowned upon by the Java programmer.

Java authors must specify a method's data type when declaring any method that returns a value. That's why the `String` data type appears in the preceding example.

Your scripts can access these variables with the same kind of syntax that you use to access JavaScript object properties. In order to access the `fontName` field, you would execute its corresponding getter and setter, as shown in the following example:

```
var theFont = document.applets[0].getFontName();
document.applets[0].setFontName("courier");
```

Scripting Applets in Real Life

Because the purpose of scripting an applet is to gain access to the inner sanctum of a compiled program, the program should be designed to handle such rummaging around by scripters. If you can't acquire a copy of the source code or don't have any other documentation about the scriptable parts of the applet, you may have a difficult time knowing what to script and how to do it.

Getting to scriptable methods

If you write your own applets or are fortunate enough to have the source code for an existing applet, the safest way to modify the applet variable settings or the running processes is through applet methods. An applet designed for scriptability should have a number of methods defined that enable you to make scripted changes to private variable values.

To view a sample of an applet designed for scriptability, open the Java source code file for Listing 47-2 from the CD-ROM. A portion of that program listing is shown in the following example.

LISTING 47-2

Partial Listing for ScriptableClock.java

```
import java.awt.*;
import java.util.*;
import java.text.*;

public class ScriptableClock extends java.applet.Applet implements Runnable {
```

continued

LISTING 47-2 *(continued)*

```
// control variables
private final boolean GMT = true;
private final boolean LOCALE = false;
private String displayDate;
private Font displayFont;
private Thread thread;
// parameter options
private Color bgColor;
private Color fgColor;
private Rectangle displayArea;
private String fontName;
private int fontSize;
private int fontStyle;
private int height, width;
private boolean timeZone;

@Override
public void init() {
   parseArgs();
   displayFont = new Font(fontName, fontStyle, fontSize);
   timeZone = LOCALE;
   displayArea = getBounds();
   height = displayArea.height;
   width = displayArea.width;
   resize(width, height);
}

@Override
public void start() {
   if (thread == null) {
      thread = new Thread(this);
      thread.start();
   }
}

@Override
public void stop() {
   if (thread != null) {
      thread = null;
   }
}

public void run() {
   while (thread != null) {
      Date theDate = new Date();
      DateFormat dateFormat = DateFormat.getDateTimeInstance(DateFormat.
                              MEDIUM, DateFormat.FULL);
      if (timeZone) {
         dateFormat.setTimeZone(TimeZone.getTimeZone("GMT-0:00"));
      }
```

```
            displayDate = dateFormat.format(theDate);
            repaint();
            try {
               Thread.sleep(1000);
            } catch (InterruptedException e) {
            }
         }
      }

      @Override
      public void paint(Graphics g) {
         g.setColor(bgColor);
         g.fillRect(0, 0, width, height);

         g.setColor(fgColor);
         g.setFont(displayFont);

         FontMetrics fm = getFontMetrics(getFont());
         int textWidth = fm.stringWidth(displayDate);
         int horizOffset = (width / 2) - (textWidth / 2);
         g.drawString(displayDate, 5, 35);
      }

      /*
      Begin public methods for getting
      and setting data via LiveConnect
       */
      public void setTimeZone(String zone) {
         stop();
         timeZone = (zone.startsWith("GMT")) ? true : false;
         start();
      }

      public void setFont(String newFont, String newStyle, String newSize) {
         stop();
         if (newFont != null && newFont.length() != 0) {
            fontName = newFont;
         }
         if (newStyle != null && newStyle.length() != 0) {
            setFontStyle(newStyle);
         }
         if (newSize != null && newSize.length() != 0) {
            setFontSize(newSize);
         }
         displayFont = new Font(fontName, fontStyle, fontSize);
         start();
      }
      public void setColor(String newbgColor, String newfgColor) {
         stop();
         bgColor = parseColor(newbgColor);
         fgColor = parseColor(newfgColor);
         start();
      }
```

continued

LISTING 47-2 *(continued)*

```
public String getInfo() {
   String result = "Info about ScriptableClock.class\r\n";
   result += "Version/Date: 1.0d1/2 May 1996\r\n";
   result += "Author: Danny Goodman (dannyg@dannyg.com)\r\n";
   result += "Public Variables:\r\n";
   result += "   (None)\r\n\r\n";
   result += "Public Methods:\r\n";
   result += "   setTimeZone(\"GMT\" | \"Locale\")\r\n";
   result += "   setFont(\"fontName\",\"Plain\" |\"Bold\" | \"Italic\",
                        \"fontSize\")\r\n";
   result += "   setColor(\"bgColorName\", \"fgColorName\")\r\n";
   result += "       colors: Black, White, Red, Green, Blue, Yellow\r\n";
   return result;
}
/*
End public methods for scripted access.
 */

private void setFontStyle(String style) {
   try {
      if (style.equalsIgnoreCase("Plain")) {
         fontStyle = Font.PLAIN;
      } else if (style.equalsIgnoreCase("Italic")) {
         fontStyle = Font.ITALIC;
      } else {
         fontStyle = Font.BOLD;
      }
   } catch (Exception e) {
      fontStyle = Font.BOLD;
   }
}

private void setFontSize(String size) {
   try {
      fontSize = Integer.parseInt(size);
   } catch (Exception e) {
      fontSize = 24;
   }
}

private void parseArgs() {
   fontName = getParameter("font");
   if (fontName == null) {
      fontName = new String("TimesRoman");
   }

   String n = getParameter("fontSize");
   setFontSize(n);
```

```
        n = getParameter("fontStyle");
        setFontStyle(n);

        bgColor = parseColor(getParameter("bgColor"));

        if (bgColor == null) {
            bgColor = Color.white;
        }

        fgColor = parseColor(getParameter("fgColor"));
        if (fgColor == null) {
            fgColor = Color.black;
        }
    }

    private Color parseColor(String c) {
        try {
            if (c.equalsIgnoreCase("Black")) {
                return Color.black;
            } else if (c.equalsIgnoreCase("White")) {
                return Color.white;
            } else if (c.equalsIgnoreCase("Red")) {
                return Color.red;
            } else if (c.equalsIgnoreCase("Green")) {
                return Color.green;
            } else if (c.equalsIgnoreCase("Blue")) {
                return Color.blue;
            } else if (c.equalsIgnoreCase("Yellow")) {
                return Color.yellow;
            } else {
                return Color.black;
            }
        } catch (Exception e) {
            return null;
        }
    }
}
```

The methods shown in Listing 47-2 are defined specifically for scripted access. In this case, they safely stop the applet thread before changing any values. The last method is one we recommend to applet authors. The method returns a small bit of documentation containing information about the kind of methods that the applet likes to have scripted and what you can have as the passed parameter values.

Now that you see the amount of scriptable information in this applet, look at Listing 47-3, which takes advantage of that scriptability by providing several HTML form elements as user controls for the clock. The results are shown in Figure 47-1.

LISTING 47-3

A More Fully Scripted Clock

```
<!DOCTYPE html>
<html>
   <head>
     <meta http-equiv="content-type" content="text/html;charset=utf-8">
     <title>Clock with Lots o' Widgets</title>
     <script type="text/javascript">
       function setTimeZone(popup)
       {
          var choice = popup.options[popup.selectedIndex].value;
          var applet = (document.clock2.length) ? document.clock2[1] :
                        document.clock2;
          applet.setTimeZone(choice);
       }

       function setColor(form)
       {
          var bg =
form.backgroundColor.options[form.backgroundColor.selectedIndex].value;
          var fg =
form.foregroundColor.options[form.foregroundColor.selectedIndex].value;
          var applet = (document.clock2.length) ? document.clock2[1] :
document.clock2;
          applet.setColor(bg, fg);
       }

       function setFont(form)
       {
          var fontName = form.theFont.options[form.theFont.selectedIndex].value;
          var fontStyle =
 form.theStyle.options[form.theStyle.selectedIndex].value;
          var fontSize = form.theSize.options[form.theSize.selectedIndex].value;
          var applet = (document.clock2.length) ? document.clock2[1] :
 document.clock2;
          applet.setFont(fontName, fontStyle, fontSize);
       }
       function getAppletInfo(form)
       {
          var applet = (document.clock2.length) ? document.clock2[1] :
document.clock2;
          form.details.value = applet.getInfo();
       }

       function showSource()
       {
          var newWindow = window.open("ScriptableClock.java","",
"WIDTH=450,HEIGHT=300,RESIZABLE,SCROLLBARS");
       }
```

```
 </script>
</head>
<body>
   <object name="clock2" classid="clsid:8AD9C840-044E-11D1-B3E9-00805F499D93"
           codebase=
   "http://java.sun.com/update/1.5.0/jinstall-1_5_0-windows-i586.cab"
           width="500" height="45">
    <param name="code" value="ScriptableClock.class" />
    <param name="bgColor" value="Black" />
    <param name="fgColor" value="Red" />
    <!--[if !IE]> Non-WinIE Browsers -->
       <object name="clock2" classid="java:ScriptableClock.class"
               type="application/x-java-applet"
               width="500" height="45">
          <param name="bgColor" value="Black" />
          <param name="fgColor" value="Red" />
       </object>
    <!-- <![endif]-->
 </object>
 <form name="widgets2">
    Select Time Zone:
    <select name="zone" onchange="setTimeZone(this)">
       <option selected="selected" value="Locale">Local Time</option>
       <option value="GMT">Greenwich Mean Time</option>
    </select>
    <p>
       Select Background Color:
       <select name="backgroundColor" onchange="setColor(this.form)">
          <option value="White">White</option>
          <option selected="selected" value="Black">Black</option>
          <option value="Red">Red</option>
          <option value="Green">Green</option>
          <option value="Blue">Blue</option>
          <option value="Yellow">Yellow</option>
       </select>
       Select Color Text Color:
       <select name="foregroundColor" onchange="setColor(this.form)">
          <option value="White">White</option>
          <option value="Black">Black</option>
          <option selected="selected" value="Red">Red</option>
          <option value="Green">Green</option>
          <option value="Blue">Blue</option>
          <option value="Yellow">Yellow</option>
       </select>
    </p>
    <p>
       Select Font:
       <select name="theFont" onchange="setFont(this.form)">
          <option selected="selected" value="TimesRoman">Times Roman</option>
          <option value="Helvetica">Helvetica</option>
          <option value="Courier">Courier</option>
          <option value="Arial">Arial</option>
       </select>
```

continued

LISTING 47-3 *(continued)*

```
            <br />
            Select Font Style:
            <select name="theStyle" onchange="setFont(this.form)">
               <option selected="selected" value="Plain">Plain</option>
               <option value="Bold">Bold</option>
               <option value="Italic">Italic</option>
            </select>
            <br />
            Select Font Size:
            <select name="theSize" onchange="setFont(this.form)">
               <option value="12">12</option>
               <option value="18">18</option>
               <option selected="selected" value="24">24</option>
               <option value="30">30</option>
            </select>
         </p>
         <hr />
         <input type="button" name="getInfo" value="Applet Info"
                onclick="getAppletInfo(this.form)" />
         <p>
            <textarea name="details" rows="11" cols="70">
            </textarea>
         </p>
      </form>
      <hr />
   </body>
</html>
```

Very little of the code here controls the applet — only the handful of functions near the top. The rest of the code makes up the HTML user interface for the form element controls. After you open this document from the CD-ROM, be sure to click the Applet Info button to see the methods that you can script and the way that the parameter values from the JavaScript side match up with the parameters on the Java method side.

Applet limitations

Because of concerns about security breaches via the NPAPI, Mozilla clamps down on some powers that would be nice to have via a scripted applet. The most noticeable barrier is the one that prevents applets from accessing the network under scripted control. Therefore, even though a Java applet has no difficulty reading or writing text files from the server, such capabilities — even if built into an applet of your own design — won't be carried out if triggered by a JavaScript call to the applet. However, signed scripts (see Chapter 49) and applets can break through these security barriers after the user has given explicit permission for them to do so.

FIGURE 47-1

Scripting more of the ScriptableClock applet.

Faceless applets

Until the NPAPI came along, Java applets were generally written to show off data and graphics — to play a big role in the presentation on the page. But if you prefer to let an applet do the heavy algorithmic lifting for your pages, while the HTML form elements and images (or Dynamic HTML facilities) do the user interface, you essentially need what we call a *faceless applet*.

The method for embedding a faceless applet into your page is the same as embedding any applet: Use the `<object>` tag. For a faceless applet, specify only 1 pixel for both the `height` and `width` attributes (0 has strange side effects). This setting creates a dot on the screen, which, depending on your page's background color, may be completely invisible to page visitors. Place it at the bottom of the page, if you like.

To show how nicely this method can work, Listing 47-4 provides the Java source code for a simple applet that retrieves a specific text file, and stores the results in a Java variable available for fetching by the JavaScript shown in Listing 47-5. The HTML even automates the loading process by triggering the retrieval of the Java applet's data from an `onload` event handler.

LISTING 47-4

Java Applet Source Code

```
import java.net.*;
import java.io.*;

public class FileReader extends java.applet.Applet implements Runnable {

   private Thread thread;
   private URL url;
   private String output;
   private String fileName = "Bill of rights.txt";

   public void getFile(String fileName) throws IOException {
      String result = null;
      String line = null;
      InputStream connection = null;
      DataInputStream dataStream = null;
      StringBuffer buffer = new StringBuffer();

      try {
         url = new URL(getDocumentBase(), fileName);
         connection = url.openStream();
         dataStream = new DataInputStream(new BufferedInputStream(connection));

         while ((line = dataStream.readLine()) != null) {
            buffer.append(line + "\n");
         }
         result = buffer.toString();
      } catch (MalformedURLException e) {
         result = "AppletError " + e;
      } catch (IOException e) {
         result = "AppletError: " + e;
      } finally {
         if (dataStream != null) {
            dataStream.close();
         }
         if (connection != null) {
            connection.close();
         }
      }
      output = result;
   }

   public String fetchText() {
      return output;
   }

   @Override
   public void init() {
   }
```

```
    @Override
    public void start() {
       if (thread == null) {
          thread = new Thread(this);
          thread.start();
       }
    }

    @Override
    public void stop() {
       if (thread != null) {
          thread = null;
       }
    }

    public void run() {
       try {
          getFile(fileName);
       } catch (IOException e) {
          output = "AppletError: " + e;
       }
    }
}
```

All the work of actually retrieving the file is performed in the `getFile()` method (which runs immediately after the applet loads). Notice that the name of the file to be retrieved, `Bill of Rights.txt`, is stored as a variable near the top of the code, making it easy to change for a recompilation, if necessary. You can also modify the applet to accept the filename as an applet parameter, specified in the HTML code. Meanwhile, the only hook that JavaScript needs is the one public method called `fetchText()`, which merely returns the value of the output variable, which in turn holds the file's contents.

This Java source code must be compiled into a Java class file (already compiled and included on the CD-ROM as `FileReader.class`) and placed in the same directory as the HTML file that loads this applet. Also, no explicit pathname for the text file is supplied in the source code, so the text file is assumed to be in the same directory as the applet.

LISTING 47-5

HTML Asking Applet to Read Text File

```
<!DOCTYPE html>
<html>
   <head>
     <meta http-equiv="content-type" content="text/html;charset=utf-8">
     <title>Letting an Applet Do The Work</title>
     <script type="text/javascript">
       function getFile(form)
```

continued

LISTING 47-5 *(continued)*

```
        {
         var applet = (document.readerApplet.length) ?
                        document.readerApplet[1] : document.readerApplet;
         var output = applet.fetchText();
           form.fileOutput.value = output;
        }
        function autoFetch()
        {
         var applet = (document.readerApplet.length) ? document.readerApplet[1] :
                        document.readerApplet;
         var output = applet.fetchText();
              if (output != null)
              {
                 document.forms[0].fileOutput.value = output;
                 return;
              }
         var t = setTimeout("autoFetch()",1000);
        }
       </script>
    </head>
    <body onload="autoFetch()">
       <h1>Text from a text file...</h1>
       <form name="reader">
          <input type="button" value="Get File" onclick="getFile(this.form)" />
          <p>
             <textarea name="fileOutput" rows="10" cols="60" wrap="hard">
             </textarea>
          </p>
          <p>
             <input type="Reset" value="Clear" />
          </p>
       </form>
       <object name="readerApplet"
 classid="clsid:8AD9C840-044E-11D1-B3E9-00805F499D93"
              codebase=
       "http://java.sun.com/update/1.5.0/jinstall-1_5_0-windows-i586.cab"
              width="1" height="1">
          <param name="code" value="FileReader.class" />
          <!--[if !IE]> Non-WinIE Browsers -->
             <object name="readerApplet" classid="java:FileReader.class"
                     type="application/x-java-applet"
                     width="1" height="1">
             </object>
          <!-- <![endif]-->
       </object>
    </body>
 </html>
```

Because an applet is usually the last detail to finish loading in a document, you can't use an applet to generate the page immediately. At best, an HTML document can display a pleasant welcome screen while the applet finishes loading and running whatever it does to prepare data for the page's form elements. Fortunately for modern browsers, the page can then be dynamically constructed out of the retrieved data. Notice in Listing 47-5 that the `onload` event handler calls a function that checks whether the applet has supplied the requested data. If not, the same function is called repeatedly in a timer loop, until the data is ready and the textarea can be set. The `<object>` tag is located at the bottom of the Body, set to 1 pixel square — invisible to the user. No user interface exists for this applet, so you have no need to clutter up the page with any placeholders or bumper stickers.

Figure 47-2 shows the page generated by the HTML and the applet working together. The Get File button is merely a manual demonstration of calling the same applet method that the `onload` event handler calls.

FIGURE 47-2

The page with text retrieved from a server file.

A faceless applet may be one way for web authors to hide what may otherwise be JavaScript code that is open to any visitor's view. For example, if you want to deliver a small data collection lookup with a document, but don't want the array of data to be visible in the JavaScript code, you can create the array and lookup functionality inside a faceless applet. Then, use form controls and JavaScript to act as query entry and output display devices (or to dynamically generate a table). Because the parameter values passed between JavaScript and Java applets must be string, numeric, or Boolean values, you won't be able to pass arrays without performing some amount

of conversion within either the applet or the JavaScript code (JavaScript's `string.split()` and `array.join()` methods help a great deal here).

Data type conversions

The example in this chapter does not pass any parameters to the applet's methods, but you are free to do so. You need to pay attention to the way in which values are converted to Java data types. JavaScript strings and Boolean values are converted to Java String and Boolean objects. All JavaScript numbers, regardless of their subtype (that is, integer or floating-point number), are converted to `Float` objects. Therefore, if a method must accept a numeric parameter from a script, the parameter variable in the Java method must be defined as a `Float` type.

The distinction between JavaScript string values and string objects can impact data being passed to an applet. If an applet method requires a string object as a parameter, you may have to explicitly convert a JavaScript string value (for example, a string from a text field) to a string object via the `new String()` constructor (see Chapter 15).

You can also pass references to objects, such as form control elements. Such objects get wrapped with a `JSObject` type (see the discussion about this class later in the chapter). Therefore, parameter variables must be established as type `JSObject` (and the `netscape.javascript.JSObject` class must be imported into the applet).

Applet-to-Script Communication

The flip side of scripted applet control is having an applet control both script and HTML content in the page. Before you undertake this avenue in page design, you must bear in mind that any calls made from the applet to the page are hard-wired for the specific scripts and HTML elements in the page. If this level of tight integration and dependence suits the application, the link up will be successful.

Note

The discussion of applet-to-script communication assumes you have experience writing Java applets. We use Java jargon quite freely in this discussion. ■

What your applet needs

Most modern browsers come with a zipped set of special class files tailored for use in the NPAPI. For Mozilla, the class files are located in an archive called `-plugin.jar` (Windows) or `JavaPluginCocoa.bundle` (Mac). Use the file search facility of the OS to locate the relevant file on your system. Microsoft versions of these class files are usually located in `C:\Program Files\Java\jre6\lib`. If you don't find `jre6`, look for `jre5`. The browser must see these class files (and have both Java and JavaScript enabled in the preferences screens) for the NPAPI to work.

You then need to add the `plugin.jar` file to the Java `classpath` environment variable. Following is an example of how this is accomplished at the command line:

```
set CLASSPATH= C:\jdk1.6.0\jre\lib\plugin.jar
```

Of course, your specific Java SDK installation may be different in terms of version numbering, but the command should be very similar. With the `plugin.jar` file available in the `classpath`, you're ready to use the NPAPI objects in Java code, and build an applet.

Following are the two vital classes in the `netscape` package (yes, even for IE), which is the Java package made available in the NPAPI ZIP file (`-plugin.jar`):

```
netscape.javascript.JSObject
netscape.javascript.JSException
```

Both classes must be imported into your applet via the Java `import` compiler directive:

```
import netscape.javascript.*;
```

When the applet runs, the NPAPI-aware browser knows how to find the two classes, so that the user doesn't have to do anything special as long as the supporting files are in their default locations.

What your HTML needs

As a security precaution, an `<object>` tag requires one extra parameter to give the applet permission to access the HTML and scripting inside the document. That parameter is `mayscript`, and it can go anywhere inside the `<object>` tag, as follows:

```
<param name="mayscript" value="true" />
```

Permission is not required for JavaScript to access an applet's methods or properties, but if the applet initiates contact with the page, this attribute is required.

About the JSObject class

The portal between the applet and the HTML page that contains it is the `netscape.javascript.JSObject` class. This object's methods allow the applet to contact document objects and invoke JavaScript statements. Table 47-1 shows the object's methods and one static method.

Just as the `window` object is the top of the document object hierarchy for JavaScript references, the `window` object is the gateway between the applet code and the scripts and document objects. To open that gateway, use the `JSObject.getWindow()` method to retrieve a reference to the document window. Assign that object to a variable that you can use throughout your applet code. The following code fragment shows the start of an applet that assigns the window reference to a variable named `mainwin`:

```
import netscape.javascript.*;

public class myClass extends java.applet.Applet {
   private JSObject mainwin;

   @Override
```

```
      public void init() {
         mainwin = JSObject.getWindow(this);
      }
   }
```

TABLE 47-1

JSObject Class Methods

Method	Description
`call(String functionName, arrayObject args[])`	Invokes JavaScript function; argument(s) passed as an array
`eval(String expression)`	Invokes a JavaScript statement
`getMember(String elementName)`	Retrieves a named object belonging to a container
`getSlot(Int index)`	Retrieves indexed object belonging to a container
`getWindow(Applet applet)`	Static method retrieves applet's containing window
`removeMember(String elementName)`	Removes a named object belonging to a container
`setMember(String elementName, Object value)`	Sets value of a named object belonging to a container
`setSlot(int index, Object value)`	Sets value of an indexed object belonging to a container
`toString()`	Returns string version of JSObject

If your applet will be making frequent trips to a particular object, you may want to create a variable holding a reference to that object. To accomplish this, the applet needs to make progressively deeper calls into the document object hierarchy with the `getMember()` method. For example, the following sequence assumes `mainwin` is a reference to the applet's document window. Eventually, the statements set a form's field object to a variable for use elsewhere in the applet:

```
JSObject doc = (JSObject) mainwin.getMember("document");
JSObject form = (JSObject) doc.getMember("entryForm");
JSObject phonefld = (JSObject) form.getMember("phone");
```

Another option is to use the Java `eval()` method to execute an expression from the point of view of any object. For example, the following statement gets the same field object as the preceding fragment:

```
JSObject phonefld = mainwin.eval("document.entryForm.phone");
```

As soon as you have a reference to an object, you can access its properties via the `getMember()` method, as shown in the following example, which reads the `value` property of the text box, and casts the value into a Java `String` object:

```
String phoneNum = (String) phonefld.getMember("value");
```

Two `JSObject` class methods let your applet execute arbitrary JavaScript expressions, and invoke object methods: the `eval()` and `call()` methods. Use these methods with any `JSObject`. If a value is to be returned from the executed statement, you must cast the result into the desired object type. The parameter for the `eval()` method is a string of the expression to be evaluated by JavaScript. Scope of the expression depends on the object attached to the `eval()` method. If you use the `window` object, the expression would exist as if it were a statement in the document script (not defined inside a function).

Using the `call()` method is convenient for invoking JavaScript functions in the document, although it requires a little more preparation. The first parameter is a string of the function name. The second parameter is an array of arguments for the function. Parameters can be of mixed data types, in which case the array would be of type `Object`. If you don't need to pass a parameter to the function call, you can define an array of a single empty string value (for example, `String arg[] = {""}`) and pass that array as the second parameter.

Data type conversions

The strongly-typed Java language is a mismatch for loosely-typed JavaScript. As a result, with the exception of Boolean and string objects (which are converted to their respective JavaScript objects), you should be aware of the way the NPAPI adapts data types to JavaScript.

Any Java object that contains numeric data is converted to a JavaScript number value. Because JavaScript numbers are `IEEE` doubles, they can accommodate just about everything Java can throw their way.

If the applet extracts an object from the document, and then passes that `JSObject` type back to JavaScript, that passed object is converted to its original JavaScript object type. But objects of other classes are passed as their native objects wrapped in JavaScript "clothing." JavaScript can access the applet object's methods as if the object were a JavaScript object. Finally, Java arrays are converted to the same kind of JavaScript array created via the `new Array()` constructor. Elements can be accessed by integer index values (not named index values). All other JavaScript array properties and methods apply to this object as well.

Example applet-to-script application

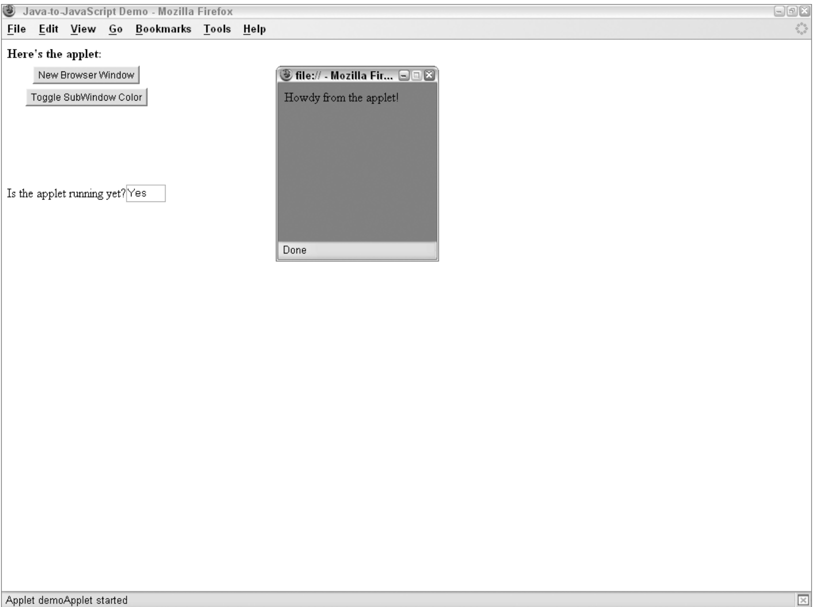
To demonstrate several techniques for communicating from an applet to both JavaScript scripts and document objects, we present an applet that displays two simple buttons (see Figure 47-3). One button generates a new window, spawned from the main window, and fills the window with dynamically-generated content from the applet.

Note

Be sure to turn off pop-up window blocking temporarily to run this example. WebKit-based browsers do not handle pop-up windows in the same way as IE, Opera, and Mozilla-based browsers, so you will have problems when running this example. ■

FIGURE 47-3

The applet displays two buttons seamlessly on the page.



The second button communicates from the applet to that second window by invoking a JavaScript function in the document. The last part of the demonstration shows the applet changing the value of a text box when the applet starts up.

Listing 47-6 shows the source code for the Java applet. Because the applet generates two buttons, the code begins by importing the `AWT` interface builder classes. We also import the `netscape.javascript` package to get the `JSObject` class. The name of this sample class is `JtoJSDemo`. We declare four global variables: two for the windows, two for the applet button objects.

LISTING 47-6

Java Applet Source Code

```
import java.awt.*;
import java.awt.event.*;
import netscape.javascript.*;

public class JtoJSDemo extends java.applet.Applet implements ActionListener {
   private JSObject mainwin, subwin;
   private Button newWinButton, toggleButton;
```

The applet's `init()` method establishes the user interface elements for this simple applet. A white background is matched in the HTML with a white document background color, making the applet appear to blend in with the page. We use this opportunity to set the `mainwin` variable to the browser window that contains the applet.

```
@Override
public void init() {
   setBackground(Color.white);
   newWinButton = new Button("New Browser Window");
   toggleButton = new Button("Toggle SubWindow Color");
   this.add(newWinButton);
   this.add(toggleButton);
   newWinButton.addActionListener(this);
   toggleButton.addActionListener(this);
   mainwin = JSObject.getWindow(this);
}
```

As soon as the applet starts, it changes the `value` property of a text box in the HTML form. Because this is a one-time access to the field, we elected to use the `eval()` method from the point of view of the main window, rather than build successive object references through the object hierarchy with the `getMember()` method.

```
@Override
public void start() {
   mainwin.eval("document.indicator.running.value = 'Yes'");
}
```

Event handling is quite simple in this application. A click of the first button invokes `doNewWindow()`; a click of the second invokes `toggleColor()`. Both methods are defined later in the applet.

```
public void actionPerformed(ActionEvent evt) {
   Button source = (Button)evt.getSource();
   if (source == newWinButton) {
      doNewWindow();
   } else if (source == toggleButton) {
      toggleColor();
   }
}
```

One of the applet's buttons calls the `doNewWindow()` method defined here. We use the `eval()` method to invoke the JavaScript `window.open()` method. The string parameter of the `eval()` method is exactly like the statement that appears in the page's JavaScript to open a new window. The `window.open()` method returns a reference to that subwindow, so that the statement here captures the returned value, casting it as a `JSObject` type for the `subwin` variable. That `subwin` variable can then be used as a reference for another `eval()` method that writes to that second window. Notice that the object to the left of the `eval()` method governs the recipient of the `eval()` method's expression. The same is true for closing the writing stream to the subwindow.

Note

Unfortunately, earlier IE implementations of `JSObject` do not provide a suitable reference to the external window after it is created. Therefore, the window neither receives its content nor responds to color changes in this example. Due to other anomalies with subwindows, we advise against using NPAPI powers with multiple windows in older versions of IE. ■

LISTING 47-6 *(continued)*

Java Applet Source Code

```
   void doNewWindow() {
      subwin = (JSObject) mainwin.eval(
         "window.open('','fromApplet','height=200,width=200')");
      subwin.eval("document.write('<html><body bgcolor=white>Howdy from the
         applet!</body></html>')");
      subwin.eval("document.close()");
   }
```

The second button in the applet calls the `toggleColor()` method. In the HTML document, a JavaScript function named `toggleSubWindowColor()` takes a `window` object reference as an argument. Therefore, we first assemble a one-element array of type `JSObject`, consisting of the `subwin` object. That array is the second parameter of the `call()` method, following a string version of the JavaScript function name being called.

```
         void toggleColor() {
            if (subwin != null) {
               JSObject arg[] = {subwin};
               mainwin.call("toggleSubWindowColor", arg);
            }
         }
      }
```

Now onto the HTML that loads the preceding applet class and receives its calls. The document is shown in Listing 47-7. One function is called by the applet. A text box in the form is initially set to "No" but gets changed to "Yes" by the applet after it has finished its initialization. The only other item of note is that the `<object>` tag includes a `mayscript` parameter to allow the applet to communicate with the page.

LISTING 47-7

HTML Document Called by Applet

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Java-to-JavaScript Demo</title>
```

```
      <script type="text/javascript">
         function toggleSubWindowColor(wind)
         {
           if (wind.closed)
           {
              alert("The subwindow is closed.  Can't change it's color.");
           }
           else
           {
            // The applet sets the color to white.
            // IE only works if test for hex value.
            // Other browsers only work if test for white.
            wind.document.bgColor = (wind.document.bgColor == "#ffffff" ||
                                     wind.document.bgColor == "white") ? "red" :
                                     "white";
           }
         }
      </script>
   </head>
   <body bgcolor="#FFFFFF">
      <b>Here's the applet:</b>
      <br />
      <object name="demoApplet"
              classid="clsid:8AD9C840-044E-11D1-B3E9-00805F499D93"
              codebase="http://java.sun.com/update/1.5.0/jinstall-1_5_0-windows-
                        i586.cab"
              width="200" height="150">
         <param name="code" value="JtoJSDemo.class" />
         <param name="mayscript" value="true" />
         <!--[if !IE]> Non-WinIE Browsers -->
            <object name="demoApplet" classid="java:JtoJSDemo.class"
                    type="application/x-java-applet"
                    width="200" height="150">
               <param name="mayscript" value="true" />
         </object>
         <!-- <![endif]-->
      </object>
      <form name="indicator">
         Is the applet running yet?
         <input type="text" name="running" size="4" value="No" />
      </form>
   </body>
</html>
```

Scripting Plug-Ins

Controlling a plug-in (or Windows ActiveX control in IE) from JavaScript is much like controlling a Java applet, but with more browser-specific concerns to worry about, even at the HTML level. Not all plug-ins are scriptable, of course, nor do all browsers permit such scripting, as mentioned at the start of this chapter. Yet even when you have found the right combination of

browser version(s) and plug-in(s), you must also learn what the properties and/or methods of the plug-in are, so that your scripts can control them. Take a common plug-in duty, such as playing audio: the likelihood that all users will have the same audio playback plug-in installed in a particular browser brand and operating system is perhaps too small to entrust your programming to a single plug-in. If, on the other hand, you are using a plug-in that works only with a special data type, your page just needs to check that the plug-in is installed (and that it is the desired minimum version).

In this section of the chapter, you'll begin to understand the HTML issues, and then examine two separate audio playback examples. One example lets users change tunes being played back; the other arrives with five sounds, each of which is controlled by a different onscreen interface element. Both of these audio playback examples employ a library that has been designed to provide basic audio playback interfaces to the most popular scriptable audio playback plug-in, Windows Media Player, running in Internet Explorer for Windows.

The main goal of the library is to act as an API (Application Programming Interface) between your scripts and the player. The API presents a simple vocabulary to let your scripts control the Windows Media Player. If you want to control an older version of the player, you can modify the details to accommodate that player's API, while leaving your other interface code untouched.

The HTML side

Although several tags have been used in the past to embed plug-ins in web pages, the modern approach involves a single tag only. With the plug-in embedded within the page (even if you don't see it), the plug-in becomes part of the document's object model, which means that your scripts can address it.

The modern and now preferred way to get external media into the document is to load the plug-in as an object through the `<object>` tag. The `object` element is endorsed by the W3C HTML standard, and has been supported in Internet Explorer since IE4. In many ways the `<object>` tag works like the legacy `<applet>` tag, in that aside from specifying attributes that load the plug-in, additional nested `param` elements enable you to make numerous settings to the plug-in while it loads, including the name of the file to pre-load. As with a plug-in's attributes, an object's parameters are unique to the object and are documented (somewhere) for every object intended to be put into an HTML page.

The Windows operating system has a special (that is, far from intuitive) way it refers to the plug-in program (otherwise known in Windows as an ActiveX control): through its class `id` (also known as a `guid`). You must know this long string of numbers and letters in order to embed the object into your page.

The following example is an `object` element that loads the Windows Media Player 9.*x* plug-in (ActiveX control) into a page:

```
<object id="jukebox" width="1" height="1"
        classid="CLSID:6BF52A52-394A-11d3-B153-00C04F79FAA6"
   <param name="URL" value="Beethoven.aif" />
   <param name="autoStart" value="false" />
</object>
```

As compared to older approaches to embedded media objects, the `object` approach has many similar properties and values, which are just expressed differently (for example, `param` elements versus attributes). If you've embedded objects using one of the older techniques, the `object` approach should still be somewhat familiar.

The API approach

In this section, you see one version of an API that is tailor-made for the Windows Media Player. The API has its own initialization routine, which alerts users of ill-equipped browsers with a relevant message about why their browser can't get the most out of the page.

This API is far from the be-all, end-all library, although you will see that it does quite a bit as-is. The code is offered as a starting point for your further development. Such development may take the shape of adding more operations to the API or adding capabilities for additional scriptable plug-ins. For example, although the API as shown supports the newer Windows Media Player versions 9 and up, Microsoft continues to upgrade the Player to new versions (with new `guid`s for your `object` tags) that have new command vocabularies. There is no reason that the API cannot be extended for new generations of Windows Media Player, while maintaining backward compatibility for the version 9 and 10 generation.

You can find the complete API code on the CD-ROM, within the folder of example listings for this chapter. The API file is named `MPAudioAPI.js`. Check out the following high points of this library.

Loading the library

Adding the library to your page is no different from adding any external `.js` library file. Include the following tag in the `head` of your page:

```
<script type="text/javascript" src="MPAudioAPI.js"></script>
```

Except for two global variable initializations, no immediate code runs from the library. All of its activity is invoked from event handlers or other script statements in the main page.

Initializing the library

The first job for the library is to validate that the player has loaded successfully. Before the library can do this, all loading of the `object` elements must be concluded so that the objects exist for the initialization routine to examine. Therefore, use the `onload` event handler in the `body` to invoke the `initAudioAPI()` function. Parameters to be passed to this function are the IDs of the `object` elements that represent individual players (in case there are more than one).

The following is an excerpt from Listing 47-9 (found later in the chapter), which shows how the `jukebox` player object is initialized (all sound files for examples in this chapter have the `.aif` filename extension):

```
onload="initAudioAPI('jukebox')"
```

As you will see later in the chapter, in Listing 47-10, the `initAudioAPI()` function lets you initialize multiple player objects. Each object has its own ID. For example, the following initializes the library for two different embedded plug-in objects:

```
onload="initAudioAPI('cNatural','cSharp')"
```

When the function receives multiple arguments, it loops through them, performing the initializations in sequence. The `initAudioAPI()` function follows:

```
function initAudioAPI()
{
   var args = initAudioAPI.arguments;
   var id;
   for (var i = 0; i < args.length; i++)
   {
      // don't init any more if browser lacks scriptable sound
      if (OKToTest)
      {
         id = args[i];
         players[id] = new API(id);
         validateSupport(id);
      }
   }
}
```

Notice that parameter variables are not explicitly declared for the function, but are, instead, retrieved via the `arguments` property of the function. The global `OKToTest` flag, initialized to `true` when the library loads, is set to `false` if the validation of a plug-in fails. The conditional construction here prevents multiple alerts from appearing when multiple plug-in parameters are passed to the initialization function.

Sound player API objects

One of the jobs of the initialization routine is to create a player object for each plug-in identifier. The object's constructor is as follows:

```
// AudioAPI object constructor
function API(id) {
   this.id = id;
   this.play = API_play;
   this.stop = API_stop;
   this.pause = API_pause;
   this.rewind = API_rewind;
   this.load = API_load;
   this.getVolume = API_getVolume;
   this.setVolume = API_setVolume;
}
```

The object becomes a convenient place to preserve properties for each sound controller. But the bulk of the object is reserved for assigning methods — the methods that your main page's scripts

invoke to play and stop the player, adjust its volume, and so on. The method names to the left of the assignment statements in the object constructor are the names your scripts use; the functions in the library (for example, `API_play()`) are the ones that send the right command to the right plug-in.

Each of these objects (even if there is only one for the page) is maintained in a hash table–like array (named `players[]`) in the library. The plug-in object's identifier is the string index for the array entry. This provides the gateway to your page's scripts. For example, if you initialize the library with a single identifier, `jukebox`, you access the methods of the library's jukebox-related player object through the array and the identifier:

```
players["jukebox"].rewind();
```

Invoking methods

Many of the player's method names are simple enough as-is (e.g., `play()`), but developing an API allows you to devise your own vocabulary for more complex operations. For example, Windows Media Player has no explicit method for rewinding the current tune, but you can assemble the equivalent operations into your own `rewind()` method, as shown in Listing 47-8. When your script invokes `players["jukebox"].rewind()`, the combined operations of the `API_rewind()` method do their jobs.

LISTING 47-8

The API's Primary Functions

```
function API_play(n)
{
   if (document.all(this.id).HasError)
   {
      alert("MediaPlayer Alert: "  + document.all(this.id).ErrorDescription);
   }
   else
   {
      document.all(this.id).settings.playCount = n;
      document.all(this.id).controls.play();
   }
}

function API_stop()
{
   document.all(this.id).controls.stop();
}

function API_pause()
{
   document.all(this.id).controls.pause();
}
```

continued

LISTING 47-8 *(continued)*

```
function API_rewind()
{
   document.all(this.id).controls.stop();
   document.all(this.id).controls.currentPosition = 0;
}

function API_load(URL)
{
   document.all(this.id).URL = URL;
}
```

Building a jukebox

The first example that utilizes the `MPAudioAPI.js` library is a jukebox that provides an interface (admittedly not pretty — that's for you to whip up) for selecting and controlling multiple sound files with a single plug-in tag set. The assumption for this application is that only one sound at a time need be handy for immediate playing.

Listing 47-9 shows the code for the jukebox. All sound files specified in the example are in the same folder as the listing on the companion CD-ROM (the AIFF-format files sound better in some plug-ins than others, so don't worry about the audio quality of these demo sounds).

LISTING 47-9

A Scripted Jukebox

```
<html>
   <head>
      <title>Oldies but Goodies</title>
      <script type="text/javascript" src="MPAudioAPI.js"></script>
      <script type="text/javascript">
         // make sure currently selected tune is preloaded
         function loadFirst(id)
         {
            var choice = document.forms[0].musicChoice;
            var sndFile = choice.options[choice.selectedIndex].value;
            players[id].load(sndFile);
         }
         // swap tunes
         function changeTune(id, choice)
         {
            players[id].load(choice.options[choice.selectedIndex].value);
         }
         // control and display volume setting
         function raiseVol(id)
         {
```

```
            var currLevel = players[id].getVolume();
            currLevel += Math.ceil(Math.abs(currLevel)/10);
            players[id].setVolume(currLevel);
            displayVol(id);
         }
         function lowerVol(id)
         {
            var currLevel = players[id].getVolume();
            currLevel -= Math.floor(Math.abs(currLevel)/10);
            players[id].setVolume(currLevel);
            displayVol(id);
         }
         function displayVol(id)
         {
            document.forms[0].volume.value = players[id].getVolume();
         }
      </script>
   </head>
   <body onload="initAudioAPI('jukebox'); loadFirst('jukebox');
displayVol('jukebox')">
      <form>
         <table border="2" align="center">
            <caption align="top">
               <font size="+3">Classical Piano Jukebox</font>
            </caption>
            <tr>
               <td colspan="2" align="center">
                  <select name="musicChoice"
                          onchange="changeTune('jukebox', this)">
                     <option value="Beethoven.aif" selected="selected">
                        Beethoven's Fifth Symphony (Opening)
                     </option>
                     <option value="Chopin.aif">
                        Chopin Ballade #1 (Opening)
                     </option>
                     <option value="Scriabin.aif">
                        Scriabin Etude in D-sharp minor (Finale)
                     </option>
                  </select>
               </td>
            </tr>
            <tr>
               <th rowspan="4">Action:</th>
               <td>
                  <input type="button" value="Play"

onclick="players['jukebox'].play(parseInt(this.form.frequency[this.form.
frequency.selectedIndex].value))" />
                  <select name="frequency">
                     <option value="1" selected="selected">
                        Once
                     </option>
```

continued

LISTING 47-9 *(continued)*

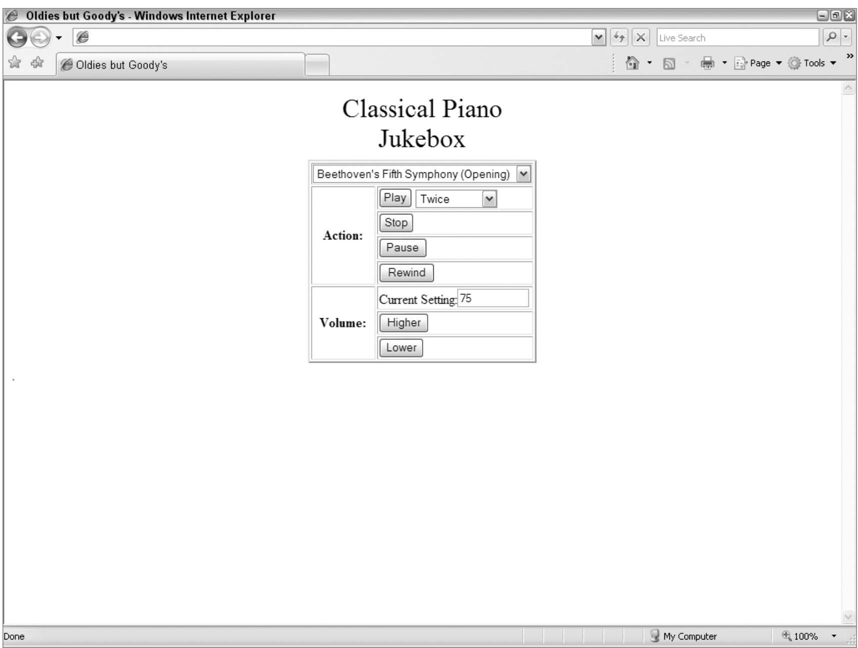
```
            <option value="2">
               Twice
            </option>
            <option value="3">
               Three times
            </option>
            <option value="TRUE">
               Continually
            </option>
         </select>
      </td>
   </tr>
   <tr>
      <td>
         <input type="button" value="Stop"
                onclick="players['jukebox'].stop()" />
      </td>
   </tr>
   <tr>
      <td>
         <input type="button" value="Pause"
                onclick="players['jukebox'].pause()" />
      </td>
   </tr>
   <tr>
      <td>
         <input type="button" value="Rewind"
                onclick="players['jukebox'].rewind()" />
      </td>
   </tr>
   <tr>
      <th rowspan="3">Volume:</th>
      <td>Current Setting:
         <input type="text" size="10" name="volume"
                onfocus="this.blur()" />
      </td>
   </tr>
   <tr>
      <td>
         <input type="button" value="Higher"
                onclick="raiseVol('jukebox')" />
      </td>
   </tr>
   <tr>
      <td>
         <input type="button" value="Lower"
                onclick="lowerVol('jukebox')" />
      </td>
   </tr>
```

```
            </table>
         </form>

         <object id="jukebox" width="2" height="2"
                 classid="clsid:6BF52A52-394A-11d3-B153-00C04F79FAA6">
            <param name="autoStart" value="false" />
         </object>
      </body>
   </html>
```

You can see the user interface in Figure 47-4. One `select` element contains a list of three possible choices. Most of the interface, however, consists of buttons that ultimately invoke methods of the current plug-in.

FIGURE 47-4

The jukebox page.



All functions defined for this page are designed to be as generalizable as possible. Thus, the identifier of the plug-in is passed as a parameter to each. If another plug-in were added to this page, the same functions could be used without modification, provided calls to the functions passed the identifier of the other plug-in.

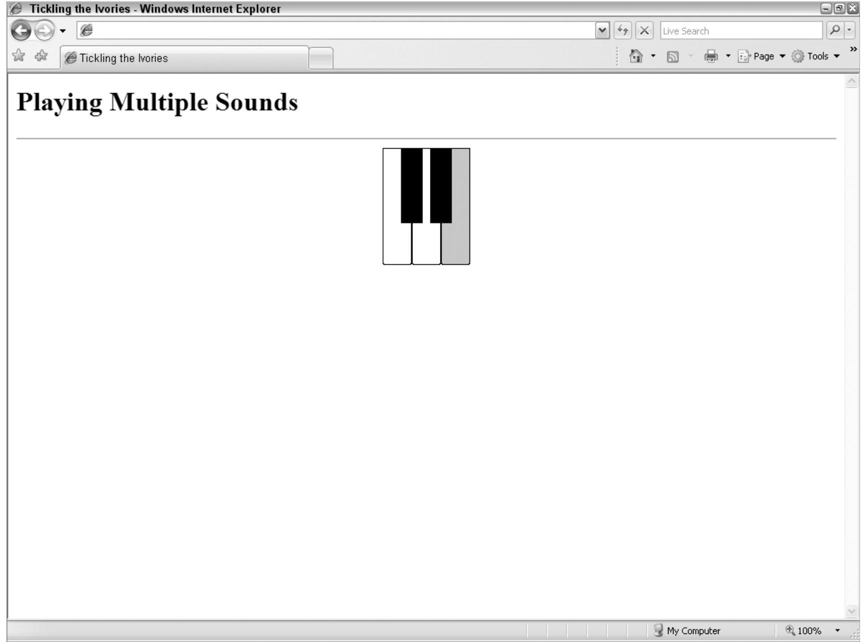
All of the button controls are pretty straightforward except the Play button's `onclick` event handler. It invokes the `players[id].play()` method, but that method requires a parameter of how many times the sound should be played. In this user interface, a `select` element controls that information. Getting the value of the selected item creates a lengthy reference, but that's what is taking up so much space in the parameter slot of the `play()` method call.

Embedding multiple sounds

The final example of embedded media serves as a base on which you can build a page that needs to play multiple sounds without the user explicitly loading them. For example, you may have buttons generate different sounds after users click them. (We're not recommending this interface, but that won't necessarily stop you.) Figure 47-5 shows you a simple five-key piano keyboard. The page loads five different sounds into the page, one for each note (actual piano sounds in this case). Each sound was recorded for about 4 seconds, so that you can get the action of attack and delay, just like a real piano. If you mouse-down on a key, the sound plays for up to 4 seconds (getting softer all the time), or until you mouse-up on the key (the attack time on the sample sounds on the CD-ROM is not instantaneous, so you may have to hold a key down for a fraction of a second to start the sound). The colors of the keys also change slightly to provide further user feedback to the action.

FIGURE 47-5

Controller for five sounds.



Thanks to the MPAudioAPI.js library, very little code in this page is associated with the sounds. Far more is involved with the image swaps and the loading of the five plug-ins. Listing 47-10 shows the code for the page.

LISTING 47-10

Scripting Multiple Sounds

```
<html>
   <head>
      <title>Tickling the Ivories</title>
      <style type="text/css">
         object
         {
            visibility:hidden
         }
         #ivories
         {
            position:relative;
         }
         #keyC
         {
            height:140; width:35; border:0;
         }
         #keyCsharp
         {
            height:90; width:26; border:0;
         }
         #ivorycHalf
         {
            position:absolute; left:22px;
         }
         #keyD
         {
            height:140; width:35; border:0;
         }
         #keyDsharp
         {
            height:90; width:26; border:0;
         }
         #ivorydHalf
         {
            position:absolute; left:57px;
         }
         #keyE
         {
            height:140; width:35; border:0;
         }
      </style>
      <script type="text/javascript" src="MPAudioAPI.js"></script>
      <script type="text/javascript">
         // pre-cache 10 images
```

continued

LISTING 47-10 *(continued)*

```
var onImages = new Array();
onImages["c"] = new Image(35, 140);
onImages["c"].src = "whiteDown.gif";
onImages["d"] = new Image(35, 140);
onImages["d"].src = "whiteDown.gif";
onImages["e"] = new Image(35, 140);
onImages["e"].src = "whiteDown.gif";
onImages["cHalf"] = new Image(26, 90);
onImages["cHalf"].src = "blackDown.gif";
onImages["dHalf"] = new Image(26, 90);
onImages["dHalf"].src = "blackDown.gif";

var offImages = new Array();
offImages["c"] = new Image(35, 140);
offImages["c"].src = "whiteUp.gif";
offImages["d"] = new Image(35, 140);
offImages["d"].src = "whiteUp.gif";
offImages["e"] = new Image(35, 140);
offImages["e"].src = "whiteUp.gif";
offImages["cHalf"] = new Image(26, 90);
offImages["cHalf"].src = "blackUp.gif";
offImages["dHalf"] = new Image(26, 90);
offImages["dHalf"].src = "blackUp.gif";

// swap images (on)
function imgOn(img)
{
   if (document.images)
   {
      document.images[img].src = onImages[img].src;
   }
}
// swap images (off)
function imgOff(img)
{
   if (document.images)
   {
      document.images[img].src = offImages[img].src;
   }
}

// play a note (mousedown)
function playNote(id)
{
   players[id].rewind();
   players[id].play(1);
}
// stop playing (mouseup)
function stopNote(id)
{
   players[id].stop();
```

```
         players[id].rewind();
      }
   </script>
</head>
<body onload="initAudioAPI('cNatural','cSharp','dNatural','dSharp',
             'eNatural')">
   <h1>
      Playing Multiple Sounds
   </h1>
   <hr />
   <table align="center">
      <tr>
         <td>
            <div id="ivories">
            <!-- no spaces between the piano keys -->
            <a href="#"
            onmousedown="playNote('cNatural');imgOn('c');return false"
            onmouseup="imgOff('c');stopNote('cNatural')"><img alt="image"
            name="c" src="whiteUp.gif"
            id="keyC" /></a><a href="#"
            onmousedown="playNote('dNatural');imgOn('d');return false"
            onmouseup="imgOff('d');stopNote('dNatural')"><img alt="image"
            name="d" src="whiteUp.gif"
            id="keyD" /></a><a href="#"
            onmousedown="playNote('eNatural');imgOn('e');return false"
            onmouseup="imgOff('e');stopNote('eNatural')"><img alt="image"
            name="e" src="whiteUp.gif"
            id="keyE" /></a> <span id="ivorycHalf"><a href="#"
            onmousedown="playNote('cSharp');imgOn('cHalf');return false"
            onmouseup="imgOff('cHalf');stopNote('cSharp')"><img
            alt="image" name="cHalf" src="blackUp.gif"
            id="keyCsharp" /></a></span><span id="ivorydHalf"><a href="#"
            onmousedown="playNote('dSharp');imgOn('dHalf');return false"
            onmouseup="imgOff('dHalf');stopNote('dSharp')"><img
            alt="image" name="dHalf" src="blackUp.gif"
            id="keyDsharp" /></a></span>
            </div>
         </td>
      </tr>
   </table>
   <object id="cNatural" width="1" height="1"
           classid="clsid:6BF52A52-394A-11d3-B153-00C04F79FAA6">
      <param name="URL" value="c.aif" />
      <param name="autoStart" value="false" />
   </object>
   <object id="cSharp" width="1" height="1"
           classid="clsid:6BF52A52-394A-11d3-B153-00C04F79FAA6">
      <param name="URL" value="cSharp.aif" />
      <param name="autoStart" value="false" />
   </object>
   <object id="dNatural" width="1" height="1"
           classid="clsid:6BF52A52-394A-11d3-B153-00C04F79FAA6">
```

continued

LISTING 47-10 *(continued)*

```
          <param name="URL" value="d.aif" />
          <param name="autoStart" value="false" />
       </object>
       <object id="dSharp" width="1" height="1"
               classid="clsid:6BF52A52-394A-11d3-B153-00C04F79FAA6">
          <param name="URL" value="dSharp.aif" />
          <param name="autoStart" value="false" />
       </object>
       <object id="eNatural" width="1" height="1"
               classid="clsid:6BF52A52-394A-11d3-B153-00C04F79FAA6">
          <param name="URL" value="e.aif" />
          <param name="autoStart" value="false" />
       </object>
    </body>
 </html>
```

Perhaps the trickiest part of this entire demonstration lies in the way that the keyboard art and user interface are created. Because the white keys are not rectangular, the black key art is dropped atop the white keys by way of positioned elements.

When you use the page, you may notice a slight delay in getting the sound to be heard after pressing down on a key. On older, slower machines, this delay is even more noticeable. Take this behavior into account when designing interactive sound.

Scripting Java Classes Directly

The NPAPI, as implemented in Netscape and Mozilla browsers, allows scripts to access Java classes as if they were part of the JavaScript environment. Because you need to know your way around Java before programming Java classes directly from JavaScript, we won't get into too much detail in this book. Fortunately, the designers of JavaScript have done a good job of creating JavaScript equivalents for the most common Java language functionality, so there is not a strong need to access Java classes on a daily basis.

To script Java classes, it helps to have a good reference guide to the classes built into Java. Though intended for experienced Java programmers, *Java in a Nutshell* (O'Reilly & Associates, Inc.) offers a condensed view of the classes, their constructors, and their methods.

Java's built-in classes are divided into major groups (called *packages*) to help programmers find the right class and method for any need. Each package focuses on one particular aspect of programming, such as classes for user interface design in application and applet windows, network access, and basic language constructs — strings, arrays, and numbers. References to each class (object) defined in Java are "dot" references, just as in JavaScript. Each item following a dot helps zero-in on the desired item. As an example, consider one class that is part of the base language class. The base language class is referred to as

```
java.lang
```

One of the objects defined in `java.lang` is the `String` object, whose full reference is

```
java.lang.String
```

To access one of its methods, you use an invocation syntax with which you are already familiar:

```
java.lang.String.methodName([parameters])
```

To demonstrate accessing Java from JavaScript, we call upon one of Java's `String` object methods, `java.lang.String.equalsIgnoreCase()`, to compare two strings. Equivalent ways are available for accomplishing the same task in JavaScript (for example, comparing both strings in their `toUpperCase()` or `toLowerCase()` versions), so don't look to this Java demonstration for some great new powers along these lines.

Before you can work with data in Java, you have to construct a new object. Of the many ways to construct a new `String` object in Java, use the one that accepts the actual string as the parameter to the constructor:

```
var mainString = new java.lang.String("TV Guide");
```

At this point, your JavaScript variable, `mainString`, contains a reference to the Java object. From here, you can call this object's Java methods directly:

```
var result = mainString.equalsIgnoreCase("tv Guide");
```

Even from JavaScript, you can use Java classes to create objects that are Java arrays, and access them via the same kind of array references (with square brackets) as JavaScript arrays. In a few cases, you can use Java classes to obtain additional information about the user environment, such as the user's IP address (but not email address). The process involves a couple of Java class calls, as follows:

```
var localHost = java.net.InetAddress.getLocalHost();
var IP = localhost.getHostAddress();
```

The more you work with these two languages, the more you see how much Java and JavaScript have in common.

CHAPTER

Debugging Scripts

IN THIS CHAPTER

Identifying the type of error plaguing a script

Interpreting error messages

Preventing problems before they occur

One of the first questions that an experienced programmer asks about a programming environment is what support there is for debugging code. Even the best coders in the world make mistakes when they draft programs. Sometimes, the mistakes are a mere slip of a finger on the keyboard; other times, they result from not being careful with expression evaluation or object references. The cause of the mistake is not the issue: finding the mistake and getting help to fix it is.

Some debugging tools are available for the latest browsers. For the most part, they have come from the browser makers themselves, or they are tied very closely to a particular authoring environment. Some of these tools are very quirky; others require significant investments in authoring environments. Discussion about debugging tools in this chapter, however, focuses on simple tools you can download online. By understanding the true meaning of error messages and working out problems with the tools provided here, you should be able to overcome your bugs.

Syntax versus Runtime Errors

As a page loads into a JavaScript-enabled browser, the browser attempts to create an object model out of the HTML and JavaScript code in the document. Some types of errors crop up at this point. These are mostly syntax errors, such as failing to include a closing brace after a function's statements. Such errors are about structure, rather than about values or object references.

Runtime errors involve failed connections between function calls and their functions, mismatched data types, and undeclared variables located on

the wrong side of assignment operators. Such runtime errors can occur as the page loads if the script lines run immediately. Runtime errors located in functions won't crop up until the functions are called — either as the page loads or in response to user action.

Because of the interpreted nature of JavaScript, the distinction between syntax and runtime errors blurs. But as you work through whatever problem halts a page from loading or a script from running, you have to be aware of differences between true errors in language and your errors in logic or evaluation.

Error Message Notification

As browsers have evolved through several generations, the ways in which script errors are reported to the user (and to you as the author) have also changed. The biggest changes came in WinIE4 and NN4.5. Prior to those versions, script errors always displayed some kind of alert dialog box with information about the error. Because these alerts could confuse non-technical users, the newer browsers are much more subtle about the presence of errors. In fact, the notification mechanism is so subtle, it is easy to miss the fact that a script error has occurred. Even if you do notice, you must then exercise your mouse a bit more to view the details.

When a script error occurs in WinIE4+, the status bar displays a yellow alert icon, plus a brief text message indicating that an error has occurred. A syntax error that occurs while the page loads usually signifies that the page has loaded, but with errors. A runtime error's message simply indicates that an error occurred. To view details about the error, you must double-click the yellow icon in the status bar. The default appearance of the error message alert dialog box window includes a button named Show Details. Clicking this button expands the window to reveal whatever details the browser is reporting about the error.

If you leave the script error window expanded, the next time it opens, it will again be expanded. It is a good idea for scripters to also check the box that forces the browser to show the error dialog box whenever an error occurs. This is simply a shortcut to manually double-clicking the status bar error icon.

Mozilla console windows

In Mozilla-based browsers, error notifications are entirely hidden from the end user. However, if you type

```
javascript:
```

into the toolbar's Location text box, or into the dialog box that enables you to open a new page, an entirely new, non-modal window is displayed. This window is called the JavaScript Console, and it keeps track of JavaScript errors. You can also open the JavaScript Console by selecting JavaScript Console from the Tools menu in Firefox, for example. The console has been renamed Error Console, starting with Firefox 2.0.

In contrast to the one message per window approach of IE, the JavaScript Console window continues to record all script errors in sequence (in a scrolling frame), even when the Console window is closed. You can keep this window open all the time, and simply bring it to the front whenever you need to view errors.

Unlike earlier Netscape browsers, Mozilla does not provide notification of errors in the status bar, so it is up to you to be vigilant for something running amok. This is all the more reason to keep the JavaScript Console window open while you are writing and debugging your scripts. Even if things appear to be OK, periodically check the Console window to be sure.

Safari errors

As delivered to users, early versions of Safari completely concealed script errors. To be able to see script errors at all, you must enter a command into the Mac OS X Terminal program window to display an otherwise hidden Debug menu. With Safari closed, launch Terminal and enter the following command exactly as shown:

```
defaults write com.apple.Safari IncludeDebugMenu 1
```

Now launch Safari, where the Debug menu should appear in the menu bar, with the Log JavaScript Exceptions item already checked (if not, select it so that a check mark appears next to the item). Choose the Show JavaScript Console item to display the JavaScript Console window.

In later versions of Safari, whether on Mac OS or Windows, simply open Safari Preferences. On the Advanced tab, select “Show Develop menu in menu bar.” The Develop menu will appear on the menu bar with several options available, including Show Error Console and Start Debugging JavaScript.

Opera errors

Selecting Advanced on the Tools menu enables you to choose Developer Tools or the Error Console. (Developer Tools includes the Error Console.) A separate window is not opened in Opera. Rather, the current window is split. This is not that obvious; the splitter bar appears almost at the bottom of the window. Use the mouse to move the splitter bar so that you can better see the console. To the right of the embedded console is an icon to unlock it into a separate window.

Chrome errors

Google Chrome does not have a traditional menu bar. To the right of its tabs are two icons. One is the “Control the current page” icon. If you click on that, you’ll see a Developer selection, which allows you to open the JavaScript console.

Multiple error messages

The modality of IE error message alert dialog boxes tends to force just one message to appear. In other words, when the first error occurs, the browser stops recording further errors. In other

browsers, however, it is not uncommon for multiple errors to be listed (or, in older versions, multiple error windows to show up). But you need to understand how to treat multiple errors to get to the root of the problem.

The usual reaction is to look at the last message to appear in the sequence. That, however, is usually the error message least likely to lead you to the true problem. Error messages are dumped to the Error Console windows in the order in which they occur. This means that the first error in the list is the most important error message of them all. More than likely, the first error points to a problem that throws off the rest of the script, thus triggering all of the other error messages. For example, if a statement that initializes a variable has a syntax error in it, all other statements that rely on that variable will fail, because the variable appears to be undefined.

When you encounter multiple errors, don't start any serious debugging until you locate the first error message. You must tackle this one before any others. The solution to the first one may cause the other errors to go away. This is all the more reason, when authoring in any browser, to keep the Console window open, and to clear it before loading any page or executing any scripts.

Error Message Details

Each browser reports errors slightly differently, and often with different levels of specificity. By and large, however, you can count on error details to include three basic clues to help you track down the error: the file in which the error occurred, the location of the error within the source code, and a textual description of the error. The Mozilla-based browsers tend to furnish the most accurate and helpful error messages.

Error filename

Although MacIE error messages do not explicitly reveal the name of the file whose source code contains the error, in practice, Opera, recent versions of WinIE, and the Mozilla browsers do the best job of telling the truth. Of course, when the script and the HTML are all on one page, it doesn't require a brain surgeon to know that the error occurs from that page's source code. But if you link in external `.js` libraries, Opera, recent versions of WinIE, and the Mozilla browsers provide the URL to the `.js` file. Early versions of WinIE, on the other hand, indicates the HTML page that loads the external library, making it difficult to know precisely where the error is.

Error location

All browsers provide a source code line number and character position where the error supposedly occurs. For self-contained pages with no dynamically created content, the reporting tends to be accurate (also see the IE "Object expected" error message details described later in this chapter), but the accuracy is much closer in Opera and the Mozilla browsers than in IE. And if your page links in an external library, the line number provided by MacIE and older versions of WinIE is practically useless. The sense you get is that the lines of the `.js` file become embedded within the main page's script, but how that is supposed to help an author find the precise

problem line is a mystery — even the most feature-laden text editor knows only how to display line numbers for a single document.

Opera and Mozilla browsers, however, not only point to the correct `.js` file, but to the line number within that file. You are much more likely to get to the root of a problem, especially in an external `.js` file, through Opera and Mozilla error messages.

Line number reporting has improved with each browser generation, but anomalies still exist. Perhaps the most egregious is the tendency for IE to report a problem at a line number whose source code is HTML with an event handler. The problem, it turns out, will be somewhere in the function being invoked by the event handler. Another possibility in all browsers is that the line number being reported is below the line that contains the problem. Consider the following simple source code listing (with line numbers from the source code editor) that intentionally contains a syntax error (a missing brace after the first function):

```
1:    <html>
2:       <head>
3:          <script type="text/javascript">
4:          function tarzan() {
5:             var x = 1;
6:
7:          function jane() {
8:             var y = 3;
9:          }
10:         </script>
11:      </head>
12:      <body>
13:      Hello.
14:      </body>
15:   </html>
```

When you load this page into browsers, all of them report a problem with a missing right brace (Mozilla is a bit more explicit with its message, indicating that a right brace is missing after a function body). But where do the browsers point to the error? By looking at the code as a human, you can see that the missing brace belongs in Line 6. But now examine the code from the point of view of a script interpreter engine: It sees the opening brace on Line 4, and then a new function declaration starting on Line 7. To the interpreter, this means that the `jane()` function is probably nested inside the `tarzan()` function, and it is the `tarzan()` function that is lacking the right brace following the `jane()` function. Therefore, the error line number comes in at Line 10 (although MacIE5 reports Line 9). Your scripts won't likely be this simple, so the distance between the reported error line number and the location of the actual problem can be substantial, and difficult to spot without using some of the tips and tools described later in this chapter. This example also demonstrates how the writing style you use in your code can help identify or hide problems from you as a human. The writing style in this example contains the opening curly brace for a function's block of code on the same line as the `function` keyword. It can be argued that if the opening curly brace had been on the line following, it would have been easier for you, a human, to spot the problem visually.

IE sometimes has a nasty habit of identifying the location of the problem at Line 1, Character 1. All this means is that you need to put your detective skills to work that much harder. Common causes for this behavior are references to HTML objects that don't exist (or a mismatch between the identifier of the element and your script reference to it) and errors that affect global functions or window methods. To find the genuine problem line, you can use tracing techniques described later in this chapter.

Error message text

Because so many permutations exist of the potential errors you can make in scripts and the ways the JavaScript interpreters in different browsers regard these errors, presenting hard-and-fast solutions to every JavaScript error message is impossible. What we can do, however, is list the most common and head scratch–inducing error messages, and relate the kinds of non-obvious problems that can trigger such messages.

"Object expected"

This error message is often one of the least helpful that you see in IE. The line number associated with the message typically points to a line in the source code that invokes a function. If you define event handlers as attributes of element tags, the line number being reported may correspond to the line containing that HTML tag.

The most obvious problem is that the function being invoked is not regarded as a valid function in the page (the "object" referred to here is the function object). This problem can be the result of an HTML or script error earlier in the document. The problem can also be the result of some error in the function itself that failed to let the interpreter treat the function as a genuine function object. Most typically, these kinds of problems are detected as syntax errors while the page loads (for example, an imbalanced set of parentheses or braces), but not always.

As a first-strike tactic, you need to determine if the function is being invoked at all. By placing an alert in the first line of the function and triggering the function, you can see if script execution is reaching that point. If that works okay, move the alert downward through the function to find out where the error is actually being triggered. The line before the alert that fails is the likely culprit. As rudimentary a technique as it may be, carefully placing an alert here or there can go a long way toward tracking down bugs.

"Expected <something>"

This message usually points straight at the problem line. Most of the "things" that the statement expects are self-explanatory. If a right parenthesis is missing from a pair, that is the "thing" shown to be expected. Detecting in the message the difference between a brace and parenthesis isn't always easy, so look at the message carefully. Not quite as intuitive is when the message says "Expected identifier." This error refers to an expression that typically is trying to use a reserved word as a variable name. See Appendix C for a list of reserved words, none of which you may use as names of things or variables.

“<Something> is undefined”

This message is fairly easy to understand, yet at times difficult to diagnose. For variable names, the message usually means that you have an uninitialized variable name sitting in the position of a right-hand operand or a unary operand. This variable name has not been declared or assigned with any value prior to this erroneous statement. Perhaps you’re attempting to use a variable name that has been initialized only as a local variable in another function. You may also have intended the right-hand value to be a string, but you forgot to enclose it in quotes, forcing JavaScript to look upon it as a reference to something. Another possibility is that you misspelled the name of a previously declared variable. JavaScript rightly regards this item as a new, undeclared variable. Misspellings, you will recall, include errors in upper- and lowercase in the very case-sensitive JavaScript world.

If the item is a function name, you may have to perform a bit of detective work. Though the function may be defined properly, a problem in the script above the function (for example, imbalanced braces) makes JavaScript fail to see the function. In other cases, you may be trying to invoke a function in another window or frame, but forgot to include the reference to that distant spot in the call to the function.

A less likely case, but a confusing one to diagnose, is when you are assembling string versions of function calls or array references out of literal strings and variable values. The following simplified example is assembling a string that is a function call to be triggered by `setTimeout()`:

```
function doA()
{
   var x = "joe";
   setTimeout("doB(" + x + ")", 5000);
}
```

Even though the value of `x` is a string when it is concatenated to the call to the `doB()` function, the value gets evaluated as if it were a variable name. An error crops up saying that “joe is undefined.” Because you want to pass the value of `x` as a parameter, you must nest its value inside a pair of quotes, as follows:

```
function doA()
{
   var x = "joe";
   setTimeout("doB('" + x + "')", 5000);
}
```

The difference in the code is extremely subtle, but absolutely necessary.

“<Something> is not a function”

As with the preceding message, this error message can be one of the most frustrating, because when you look at the script, it appears as though you have clearly defined a function by that name, and you’re simply having an event handler or other running statement call that function. The first problems to look for are mismatched letter cases in the calling statement and function, and the reuse of a variable or HTML object name by the function name.

This latter item is a no-no — it confuses JavaScript into thinking that the function doesn't exist, even though the object name doesn't have parentheses appended to it, and the function does. We've also seen this error appear when other problems existed in the script above the function named in the error message, and the named function was the last one in a script.

"Object doesn't support this property or method"

This IE message reports that a valid object does not provide support for a method you just attempted to invoke. In practice, this message rarely appears as the result of referencing an object's nonexistent property, because the language allows for extending an object's list of properties by assignment. One scenario where you may encounter this message is when testing a page in MacIE that was developed in WinIE, since the MacIE complement of implemented object methods is somewhat smaller.

"Unterminated string literal"
"Unterminated string constant"

Mozilla is far more helpful than other browsers with this type of message, because along with the error message, it displays the code fragment that tripped the error. You will see the beginning (or all) of the string that is the culprit. If you simply forgot to close a string quote pair, the error most frequently appears when you try to concatenate strings or nest quoted strings. Despite the claim that you can nest alternating double and single quotes, we often have difficulties using this nesting method beyond the second nested level (single quotes inside a double-quoted string). At different times, we've gotten away with using a pair of `\"` inline quote symbols for a third layer. If that syntax fails, we break up the string so that nesting goes no deeper than two layers. If necessary, we even back out the most nested string and assign it to a variable in the preceding line — concatenating it into the more complex string in the next line.

"Missing } after function body"
"Expected }"

This error usually is easy to recognize in a simple function definition because the closing brace is missing at the end of the function. But when the function includes additional nested items, such as `if...else` or `for` loop constructions, you begin dealing with multiple pairs of braces within the function. The JavaScript interpreter doesn't always determine exactly where the missing brace belongs, and thus it simply defaults to the end of the function. This location is a natural choice, we guess, because from a global perspective of the function, one or more of the right braces that ripple down to the end of the function usually are missing.

In any case, this error message means that a brace is missing somewhere in a function above the referenced line number. Do an inventory count for left and right braces, and see whether a discrepancy occurs in the counts. One of those nested constructions is probably missing a closing brace. Some programmer-oriented text editors also include tools for finding balanced pairs of braces and parentheses.

“<Something> is not a number”

The variable name singled out in this error message is most likely a string or null value. The line of JavaScript that trips it up has an operator that demands a number. When in doubt about the data type of a variable destined for a math operation, use the `parseInt()` or `parseFloat()` functions to convert strings to numbers.

We have also encountered this error when it provides no clue about what isn’t a number — the error message simply says, “is not a number.” In our case, the root of the problem ended up having nothing to do with numbers. A structural imbalance in the script triggered this bogus error message.

“<Something> has no property named . . .”
“<Something> has no properties”

When a statement trips this error message, an object reference has usually gone awry in an assignment or comparison expression. You probably attempted to reference a property of an object, but something is wrong with the object reference, or you’re trying to retrieve a property that doesn’t exist for that object. If the reference is an extended one, you may have to dig to find the precise problem with the reference. Consider the following two statements that attempt to access the `value` property of a button named `calcMe`:

```
document.forms.calcme.value
document.forms[0].calcme.value
```

The Mozilla errors for these two statements would read “document.forms.calcme has no properties” and “document.forms[0].calcme has no properties.” Causes for the two errors are quite different. The obvious problem with them both may seem to be that the button’s name is incorrectly referenced as `calcme` instead of `calcMe`. That, indeed, is the error for the second statement. But a more fundamental problem also plagues the first statement: the `document.forms` reference (a valid object, returning an array of forms) needs an array index in this instance, because it needs to look into a particular form for one of its objects. Unfortunately, both error messages look alike at first glance, and you cannot tell from them which statement has two errors and which has one.

But what you can do when this kind of error appears is use the reference that is returned with the error message to check your work. Start verifying the accuracy of your references from left to right. Later in this chapter, you see how to use the embeddable Evaluator tool to verify the existence of object references.

“<Something> is null or not an object”

This message is the IE version of the previous Mozilla error message. A big difference is that the reference returned as part of the error message includes the complete reference. Therefore, a reference to a nonexistent `calcme` button in a form yields the error message “‘document.forms[0].calcme.value’ is null or not an object”. Your first instinct is to be suspicious of the `value` property part of the reference. The detective work to find the problem is the same as in the Mozilla version: verify the reference piece by piece, working from left to right. Again, the embeddable Evaluator described later in this chapter can assist in this task.

"<Something> has no property indexed by [i]"

Look carefully at the object reference in this error message. The last item has an array index in the script, but the item is not an array value type. Users commonly make this mistake within the complex references necessary for radio buttons and `select` options. Make sure that you know which items in those lengthy references are arrays and which are simply object names that don't require array values.

"<Something> can't be set by assignment"

This error message tells you either that the property shown is read-only or that the reference points to an object, which must be created via a constructor function rather than by simple assignment.

"Test for equality (==) mistyped as assignment (=)? Assuming equality test"

The first time I received this error, I was amazed by JavaScript's intelligence. I had, indeed, meant to use the equality comparison function (==) but had entered only a single equal sign. JavaScript is good at picking out these situations where Boolean values are required. In Mozilla, this message is demoted to just a warning rather than an error.

"Function does not always return a value"

Often while designing deeply nested `if...else` constructions, your mind follows a single logic path to make sure that a particular series of conditions is met, and that the function returns the desired values under those conditions. What is easy to overlook is that there may be cases in which the decision process may "fall through" all the way to the bottom without returning any value, at which point the function must indicate a value that it returns, even if it is a zero or empty (but most likely a Boolean) value. JavaScript checks the organization of functions to make sure that each condition has a value returned to the calling statement. The error message doesn't tell you where you're missing the return statement, so you have to do a bit of logic digging yourself.

"Access disallowed from scripts at <URL> to documents at <URL>" "Access is denied"

These messages (Mozilla and IE versions, respectively) indicate that a script in one frame or window is trying to access information in another frame or window that has been deemed a potential security threat. Such threats include any `location` object property or other information about the content of the other frame when the other frame's document comes from a protocol, server, or host that is different from the one serving up the document doing the fetching.

Even the best of intentions can be thwarted by these security restrictions. For example, you may be developing an application that blends data in cooperation with another site. Security restrictions, of course, don't know that you have a cooperative agreement with the other web site, and you have no workaround for accessing a completely different domain unless you use signed scripts for Mozilla (see Chapter 49, "Security and Netscape Signed Scripts"), or an IE user has browser security levels set dangerously loose.

Another possible trigger for these errors is that you are using two different servers in the same domain or different protocols (for example, using `https:` for the secure part of your commerce site, while all catalog info uses the `http:` protocol). If the two sites have the same domain (for example, `giantco.com`) but different server names or protocols, you can set the `document.domain` properties of documents so that they recognize each other as equals. See Chapter 49 for details on these issues and the restrictions placed on scripts that mean well, but that can be used for evil purposes.

IE, especially Windows versions, frequently clamps down too severely on inter-window and inter-frame communication. Don't be surprised to encounter security problems trying to communicate between a main window and another window whose content is dynamically generated by scripts in the main window. This error can be incredibly frustrating. Sometimes, serving the main page from a server (instead of reading it from a local hard disk) can solve the problem, but not always. You are safest if the content of both windows or frames are documents served from the same server and domain.

"Unspecified error"

This completely unhelpful IE error message is not a good sign because it means that whatever error is occurring is not part of the well-traveled decision tree that the browser uses to report errors. All is not lost, however. That the browser has not crashed means that you can still attempt to get at the root of the problem through various tracing tactics described later in this chapter.

"Uncaught exception"

You may encounter these messages in Mozilla, although usually not as a result of your scripts, unless you are using some of the browser's facilities to dive into inner workings of the browser. These messages are triggered by the browser's own programming code, and indicate a processing error that was not properly wrapped inside error trapping mechanisms. The details associated with such an error point to the Mozilla browser's own source code modules and internal routines. If you can repeat the error and can do so in a small test case page, you are encouraged to submit a report to `http://bugzilla.mozilla.org`, the bug-tracking site for the Mozilla browser.

"Too many JavaScript errors"

You may see this message in Mozilla if it detects a runaway train generating errors uncontrollably. This message was far more important in the bygone days of separate error windows, because a buggy repeat loop could cause Navigator to generate more error windows than it could handle.

Warnings in Mozilla's Console

The Mozilla browser's JavaScript/Error Console window reports both outright errors and another class of notices called *warnings*. Whereas an error may cause script execution to stop, a warning

is generally less catastrophic. A warning typically alerts the content author of looming problems, such as deprecated DOM terms that are still supported merely for backward compatibility — but that should be avoided going forward.

Another common warning advises that the server delivered some external content without correct headers. This happens frequently when a server supplies `.css` filters for link elements without the correct content-type header (`text/css` or similar). These are server configuration issues that should be fixed if they're under your control. You may turn off warnings in the JavaScript Console window by clicking the Errors button.

Sniffing Out Problems

It doesn't take many error-tracking sessions to get you in the save-switch-reload mode quickly. Assuming that you know this routine (described in Chapter 3, "Selecting and Using Your Tools"), the following are some techniques we use to find errors in our scripts when the error messages aren't being helpful in directing us right to the problem.

Check the HTML tags

Before we look into the JavaScript code, we review the document carefully to make sure that we've written all our HTML tags properly. That includes making sure that all tags have closing angle brackets and that all tag pairs have balanced opening and closing tags. Digging deeper, especially in tags near the beginning of scripts, we make sure that all tag attributes that must be enclosed in quotes have the quote pairs in place. Running your document through an HTML validator can help you spot markup mistakes in complex documents. The W3C's Markup Validation Service is a good place to start (`http://validator.w3.org/`).

A browser may be forgiving about sloppy HTML as far as layout goes, but the JavaScript interpreter isn't as accommodating. Finally, we ensure that the `<script>` tag pairs are in place (they may be in multiple locations throughout the document) and that the `type="text/javascript"` attribute value has both of its quotes.

View the source

Your success in locating bugs by viewing the source in the browser varies widely with the kind of content on the page and the browser you use. Very frequently, authors place perhaps too much importance in what they see in the source window. For a straight, no-frame HTML page, viewing the source provides a modicum of comfort by letting you know that the entire page has arrived from the server.

Examining the source code for framesetting documents or individual frames, you must first give focus to the desired element. For an individual frame, click in the frame, and then right-click (or Ctrl-click on the Mac) on the frame's background to get the contextual menu. One of the items should indicate a source view of the frame. To view the framesetter's source, press the Tab key

until the Address/Location field of the browser is selected. Then choose to view the source from the Edit menu.

Where the source view would be most helpful, but often fails, is to display dynamically generated HTML. Firefox's Web Developer, View Source menu enables you to View Generated Source. Otherwise, your best chance will be for pages whose entire content is generated by script. This is about the only place you can appreciate the difference between `document.write()` and `document.writeln()`, because the latter puts carriage returns after the end of each string passed as a parameter to the method. The result is a more readable source view. Most modern browsers display the HTML as written by your script.

But when only part of the page is generated by script, few browsers combine the hard-wired and dynamic code in the source view. Instead, you see only the hard-wired HTML and scripts. To work around this for IE4+ and Mozilla (other than Firefox), you can use the embeddable Evaluator (later in this chapter) and read the `innerHTML` property of any elements you want.

Timing problems

One problem category that is very difficult to diagnose is the so-called timing problem. There are no hard-and-fast rules that govern when you are going to experience such a problem. Very experienced scripters develop an instinct for when timing is causing a problem that has no other explanation.

A timing problem usually means that one or more script statements are executing before the complete action of an earlier statement has finished its task. JavaScript runs within a single thread inside the browser, meaning that only one statement can run at a time. But there are times when a statement invokes some browser service that operates in its own thread, and therefore doesn't necessarily finish before the next JavaScript statement runs. If the next JavaScript statement relies on the previous statement's entire task having been completed, the script statement appears to fail, even though it actually runs as it should.

These problems crop up especially when scripts work with another browser window, and especially in IE for Windows (ironic in a way). In discussions in this book about form field validation, for example, we recommend that after an instructive alert dialog box notifies the user of the problem with the form, the affected text field should be given focus and its content selected. In WinIE, however, after the user closes the alert dialog box, the script statements that focus and select operate before the operating system has finished putting the alert away and refreshing the screen. The result is that the focused and selected text box loses its focus by the time the alert has finally disappeared.

The solution is to artificially slow down the statements that perform the focus and select operations. By placing these statements in a separate function, and invoking this function through the `window.setTimeout()` method, the browser catches its breath before executing the separate function — even when the delay parameter is set to zero. A similar delay is utilized when opening and writing to a new window, as shown in the example for `window.open()` in Chapter 27, "Window and Frame Objects."

If a timing problem is caused by a time-consuming operation of some sort, you might consider performing the operation asynchronously within the confines of an Ajax construct so that you get a notification when it is complete. Structured in this manner, you can design your code so that it deliberately waits around until the time-consuming operation is finished, thereby guaranteeing that the figurative horse is indeed before the cart.

Cross-Reference

For more on Ajax and asynchronous JavaScript, check out Chapter 39, ‘‘Ajax, J4X, and XML Objects,’’ and Chapter 60, ‘‘Application: Transforming XML Data’’ (on the CD). ■

Reopen the file

If we make changes to the document that we truly believe should fix a problem, but the same problem persists after a reload, we reopen the file through the File menu. Sometimes, when you run an error-filled script more than once, the browser's internals get a bit confused. Reloading does not clear the bad stuff, although sometimes an unconditional reload (clicking Reload while holding Shift) does the job. Reopening the file, however, clears the old file entirely from the browser's memory and loads the most recently fixed version of the source file. We find this situation to be especially true with documents involving multiple frames and tables, and those that load external `.js` script library files. In severe cases, you may even have to restart the browser to clear its cobwebs, but this is less necessary in recent browsers. You should also consider turning off the cache in your development browsers.

Another scenario where the browser cache can cause you fits is when you are loading and accessing XML data from an external file. The browser may very well reload the HTML document and relevant JavaScript code just fine, only to open the XML file from the cache. We've banged our heads on our keyboards many a time trying to figure out why our JavaScript code won't process the XML data properly, only to find out that changes in the XML document were never entering the equation due to the browser cache. Either turn off the cache or make sure to clear it between changes to any XML data files that impact your JavaScript code.

Find out what works

When an error message supplies little or no clue about the true location of a runtime problem, or when you're faced with crashes at an unknown point (even during document loading), you need to figure out which part of the script execution works properly.

If you have added a lot of scripting to the page without doing much testing, we suggest removing (or commenting out) all scripts except the one(s) that may get called by the document's `onload` event handler. This is primarily to make sure that the HTML is not way out of whack. Browsers tend to be quite lenient with bad HTML, so this tactic won't necessarily tell the whole story (specifying a `DOCTYPE` and running your HTML through a validator will squelch bad HTML problems). Next, add back the scripts in batches. Eventually, you want to find where the problem really is, regardless of the line number indicated by the error message alert.

To narrow down the problem spot, insert one or more alert dialog boxes into the script with a unique, brief message that you will recognize as reaching various stages (such as `alert("HERE-1")`). Start placing alert dialog boxes at the beginning of any groups of statements that execute, and try the script again. Keep moving these dialog boxes deeper into the script (perhaps into other functions called by outer statements), until the error or crash occurs. You now know where to look for problems. See also an advanced tracing mechanism described later in this chapter.

Comment out statements

If the errors appear to be syntactical (as opposed to errors of evaluation), the error message may point to a code fragment several lines away from the problem. More often than not, the problem exists in a line somewhere above the one quoted in the error message. To find the offending line, begin commenting out lines one at a time (between reloading tests), starting with the line indicated in the error message. Keep doing this until the error message clears the area you're working on and points to some other problem below the original line (with the lines commented out, some value is likely to fail below). The most recent line you commented out is the one that has the beginning of your problem. Start looking there.

Check runtime expression evaluation

We've said many times throughout this book that one of the two most common problems scripters face is an expression that evaluates to something you don't expect (the other common problem is an incorrect object reference). In lieu of a debugger that would let you step through scripts one statement at a time while watching the values of variables and expressions, you have a few alternatives to displaying expression values while a script runs.

The simplest approaches to implement are an alert box and the status bar. Both the alert dialog box and status bar show you almost any kind of value, even if it is not a string or number. An alert dialog box can even display multiple-line values.

Because most expression evaluation problems come within function definitions, start such explorations from the top of the function. Every time you assign an object property to a variable or invoke a string, math, or date method, insert a line below that line with an `alert()` method or `window.status` assignment statement (`window.status = someValue`) that shows the contents of the new variable value. Do this one statement at a time, save, switch, and reload. Study the value that appears in the output device of choice to see if it's what you expect. If not, something is amiss in the previous line involving the expression(s) you used to achieve that value.

This process is excruciatingly tedious for debugging a long function, but it's absolutely essential for tracking down where a bum object reference or expression evaluation is tripping up your script. When a value comes back as being undefined or null, more than likely the problem is an object reference that is incomplete (for example, trying to access a frame without the `parent.frames[i]` reference), using the wrong name for an existing object (check case), or accessing a property or method that doesn't exist for that object.

When you need to check the value of an expression through a long sequence of script statements or over the lifetime of a repeat loop's execution, you are better off with a listing of values along the way. In the section "A Simple Trace Utility" later in this chapter, we show you how to capture trails of values through a script.

Debugging Tools

For a truly interactive debugger to work effectively with client-side JavaScript, the debugger must be highly integrated into the browser. This appears to have been a primary stumbling block in the availability of JavaScript debuggers. At best, we have some browser-specific debuggers, but this leaves other browsers completely out in the cold. Modern browsers that currently have debuggers available for them are Internet Explorer for Windows, Mozilla (including one specifically for Firefox), and Safari.

WinIE's Script Debugger

You can download the Microsoft Script Debugger and documentation from `http://www.microsoft.com/downloads/details.aspx?familyid=2f465be0-94fd-4569-b3c4-dffdf19ccd99&displaylang=en`.

After you install the Script Debugger, you must tell your copy of IE to enable it. Go to the Tools ➪ Internet Options menu, click the Advanced tab, and look for the "Disable script debugging (Internet Explorer)" item in the Browsing category. Make sure that this check box is cleared. For good measure, quit and relaunch the browser. There is another option called "Disable script debugging (Other)" that applies to script errors as they are handled in other applications that utilize the IE browser engine.

When a runtime error occurs, you will be brought to the Debugger window, which will highlight the affected line. Script execution is paused, enabling you to inspect the value of variables and such. Open the Command window, and type any variable name (local or global) that should be active at the paused line. Press Enter, and the next line of the Command window shows the value of that variable.

You can also explicitly set breakpoints in the code, and single-step through the code. At each step, you can enter another variable name or property reference to see its current value.

Be prepared, however, for a less than satisfactory experience with this debugger. If your scripts are loaded from external `.js` files, the debugger does not track that code or line numbers reliably. In fact, it's not uncommon for the debugger to open with the completely wrong line of code highlighted, making single stepping impossible. Simply put, don't bring high expectations to the Script Debugger.

Perhaps Microsoft would rather you use the more sophisticated debugging facilities of their Visual Studio development software. It is a rather expensive solution if all you need is script debugging.

Mozilla's Venkman Debugger

Venkman is the code name of Mozilla's official debugger that is designed for use in Mozilla-based browsers. Venkman is automatically installed in some Mozilla browsers, namely Netscape browsers, but you'll likely need to download and install it in other Mozilla browsers, such as Firefox and Camino. You can easily tell if Venkman is installed in your browser by clicking Tools ➪ Web Development. If you see JavaScript Debugger on the menu, then you're good to go; otherwise, download Venkman from `http://www.mozilla.org/projects/venkman/`.

When the debugger is open, any script error brings the debugger window to the front, with execution paused at the problem line. If the code is from an external `.js` file, that file's source code is displayed, with the problem line highlighted. This debugger automatically tracks objects and their property values in one of the window's panes. You may also set breakpoints in code so that you can single-step through code and observe values at each step.

Firefox's FireBug Debugger

Perhaps the overall best of JavaScript debuggers is the debugger developed exclusively for Firefox. We're referring to FireBug, which is a lightweight debugger with tight integration into the Firefox browser. Similar to the error notification icon in the lower-left corner of the IE browser window, FireBug alerts you to a script problem via a small status bar icon. You can then open up the FireBug console within Firefox and begin analyzing the problematic code.

One interesting feature unique to FireBug is an Ajax log that enables you to carefully track Ajax requests as they are made and responded to. There is also a JavaScript command line where you can execute JavaScript statements and examine variables.

FireBug was developed by Joe Hewitt and is freely available for download from `http://joehewitt.com/software/firebug/`.

WebKit's Drosera Debugger

Relatively new on the debugging scene is a debugger exclusively for WebKit-based browsers (such as Safari and Chrome) called Drosera. Unlike Venkman, which is a reference to a character from the movie Ghostbusters, Drosera takes a more literal approach to its name in that Drosera is the largest genera of bug-eating plants. Drosera is a modern debugger with most of the bells and whistles found in other full-featured JavaScript debuggers such as Firebug.

Drosera is automatically included in the WebKit application framework that forms the basis for the Safari and Chrome browsers. So, by updating your WebKit installation you will automatically get the latest Drosera debugger. WebKit is updated regularly, and is freely available for download at `http://nightly.webkit.org/`.

Opera's Dragonfly Debugger

You can view documentation for Opera's script debugger at `http://dev.opera.com/articles/view/how-to-debug-javascript-problems-with-op/`. In a similar manner

to the other debuggers, you can step through scripts and view threads. You can also examine the DOM.

Using the embeddable Evaluator

As soon as a page loads, or after some scripts run, the window contains objects whose properties very likely reveal a lot about the environment at rest (that is, not while scripts are running). Those values are normally disguised from you, and the only way to guarantee successful access to view those values is through scripting within the same window or frame. That's where the embeddable Evaluator comes in handy.

As you probably recall from Chapter 4, "JavaScript Essentials," and the many example sections of Parts III and IV of this book, the code within the standalone Evaluator provides two text boxes for entry of expressions (in the top box) and object references (the bottom box). Results of expression evaluation and object property dumps are displayed in the Results `textarea` between the two input boxes. A compact version of The Evaluator is contained by a separate library version called `evaluator.js` (located in the Chapter 48 folder of listings on the CD-ROM). As you embark on any substantial page development project, you should link in the library with the following tag at the top of your `head` section:

```
<script type="text/javascript" src="evaluator.js"></script>
```

Be sure to either have a copy of the `evaluator.js` file in the same directory as the document under construction or to specify a complete `file:` URL to the library file on your hard drive for the `src` attribute.

Immediately above the closing `body` tag of your document, include the following:

```
<script type="text/javascript">
   printEvaluator();
</script>
```

If your page contains lots of positioned content, you'll need to put The Evaluator into its own positioned layer, out of the way of your primary content. For example:

```
<div style="position:absolute; top:800px; left:100px">
   <script type="text/javascript">
      printEvaluator();
   </script>
</div>
```

The `printEvaluator()` function draws a horizontal rule (`hr`) followed by the complete control panel of The Evaluator, including the codebase principal support for Mozilla. From this control panel, you can reference any document object supported by the browser's object model or global variable. You can even invoke functions located in other scripts of the page by entering them into the top text box. Whatever references are available to other scripts on the page are also available to The Evaluator, including references to other frames of a frameset and even other windows (provided a reference to the newly opened window has been preserved as a global variable, as recommended in Chapter 27).

If you are debugging a page on multiple browsers, you can switch between the browsers and enter property references into The Evaluator on each browser, making sure all browsers return the same values. Or, you can verify that a DOM object and property are available on all browsers under test. If you are working on W3C DOM–compatible browsers, invoke the `walkChildNodes()` function of The Evaluator to make sure that modifications your scripts make to the node tree are achieving the desired goals. Experiment with direct manipulation of the page's objects and node tree by invoking DOM methods as you watch the results on the page.

You should be aware of only two small cautions when you embed The Evaluator into the page. First, The Evaluator declares its own one-letter lowercase global variable names (`a` through `z`) for use in experiments. Your own code should therefore avoid one-letter global variables (but local variables, such as the `i` counter of a `for` loop, are fine, provided they are initialized inside a function with a `var` keyword). Second, while embedding The Evaluator at the bottom of the page should have the least impact on the rest of your HTML and scripts, any scripts that rely on the length of the `document.forms` array will end up including the form that is part of The Evaluator. You can always quickly turn off The Evaluator by commenting out the `printEvaluator()` statement in the bottom script to test your page on its own.

The embeddable Evaluator is without question the most valuable and frequently used debugging tool in our personal arsenal. It provides x-ray vision into the object model of the page at any resting point.

Emergency evaluation

Using The Evaluator assumes that you thought ahead of time that you want to view property values of a page. But what if you haven't yet embedded The Evaluator, and you encounter a state that you'd like to check out without disturbing the currently loaded page?

To the rescue comes the `javascript:` URL and the Location/Address box in your browser's toolbar. By invoking the `alert()` method through this URL, you can view the value of any property. For example, to find out the content of the cookie for the current document, enter the following into the Location/Address box in the browser:

```
javascript:alert(document.cookie)
```

Object methods or script functions can also be invoked this way, but you must be careful to prevent return values from causing the current page to be eliminated. If the method or function is known to return a value, you can display that value in an alert dialog box. The syntax for a function call is:

```
javascript:alert(myFunction("myParam1"))
```

And if you want to invoke a function that does not necessarily return a value, you should also protect the current page by using the void operator, as follows:

```
javascript:void myFunction("myParam1")
```

One more way to grab a "snapshot" of the current document tree, complete with all tags, is to use the following pseudo-URL in the Location/Address box:

```
javascript: "<textarea cols=120 rows=40>" +
     document.body.parentNode.innerHTML + "</textarea>"
```

When you execute this script, the current page goes away, replaced by a page containing a textarea filled with the current HTML state of the entire document (except for the `<html>` tag). If your scripts perform document tree modification, and you want to observe the current tree state, this command acts as a cross-browser x-ray into the document. You should consider making a bookmark out of the above URL, and keeping it handy.

A Simple Trace Utility

Single-stepping through running code with a JavaScript debugger is a valuable aid when you know where the problem is. But when the bug location eludes you, especially in a complex script, you may find it more efficient to follow a rapid trace of execution and viewing intermediate values along the way. The kinds of questions that this debugging technique addresses include:

- How many times is that loop executing?
- What are the values being retrieved each time through the loop?
- Why won't the while loop exit?
- Are comparison operators behaving as I'd planned in `if...else` constructions?
- What kind of value is a function returning?

A bonus feature of the embeddable Evaluator is a simple trace utility that lets you control where in your running code values can be recorded for viewing after the scripts run. The resulting report you get after running your script can answer questions like these and many more.

The trace() function

Listing 48-1 shows the `trace()` function that is built into the `evaluator.js` library file. By embedding the Evaluator into your page under construction, you can invoke the `trace()` function wherever you want to capture an interim value.

LISTING 48-1

trace() function

```
function trace(flag, label, value)
{
   if (flag)
   {
      var msg = "";
         if (trace.caller)
         {
            var funcName = trace.caller.toString();
            funcName = funcName.substring(9, funcName.indexOf(")") + 1);
            msg += "In " + funcName + ": ";
         }
      msg += label + "=" + value + "\n";
      document.forms["ev_evaluator"].ev_output.value += msg;
   }
}
```

The `trace()` function takes three parameters. The first, `flag`, is a Boolean value that determines whether the trace should proceed. (We show you a shortcut for setting this flag later.) The second parameter is a string used as a plain-language way for you to identify the value being traced. The value to be displayed is passed as the third parameter. Virtually any type of value or expression can be passed as the third parameter — which is precisely what you want in a debugging aid.

Only if the flag parameter is `true` does the bulk of the `trace()` function execute. The first task is to extract the name of the function from which the `trace()` function was called. By retrieving the rarely used `caller` property of a function, the script grabs a string copy of the entire function that has just called `trace()`. A quick extraction of a substring from the first line yields the name of the function. That information becomes part of the message text that records each trace. The message identifies the calling function followed by a colon; after that comes the label text passed as the second parameter, plus an equal sign and the value parameter. The format of the output message adheres to the following syntax:

```
In <funcName>: <label>=<value>
```

The results of the trace — one line of output per invocation — are appended to the Results textarea in The Evaluator. It's a good idea to clear the textarea before running a script that has calls to `trace()`, so that you can get a clean listing.

Preparing documents for trace.js

As you build your document and its scripts, you need to decide how granular you want tracing to be: global or function-by-function. This decision affects at what level you place the Boolean "switch" that turns tracing on and off.

You can place one such switch as the first statement in the first script of the page. For example, specify a clearly named variable and assign either false or zero to it so that its initial setting is off:

```
var TRACE = 0;
```

To turn debugging on at a later time, simply edit the value assigned to `TRACE` from zero to one:

```
var TRACE = 1;
```

Be sure to reload the page each time you edit this global value.

Alternatively, you can define a local `TRACE` Boolean variable in each function for which you intend to employ tracing. One advantage of using function-specific tracing is that the list of items to appear in the Results textarea will be limited to those of immediate interest to you, rather than all tracing calls throughout the document. You can turn each function's tracing facility on and off by editing the values assigned to the local `TRACE` variables.

Invoking trace()

All that's left now is to insert the one-line calls to `trace()`, according to the following syntax:

```
trace(TRACE,"label",value);
```

By passing the current value of `TRACE` as a parameter, you let the library function handle the decision to accumulate and display the trace. The impact on your running code is kept to a one-line statement that is easy to remember. To demonstrate how to make the calls to `trace()`, Listing 48-2 shows a pair of related functions that convert a time in milliseconds to the string format "hh:mm". To help verify that values are being massaged correctly, the script inserts a few calls to `trace()`.

LISTING 48-2

Calling trace()

```
function timeToString(input)
{
   var TRACE = 1;
   trace(TRACE,"input",input);
   var rawTime = new Date(eval(input));
   trace(TRACE,"rawTime",rawTime);
   var hrs = twoDigitString(rawTime.getHours());
   var mins = twoDigitString(rawTime.getMinutes());
   trace(TRACE,"result", hrs + ":" + mins);
   return hrs + ":" + mins;
}

function twoDigitString(val)
```

continued

LISTING 48-2 *(continued)*

```
{
   var TRACE = 1;
   trace(TRACE,"val",val);
   return (val < 10) ? "0" + val : "" + val;
}
```

After running the script, the Results box in The Evaluator shows the following trace:

```
In  timeToString(input): input=976767964655
In  timeToString(input): rawTime=Wed Dec 13 20:26:04 GMT-0800 2006
In  twoDigitString(val): val=20
In  twoDigitString(val): val=26
In  timeToString(input): result=20:26
```

Having the name of the function in the trace is helpful in cases in which you might justifiably reuse variable names (for example, `i` loop counters). You can also see more clearly when one function in your script calls another.

One of the most valuable applications of the `trace()` function comes when your scripts accumulate HTML that gets written to other windows or frames, or replaces HTML segments of the current page. Because the source view may not display the precise HTML that you assembled, you can output it via the `trace()` function to the Results box. From there, you can copy the HTML and paste it into a fresh document to test in the browser by itself. You can also verify that the HTML content is being formatted the way that you want it.

Browser Crashes

Each new browser generation is less crash-prone than its predecessor, which is obviously good news for everyone. It seems that most crashes, if they occur, do so as the page loads. This can be the result of some ill-advised HTML, or something happening as the result of script statements that either run immediately as the page loads or in response to the `onload` event handler.

Finding the root of a crash problem is perhaps more time consuming because you must relaunch the browser each time (and in some cases, even reboot your computer). But the basic tactics are the same. Reduce the page's content to the barest minimum HTML by commenting out both scripts and all but `head` and `body` tags. Then begin enabling small chunks to test reloading of the page. Be suspicious of `meta` tags inserted by authoring tools. Their removal can sometimes clear up all crash problems. Eventually, you will add something into the mix that will cause the crash. It means that you are close to finding the culprit.

Preventing Problems

Even with the help of authoring and debugging tools, you probably want to avoid errors in the first place. We offer a number of suggestions that can help in this regard.

Getting structure right

Early problems in developing a page with scripts tend to be structural: knowing that your objects are displayed correctly on the page; making sure that your `<script>` tags are complete; and completing brace, parenthesis, and quoted pairs. We start writing our page by first getting down the HTML parts — including all form definitions. Because so much of a scripted page tends to rely on the placement and naming of interface elements, you will find it much easier to work with these items after you lay them out on the page. At that point, you can start filling in the JavaScript.

When you begin defining a function, repeat loop, or `if` construction, fill out the entire structure before entering any details. For example, when we define a function named `verifyData()`, we enter the entire structure for it:

```
function verifyData()
{

}
```

We leave a blank line between the beginning of the function and the closing brace in anticipation of entering at least one line of code.

After we decide on a parameter to be passed and assign a variable to it, we may want to insert an `if` construction. Again, we fill in the basic structure:

```
function verifyData(form)
{
   if (form.checkbox.checked)
   {

   }
}
```

This method automatically pushes the closing brace of the function lower, which is what we want — putting it securely at the end of the function where it belongs. It also ensures that we line up the closing brace of the `if` statement with that grouping. Additional statements in the `if` construction push down the two closing braces.

If you don't like typing, or don't trust yourself to maintain this kind of discipline when you're in a hurry to test an idea, you should prepare a separate document that has templates for the

common constructions: `<script>` tags, function, `if`, `if...else`, `for` loop, `while` loop, and conditional expressions. Then, if your editor and operating system support it, drag and drop the necessary segments into your working script.

Build incrementally

The worst development tactic you can follow is to write tons of code before trying any of it. Error messages may point to so many lines away from the source of the problem that it could take hours to find the true source of difficulty. The save-switch-reload sequence is not painful, so the better strategy is to try your code every time you have written a complete thought — or even enough to test an intermediate result in an alert dialog box — to make sure that you're on the right track.

Test expression evaluation

Especially while you are learning the ins and outs of JavaScript, you may feel unsure about the results that a particular string, math, or date method yields on a value. The longer your scripted document gets, the more difficult it will be to test the evaluation of a statement. You're better off trying the expression in a more controlled, isolated environment, such as The Evaluator. By doing this kind of testing in the browser, you save a great deal of time over experimenting by going back and forth between the source document and the browser.

Build function workbenches

A similar situation exists for building and testing functions, especially generalizable ones. Rather than test a function inside a complex scripted document, drop it into a skeletal document that contains the minimum number of user interface elements that you need to test the function. This task gets difficult when the function is closely tied to numerous objects in the real document, but it works wonders for making you think about generalizing functions for possible use in the future. Display the output of the function in a text or textarea object, or include it in an alert dialog box.

Testing Your Masterpiece

If your background strictly involves designing HTML pages, you probably think of testing as determining your user's ability to navigate successfully around your site. But a JavaScript-enhanced page — especially if the user enters input into fields or implements Dynamic HTML techniques — requires a substantially greater amount of testing before you unleash it to the online masses.

A large part of good programming is anticipating what a user can do at any point, and then being sure that your code covers that eventuality. With multiframe windows, for example, you need to see how unexpected reloading of a document affects the relationships between all the frames — especially if they depend on each other. Users will be able to click Reload at any time

or suspend document loading in the middle of a download from the server. How do these activities affect your scripting? Do they cause script errors based on your current script organization?

The minute you enable a user to type an entry into a form, you also invite the user to enter the wrong kind of information into that form. If your script expects only a numeric value from a field, and the user (accidentally or intentionally) types a letter, is your script ready to handle that "bad" data? Or no data? Or a negative floating-point number?

Just because you, as author of the page, know the "proper" sequence to follow and the "right" kind of data to enter into forms, your users will not necessarily follow your instructions. In days gone by, such mistakes were relegated to "user error." Today, with an increasingly consumer-oriented web audience, any such faults rest solely on the programmer — you.

If we sound as though we're trying to scare you, we have succeeded. We were serious in the early chapters of this book when we said that writing JavaScript is *programming*. Users of your pages are expecting the same polish and smooth operation (no script errors and certainly no crashes) from your site as from the most professional software publisher on the planet. Don't let them or yourself down. Test your pages extensively on as many browsers and as many operating systems as you can, and with as wide an audience as possible, before putting the pages on the server for all to see.

CHAPTER 49

Security and Netscape Signed Scripts

IN THIS CHAPTER

Exploring browser security policies

Applying JavaScript to Navigator security mechanisms

Using Netscape signed scripts

The paranoia levels about potential threats to security and privacy on the Internet are at an all-time high. As more people rely on email and web site content for their daily lives and transactions, the fears will only increase for the foreseeable future (an indeterminate number of web weeks). As a jokester might say, however, "I may be paranoid, but how do I know someone really isn't out to get me?" The answer to that question is that you don't know, and such a person may be out there.

But web software developers are doing their darnedest to put up roadblocks to those persons out to get you — hence, the many levels of security that pervade browsers. Unfortunately, these roadblocks also get in the way of scripters who have completely honest intentions. Designing a web site around these barriers is one of the greatest challenges that many scripters face.

Battening Down the Hatches

When Navigator 2 first shipped to the world, way back in the previous century (February 1996), it was the first browser released to include support for Java applets and scripting — two entirely different, but often confused, technologies. It didn't take long for clever programmers in the Internet community to find the ways in which one or the other technology provided inadvertent access to client computer information (such as reading file directories) and web surfer activities (such as histories of where you've been on the Net and even the passwords you may have entered to access secure sites).

JavaScript, in particular, was the avenue that many of these programmers used to steal such information from web site visitors' browsers. The sad part is that the same features that provide the access to the information were

intentionally made a part of the initial language to aid scripters who would put those features to beneficial use in controlled environments, such as intranets. But out in the Wild Wide Web, a scripter could capture a visitor's email address by having the site's home page surreptitiously send a message on behalf of the visitor to the site's author without the visitor even knowing it. It was a spammer's delight.

Word of security breaches of this magnitude not only circulated throughout the Internet, but also reached both the trade and mainstream press. As if the security issues weren't bad enough on their own, the public relations nightmare compounded the sense of urgency to fix the problem. To that end, Netscape released two revised editions of Navigator 2. The final release of that generation of browser, Navigator 2.02, took care of the scriptable security issues by turning off some of the scripted capabilities that had been put into the original 2.0 version: no more capturing visitors' browser histories; no more local file directory listings; no more silent email. Users could even turn off JavaScript support entirely if they so desired.

The bottom line on security is that scripts are prevented from performing automated processes that invade the private property of a web author's page or a client's browser. Thus, any action that may be suspect, such as sending an email message, requires an explicit action on the part of the user — clicking a Submit button, in this case — to carry it out. Security restrictions must also prevent a web site from tracking your activity beyond the boundaries of that web site.

When Worlds Collide

If a script tries to do something that is not allowed or is a potential personal security breach, the browser reports the situation to the user. For example, browsers display a warning to the user when a script in a subwindow tries to close the main window.

Another security error message often confuses scripters who don't understand the possible privacy invasions that can accrue from one window or frame having access to the URL information in another window or frame. In WinIE, for example, an ominous error message — "Access denied" — warns users of an attempt to access URL information from another frame if that URL is from a different web site.

Despite the fact that a scripted web site may have loaded the foreign URL into the other frame, the security restrictions guard against unscrupulous usage of the ability to snoop in other windows and frames.

More recently, browsers have started cracking down on web pages that open new browser windows. Most browsers now offer this feature as an option that allows the user to permit designated web sites to open pop-up windows, while preventing pop-ups in all others.

The Java Sandbox

Much of the security model for JavaScript is similar to that originally defined for Java applets. Applets had a potentially dangerous facility of executing Java code on the client machine. That is a far cry from the original deployment of the World Wide Web as a read-only publishing

medium on the Internet. Here were mini-programs downloaded into a client computer that, if unchecked, could have the same access to the system as a local software program.

Access of this type would clearly be unacceptable. Imagine the dismay caused by someone clicking a link that said Free Money, only to have the linked page download an applet that read or damaged local disk files, unbeknownst to the user. In anticipation of pranksters, the designers of Java and the Java virtual machine built in a number of safeguards to prevent applets from gaining access to local machines. This mechanism is collectively referred to as the *sandbox*, a restricted area in which applets can operate. Applets cannot extend their reach outside of the sandbox to access local file systems and many sensitive system preferences. Any applet runs only while its containing page is still loaded in the browser. When the page goes away, so does the applet, without being saved to the local disk cache.

JavaScript adopted similar restrictions. The language provided no read or write access to local files beyond the highly regulated cookie file. Moreover, because JavaScript works more closely with the browser and its documents than applets typically do, the language had to build in extra restrictions to prevent browser-specific privacy invasions. For example, it was not possible for a script in one window to monitor the user's activity in another window, including the URL of the other window, if the page didn't come from the same server as the first window. Sometimes the restrictions on the JavaScript side are even more severe than in Java. For example, although a Java applet is permitted to access the network anytime after the applet is loaded, an applet is prevented from reaching out to the Net if the trigger for that transaction comes from JavaScript (see Chapter 47, "Scripting Java Applets and Plug-Ins"). Only partial workarounds are available.

Neither the Java nor the JavaScript security blankets were fully bug-free at the outset. Some holes were uncovered by the languages' creators and others in the community. To their credit, Sun, Netscape, and Microsoft have been quick to plug any discovered holes. Although the plugs don't necessarily fix existing copies of unsecure browsers out there, it does mean that a bad guy can't count on every browser to offer the same security hole for exploitation. That generally makes the effort not worth the bother.

Security Policies

Netscape originally defined security mechanisms under the term *policies*. This usage of the word mirrors that of institutions and governments: A policy defines the way potentially insecure or invasive requests are handled by the browser or scripting language. Modern Mozilla-based browsers include two different security policies: *same origin* and *signed script* policies. The same origin policy dates back to Navigator 2, although some additional rules have been added to that policy as the browser has matured. The signed script policy started with NN4 and utilizes the state of the art in cryptographic signatures of executable code inside a browser, whether that code is a plug-in, a Java applet, or a JavaScript script. Signed script facilities allow scripts to have a wider range of control over the browser's interior working parts, provided the user grants permission for such activity (more about this later in the chapter).

By and large, the same origin policy is in force inside Internet Explorer, Safari, and other scriptable browsers. Precise details may not match up with Mozilla browsers one-for-one, but the most common features are identical. The signed script policy is a different story, with respect

to Internet Explorer. Although Microsoft offers digital signatures for some items that may be embedded within an HTML page (such as ActiveX controls and other components), scripts that are in an HTML page's source code or linked in as a `.js` library cannot be signed for Internet Explorer. Although everything you read in this chapter about signed scripts applies only to Mozilla, you should find the next couple of sections informative, even if you develop solely for Internet Explorer or only cross-platform JavaScript code.

The Same Origin Policy

The "origin" in *same origin policy* refers to the protocol and domain of a source document. If all the source files currently loaded in the browser come from the same server and domain, scripts in any one part of the environment can poke around the other documents. Restrictions come into play when the script doing the poking and the document being poked come from different origins. The potential security and privacy breaches this kind of access can cause put it out of bounds within the same origin policy.

An origin is not the complete URL of a document. Consider the two popular URLs for Mozilla's web sites:

```
http://www.mozilla.org
http://developer.mozilla.org
```

The protocol for both sites is `http:`. Both sites also share the same domain name: `mozilla.org`. But the sites run on two different servers: `www` and `developer` (at least this is how the sites appear to browsers accessing them; the physical server arrangement may be quite different).

If a frameset contains documents from the same server at `mozilla.org`, and all frames are using the same protocol, they have the same origin. Completely open and free access to information, such as `location` object properties, is available to scripts in any frame's document. But if one of those frames contains a document from the other server, their origins don't match. A script in a document from one server would display an "access disallowed" or "permission denied" error message if it tried to get the `location` property of that other document.

A similar problem occurs if you were creating a web-based shopping service that displays the product catalog in one window and the order form from a secure server in another window. The order form, whose protocol might be `https:`, would not be granted access to the `location` object properties in a catalog page whose protocol is `http:`, even though both share the same server and domain name.

Setting the document.domain

When both pages in an origin-protected transaction are from the same domain (but different servers or protocols), you can instruct JavaScript to set the `document.domain` properties of both pages to the domain that they share. When this property is set to that domain, the pages are treated as if from the same origin. Making this adjustment is safe, because JavaScript doesn't

allow setting the `document.domain` property to any domain other than the origin of the document making the setting. See the `document.domain` property entry in Chapter 29, "The Document and Body Objects," for further details.

Origin checks

Scattered throughout the language reference chapters are notes about items that undergo what you now know to be origin checks. For the sake of convenience, we list them all here to help you get a better feeling for the kind of information that is protected. The general rule is that any object property or method that exposes a local file in a user's system, or can trace web surfing activity in another window or frame, undergoes an origin check. Failure to satisfy the same origin rule yields an "access disallowed" or "permission denied" error message on the client's machine.

Window object checks

The document object models of windows and frames that don't share the same origin are not available to each other. Each separate origin window or frame is its own little world that has very little ability to communicate with another window or frame. Internet Explorer sometimes takes this to the extreme, causing problems between a main window and a subwindow whose content is entirely dynamically generated from the main window's scripts.

Location object checks

All `location` object properties are restricted to same origin access. Of all same origin policy restrictions, this one seems to interfere the most with well-meaning page authors' plans, preventing them from providing a frame for users to navigate around the Web. Such access, however, would allow spying on your users.

Document object checks

A `document` object's properties are by necessity loaded with information about the content of that document. Just about every property other than the ones that specify color properties are off-limits if the origin of the target document is different from the one making the request:

```
anchors[]    lastModified
applets[]    length
cookie       links[]
domain       referrer
embeds       title
forms[]      URL
```

In addition, no normally modifiable `document` property can be modified if the origin check fails. This, of course, does not prevent you from using `document.write()` to write an entirely new page of content to the frame to replace a document from a different origin. But in modern browsers, scripts from one origin won't be able to modify (or even copy) partial content from a frame whose content comes from another origin.

Form object checks

Form data is generally protected by the restriction to a document's `forms[]` array. But should a script in another window or frame also know the name of the form, that, too, won't enable access unless both documents come from the same origin.

Applet object checks

The same goes for named Java applets. A script cannot retrieve information about the class filename unless both documents are from the same origin (although the applet can be from anywhere).

Access from a Java applet to JavaScript is not an avenue to other windows and frames from other origins. Any calls from the applet to the objects and protected properties described here undergo origin checks when those objects are in other frames and windows. The applet assumes the origin of the document that contains the applet, not the applet codebase.

Image object checks

Although image objects are accessible from other origins, their `src` and `lowsrc` properties are not. These URLs could reveal some or all of the URL info about the document containing them.

Linked script library checks

To prevent a network-based script from hijacking a local script library file, Mozilla prevents a page from loading a `file:` protocol library in the `src` attribute of a `<script>` tag unless the main document also comes from a `file:` protocol source. If you are beginning to think that security engineers are a suspicious and paranoid lot, you are starting to get the idea. It's not easy to curb potential abuses of Bad Guys in a networked environment initially established for openness and free exchange of information among trusted individuals.

The Netscape Signed Script Policy

Just as there are excellent reasons to keep web pages from poking around your computer and browser, there are equally good reasons to allow such access to a web site that you trust not to be a Bad Guy. To permit trusted access to the client machine and browser, Sun Microsystems and Netscape (in cooperation with other sources) developed a way for web application authors to identify themselves officially as authors of the pages, and to request permission of the user to access well-defined parts of the computer system and browser.

The technology is called object *signing*. In broad terms, object signing means that an author can electronically lock down a chunk of computer code (whether it be a Java applet, a plug-in, or a script) with the electronic equivalent of a wax seal stamped by the author's signet ring. At the receiving end, a user is informed that a sealed chunk of code is requesting some normally-protected access to the computer or browser. The user can examine the electronic seal to see who authored the code and the nature of access being requested. If the user trusts the author not to be a Bad Guy, the user grants permission for that code to execute; otherwise

the code does not run at all. Additional checks take place before the code actually runs. That electronic seal contains an encrypted, reduced representation of the code as it was locked by the author. If the encrypted representation cannot be re-created at the client end (it takes only a fraction of a second to check), it means the code has been modified in transit, and it will not run.

In truth, nothing prevents an author from being a Bad Guy, including someone you may normally trust. The point of the object signing system, however, is that a trail leads back to the Bad Guy. An author cannot use this technology to sneak into your computer or browser without your explicit knowledge and permission. Think of it like this: just because you willingly let someone into your home doesn't guarantee they won't steal your television, but the fact that you were able to decide whether or not to let them in at the front door goes a long way toward preventing a theft. And should it still occur, you know who the culprit is.

Since the key principle behind object signing is simply to verify the source of a script, for better or for worse, it should come as no surprise that object signing makes more sense within the realm of intranets as opposed to the Internet as a whole. That's not to say you can't sign scripts for the whole world to see, but don't just assume that because you know you're reputable, everyone else will assume the same. Within an intranet, the assumption that your own company can be trusted is a much safer pill to swallow.

Signed objects and scripts

A special version of the signed object technology is the one that lets scripts be locked down by their author and electronically signed. Virtually any kind of script in a document can be signed: a linked `.js` library, scripts in the document, event handlers, and JavaScript entities. As described later in this chapter, you must prepare your scripts for being signed, and then run the entire page through a special tool that attaches your electronic signature to the scripts within that page.

What you get with signed scripts

If you sign your scripts, and the user grants your page permission to do its job, signed scripts open up your application to a long list of capabilities, some of which border on acting like genuine local applications. A huge number of properties and actions are exposed to authors of signed scripts.

The most obvious power you get with signed scripts is freedom from the restrictions of the same origin policy. All object properties and methods that perform origin checks for access in other frames and windows become available to your scripts without any special interaction with the user, beyond the dialog box that requests the one-time permission for the page.

Some operations that normally display warnings about impending actions — closing the main browser window under script control, for example — lose those warning dialog boxes if the user grants the appropriate permission to a signed script. Object properties considered private information, such as individual URLs stored in the history object and browser preferences, are opened up, including the possibility of altering browser preferences. Existing windows can have their chrome elements hidden. New windows can be set to be always raised or lowered, sized to

very small sizes, or positioned offscreen. The `dragdrop` event of a window reveals its URL. All these are powerful points of access, provided the user grants permission.

Again, however, we emphasize that these capabilities are accessible through the Netscape signed script policy only, which is available in all Mozilla-based browsers. Internet Explorer, at least through Version 8, does not support Netscape's (Mozilla's) signed script policy.

The Digital Certificate

Before you can sign a script or other object, you must apply for a digital certificate. A digital certificate (also called a digital `id`) is a small piece of software that is downloaded and bound to the developer's browser on a particular computer. Each downloaded digital certificate is accessible from the Certificate Manager in Mozilla browsers, which is found in Firefox, for example, under the Security tab of the Advanced Options window. Click the View Certificates button to open the Certificate Manager. The Certificate Manager displays a list of the certificates currently available on your system. If you have not yet applied for a certificate, the list is empty. When you sign a page with the certificate, information about the certificate is included in the file generated by the signing tool.

Possession of a certificate makes you what is known as a *principal*. If a user loads a page that has signed "stuff" in it, a security alert advises the user that a web site is requesting enhanced privileges.

Certificates are issued by organizations established as certificate authorities. A certificate authority (known as a *CA* for short), or a certificate server authorized by a CA, registers applicants and issues certificates to individuals and software developers. When you register for a certificate, the CA queries you for identification information, which it verifies as best it can. The certificate that is issued to you identifies you as the holder of the certificate. Under the Authorities tab of the Security Manager window are the certificate authorities that were loaded into the browser when you installed it. These are organizations that issue certificates. The CA of the organization that issues your certificate must be listed for you to sign scripts.

How to get a certificate

You must visit a certificate vendor to obtain your certificate. The cost ranges from about $100 to many hundreds of dollars, depending on the vendor and the type of certificate you want to obtain. Vendors that are aware of the needs of Netscape object signing include Thawte Digital Certificate Services (`www.thawte.com`), VeriSign (`www.verisign.com`), and Instant SSL (`www.instantssl.com`). These certificate authorities offer certificates expressly for developers performing Netscape object signing. Make sure the certificate is specifically designed for signing objects.

Because one of the foundations of a certificate is the identity of the certificate owner, registration requires submitting documentation that proves the identity of your organization. Each CA has different requirements, so check the latest information at the CA's enrollment web site. After the

CA processes your application, the company sends you an email message with a code number to pick up your certificate at the CA's web site. The act of picking up the certificate is actually downloading the certificate into your browser. Therefore, be sure you are using a Mozilla-based browser on the computer with which you will use to sign your pages.

Activating the codebase principal

If you want to try out the capabilities available to signed scripts from a server without purchasing a certificate (or without going through the signing process described later in this chapter during script development and debugging), you can set up Mozilla to accept what is called a *codebase* principal in place of a genuine certificate. A codebase principal means that the browser accepts the source file as a legitimate principal, although it contains no identification as to the owner or certificate.

Note

You can experiment with properties and methods that normally require signed scripts without modifying the codebase principal preferences or signing scripts with a certificate. Each time you activate the Privilege Manager (as The Evaluator does when you select the "Use NN Codebase Security" check box), a security alert lets you elect to continue with the untrusted access to your browser or system. Do not select the box in the alert dialog box that remembers this choice, because it could leave your browser and system vulnerable to other unsigned scripts without your knowledge. ■

Depending on which Mozilla-based browser you are running, if you set up your browser for codebase principals, you may not be able to verify a certificate that is presented to you when accessing someone else's web site — even if it is a valid cryptographic certificate. Therefore, even though secure requests won't slip past you silently, your browser won't necessarily have the protective shield it normally does to identify certificate holders beyond the URL of the code. Enable codebase principals only on a browser installation that doesn't venture beyond your development environment. As an example, to activate codebase principals for Firefox browsers, follow these steps:

1. Navigate to the URL `about:config` in Firefox.
2. Scroll down the list of preferences until you find the one named `signed.applets.codebase_principal_support`.
3. Double-click the `signed.applets.codebase_principal_support` preference to change its value from `false` to `true`.

To deactivate codebase principals, change the `true` setting of the `signed.applets.codebase_principal_support` preference back to `false`. This preferences setting does not affect your ability to sign scripts with your certificate as described in the rest of this chapter.

Signing Scripts

The process of signing scripts entails some new concepts for even experienced JavaScript authors. You must use a separate signing tool program. You must also prepare the page that bears scripts so that the tool and the object signing facilities of the browser can do their jobs.

Signing tool

Download the latest version of Mozilla's SignTool from links you find at `http://www.mozilla.org/projects/security/pki/nss/tools/`. This tool includes a utility program called a `JAR` Packager. A `JAR` file is a special kind of zipped file collection that has been designed to work with the Navigator security infrastructure. The letters of the name stand for Java ARchive, which is a file format standard developed primarily by Sun Microsystems, in cooperation with Netscape and others.

Note

Unfortunately, the SignTool application is currently distributed as a suite of security applications that are available in source code form only. This means that you'll have to create the binary executables for the tools yourself by building them after downloading the source code. The source is freely available for download at `http://www.mozilla.org/projects/security/pki/nss/tools/`**.** ■

A `JAR` file's extension is `.jar`, and when it contains signed script information, it holds at least one file, known as the *manifest*, or list of items zipped together in the file. Among the items in the manifest is certificate information and data (a hash value code) about the content of the signed items at the instant they were signed. In the case of a single page containing signed scripts, the `JAR` file contains only the certificate and hash values of the signed scripts within the document. If the document links in an external `.js` script library file, that library file is also packaged in the `JAR` file. Thus, a page with signed scripts occupies space on the server for the HTML file and its companion `JAR` file.

The SignTool program combines the `JAR` Packager with the script signing functions. Follow links on the SignTool download page to the latest instructions on packaging and signing your finished files from the command line (there is no GUI for this tool). But before you reach that point, you need to compose your pages in such a way that the security mechanism can protect your scripts.

Preparing scripts for signing

In NN4, signifying which items in a page are script items that require signing was an important consideration. Modern Mozilla-based browsers require you to sign an entire page, not just the scripts within the page. Of course, any externally referenced scripts must still be signed if they need to be granted privileges within the context of the page. The good news is that the SignTool application makes it relatively painless to sign all the files associated with a page.

Accessing signed scripts

Once you've signed a page and its relevant scripts, you'll end up with a JAR file, or files, containing all the resources. To access a published page that lives within a signed JAR file, you must use the following syntax:

```
jar:http://www.mydomain.com/secure/topsecret.jar!/index.html
```

In this example, the domain is `www.mydomain.com`, the page is `index.html`, and it is located in the JAR file named `topsecret.jar` within the folder `secure`. Notice that the URL is preceded by `jar:`, which lets the browser know that it is dealing with a JAR archive.

Seeing as how this is somewhat of an unusual URL to require users to enter, especially as the front end on a secure web site, you may opt to display a redirection page of some sort that automatically redirects to the secure page. Here's a <meta> tag that accomplishes this task for the preceding example URL:

```
<meta http-equiv="refresh" content="0;
     url=jar:http://www.mydomain.com/secure/topsecret.jar!/index.html" />
```

By specifying 0 as the amount of time before the redirection, the net effect is an instant redirection to the secure page that contains signed JavaScript code.

Editing and moving signed scripts

The nature of the script signing process means that even the slightest modification you make to a signed script source code requires re-signing the page. For this reason, enabling codebase principals while you create and debug your early code is a convenient alternative.

The rigid link between the hash value of a script element at both the signing and visitor loading times means that you must exercise care when shifting an HTML file that contains signed script elements between servers of differing operating systems. Windows, UNIX, and Macintosh have different ways of treating carriage returns. If you change the representation of an HTML source file when you move the source from, say, a Windows machine to a UNIX server, the signature may no longer work. However, if you perform a purely binary transfer of the HTML files, every byte is the same, and the signature should work. This operating system–specific text representation affects only how files are stored on servers, not how various client platforms interpret the source code.

Accessing Protected Properties and Methods

For the browser to allow access to protected properties or methods, it must have its privileges enabled. Only the user can grant permission to enable privileges, but it is up to your code to make the request.

Gaining privileges

Mozilla-based browsers come with some Java classes that allow signed scripts and other signed objects to display the privilege request alert windows, and then turn on the privileges if the user clicks the OK or Grant button. A lot of these classes show up in the netscape.security package, but scripters only work directly with one class and three of its methods:

```
netscape.security.PrivilegeManager.enablePrivilege(["targetName"])
netscape.security.PrivilegeManager.revertPrivilege(["targetName"])
netscape.security.PrivilegeManager.disablePrivilege(["targetName"])
```

The `enablePrivilege()` method is the one that displays the security alert for the user. In its original inception in NN4, the specific target, named as a parameter, influenced the details of the security alert message; for Mozilla browsers, the security alert is generic (and far less intimidating).

If the user grants the privilege, script processing continues with the next statement. But if the user denies access, then processing stops, and the `PrivilegeManager` class throws a Java exception that gets displayed as a JavaScript error message. Later in this chapter, we show you how to gracefully handle the user's denial of access.

Enabling a privilege in JavaScript is generally not as risky as enabling a Java applet. The latter can be more easily hijacked by an alien class to piggyback on the trusted applet's privileges. Even though the likelihood of such activity taking place in JavaScript is very low, turning privileges off after the statement that requires privileges is always a good idea. Use the `revertPrivilege()` method to temporarily turn off the privilege; another statement that enables the same privilege target will go right ahead without asking the user again. Disable privileges only when the script requiring privileged access won't be run again until the page reloads.

Specifying a target

Rather than opening blanket access to all protected capabilities in one blow, the Netscape security model defines narrow capabilities that are opened up when privileges are granted. Each set of capabilities is called a target. Netscape defines dozens of different targets, but not all of them are needed to access the kinds of methods and properties available to JavaScript. You will likely confine your targets to the nine discussed here.

Because NN4's security alerts provided detail (at times excruciating) about the nature of the privilege being requested by the web site, targets had various risk levels and categories. These concerns are less of an issue in Moz1+, but they are provided here for your more complete understanding of the mechanisms beneath the Privilege Manager.

Each target has associated with it a risk level (low, medium, or high) and two plain-language descriptions about the kinds of actions the target exposes to code. All the targets related to scripted access are medium or high risk, because they tend to open up local hard disk files and browser settings.

Netscape produced two categories of targets: primitive and macro. A *primitive target* is the most limited target type. It usually confines itself to either the reading or writing of a particular kind of data, such as a local file or browser preference. A *macro target* usually combines two or more primitive targets into a single target to simplify the user experience when your scripts require multiple kinds of access. For example, if your script must both read and write a local file, it could request privileges for each direction, but the user would be presented with a quick succession of two similar-looking security dialog boxes. Instead, you can use a macro target that combines both reading and writing into the privilege. The user sees one security dialog box, which explains that the request is for both read and write access to the local hard disk.

Likely targets for scripted access include a combination of primitive and macro targets. Table 49-1 shows the most common script-related targets and the information that appears in the security dialog box.

TABLE 49-1

Scripting-Related Privilege Targets

Target Name	Risk	Short Description	Long Description
`UniversalBrowserAcess`	High	Reading or modifying browser data	Reading or modifying browser data that may be considered `BrowserAccess` private, such as a list of websites visited or the contents of web forms you may have filled in. Modifications may also include creating windows that look like they belong to another program or positioning windows anywhere on the screen.
`UniversalBrowserRead`	Medium	Reading browser data	Access to browser data that may be considered `BrowserRead` private, such as a list of web sites visited or the contents of web page forms you may have filled in.
`UniversalBrowserWrite`	High	Modifying the browser	Modifying the browser in a potentially dangerous `BrowserWrite` way, such as creating windows that may look like they belong to another program or positioning windows anywhere on the screen.
`UniversalFileAccess`	High	Reading, modifying, or deleting any of your files	This form of access is typically required by a `FileAccess` program such as a word processor or a debugger that needs to create, read, modify, or delete files on hard disks or other storage media connected to your computer.
`UniversalFileRead`	High	Reading files stored in your computer	Reading any files stored on hard disks or other `FileRead` storage media connected to your computer.
`UniversalFileWrite`	High	Modifying files stored in your computer	Modifying any files stored on hard disks or other `FileWrite` storage media connected to your computer.
`UniversalPreferencesRead`	Medium	Reading preferences settings	Access to read the current settings of your `PreferencesRead` preferences.
`UniversalPreferencesWrite`	High	Modifying preferences settings	Modifying the current settings of your `PreferencesWrite` preferences.
`UniversalSendMail`	Medium	Sending SendMail email messages on your behalf	

For each call to `netscape.security.PrivilegeManager.enablePrivilege()`, you specify a single target name as a string, as in

```
netscape.security.PrivilegeManager.enablePrivilege("UniversalBrowserRead");
```

This specification allows you to enable, revert, and disable individual privileges as required in your script.

Adding Privileges to Scripts

The implementation of signed scripts in Mozilla browsers protects scripters from many of the potential hazards that Java applet and plug-in developers must watch for. The chance that a privilege enabled in a script can be hijacked by code from a Bad Guy is very small. Still, exercising safe practices in case you someday work with other kinds of signed objects is a good idea.

Keep the window small

Privilege safety is predicated on limiting exposure according to two techniques. The first technique is enabling only the level of privilege required for the protected access your scripts need. For example, if your script only needs to read a normally protected `document` object property, enable the `UniversalBrowserRead` target rather than the wider `UniversalBrowserAccess` target.

The second technique is to keep privileges enabled only as long as the scripts need them enabled. If a statement calls a function that invokes a protected property, enable the privilege for that property in the called function, not at the level of the calling statement. If a privilege is enabled inside a function, the browser automatically reverts the privilege at the end of the function. Even so, if the privilege isn't needed all the way to the end of the function, you should explicitly revert it after you are through with the privilege.

Think of the users

One other deployment concern focuses more on the user's experience with your signed page. You should recognize that the call to the Java `PrivilegeManager` class is an NPAPI (Netscape Plug-in Application Programming Interface) call from JavaScript. Because the Java virtual machine does not start up automatically when the browser does, a brief delay occurs the first time an NPAPI call is made in a session (the status bar displays "Starting Java ... "). Such a delay may interrupt the user flow through your page if, for example, a click of a button needs access to a privileged property. Therefore, consider gaining permission for protected access as the page loads. Execute an `enablePrivilege()` and `revertPrivilege()` method in the very beginning. If Java isn't yet loaded into the browser, the delay is added to the other loading delays for images and the rest of the page. Thereafter, when privileges are enabled again for a specific action, neither the security dialog box, nor the startup delay, get in the way for the user.

Also remember that users don't care for security dialog boxes interrupting their navigation. If your page utilizes a couple of related primitive targets, enable the macro target that encompasses those primitive targets at the outset. The user gets one security dialog box covering all potential actions in the page. Then, let your script enable and revert each primitive target as needed.

Examples

To demonstrate signed scripts in action, we show a couple of examples of scripted code that normally wouldn't be allowed based upon default browser security limitations. More specifically, you see a script that accesses the current home page setting for the browser, which is considered private and off-limits under normal circumstances. You also take a look at a page that allows the script to open an always-raised new window. No error checking occurs for the user's denial of privilege in this example. Therefore, if you experiment with this page (either with codebase principals turned on or signing them yourself), you will see the JavaScript error that displays the Java exception. Error detection is covered later in the chapter.

Accessing private browser information

Listing 49-1 contains a very simple HTML document that attempts to read the home page of the browser and display it in an alert box. Keep in mind that this is considered privileged information and is off-limits unless the script is signed and the user explicitly grants permission.

LISTING 49-1

Peeking at the Default Home Page

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Home Page Extractor</title>
      <script type="text/javascript">
         function showHome()
         {
            try
            {
netscape.security.PrivilegeManager.enablePrivilege("UniversalPreferencesRead");
               var home = navigator.preference("browser.startup.homepage");
               alert("The home page for this browser is currently set to "
               + home + ".");
            }
            catch (e)
            {
               alert("This browser's home page is none of your business.");
            }
         }
      </script>
   </head>
   <body onload="showHome();">
      <h1>Home Page Extractor</h1>
   </body>
</html>
```

In this example, a universal read privilege is first requested, which is ultimately where the check takes place to see if the secure action (reading the home page) will be allowed. If so, the home page is read via the `browser.startup.homepage` preference and then displayed in an alert box. If the action is declined, a message is displayed indicating the failure.

Accessing a protected window property

Listing 49-2 is a small document that contains one button. The button calls a function that opens a new window with the Moz-specific `modal` parameter turned on. Setting protected `window.open()` parameters in Moz1+ requires the `UniversalBrowserWrite` privilege target. Inside the function, the privilege is enabled only for the creation of the new window. For this simple example, we do not enable the privilege when the document loads.

LISTING 49-2

Creating an alwaysRaised Window

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Simple Signed Script</title>
      <script type="text/javascript">
         function newModalWindow()
         {
netscape.security.PrivilegeManager.enablePrivilege("UniversalBrowserWrite");
            var newWindow = window.open
           ("jsb49-03.html","","height=100,width=300,modal");
netscape.security.PrivilegeManager.disablePrivilege("UniversalBrowserWrite");
         }
      </script>
   </head>
   <body>
      <b>This button generates a modal-style new window.</b>
      <form>
         <input type="button" value="New Modal Window"
                onclick="newModalWindow()" />
      </form>
   </body>
</html>
```

Note

The property assignment event handling technique used in this example and throughout the chapter is a deliberate simplification to make the code more readable. It is generally better to use the more modern approach of binding events using the `addEventListener()` **(NN6+/Moz/W3C) or** `attachEvent()` **(IE5+) methods. A modern cross-browser event-handling technique is explained in detail in Chapter 32, "Event Objects."** ■

Listing 49-2 has two security-related script items: the `<script>` tag and the event handler for the button. These two components are automatically signed when you pass the entire page

through the SignTool application. Note that this example file is not signed, and therefore does not include a companion JAR archive on the CD-ROM.

Handling Privilege Manager Errors

Handling privilege manager errors is important because an error is generated any time a user denies access to advanced privileges. Although the error is not destructive in any way, and it appears only in the JavaScript/Error Console window, accounting for such factors is good coding practice.

You can use the native `try...catch` exception handling to respond to privilege errors, which means that the calls to the `enablePrivilege()` method of the `PrivilegeManager` class must be wrapped inside a `try` block. The function from Listing 49-2 is modified as follows:

```
function newModalWindow()
{
   try
   {
      netscape.security.PrivilegeManager.enablePrivilege(
         "UniversalBrowserWrite");
      var newWindow = window.open("jsb49-03.html","",
         "height=100,width=300,modal");
      netscape.security.PrivilegeManager.disablePrivilege(
         "UniversalBrowserWrite");
   }
   catch(err)
   {
      alert("You have elected not to grant privileges to this script.");
   }
}
```

Signed Script Miscellany

In this section, we list some of the more esoteric issues surrounding signed scripts. Three in particular are: how to allow unsigned scripts in other frames, windows, or layers to access signed scripts; how to make sure your signed scripts are not stolen and reused; and special notes about international text characters.

Exporting and importing signed scripts

JavaScript provides an escape route that lets you intentionally expose functions from signed scripts for access by unsigned pages. If such a function contains a trusted privilege without careful controls on how that privilege is used, a page that is not as well intentioned as yours could hijack the trust.

The command for exposing this function is export. The following example exports a function named `fileAccess()`:

```
export fileAccess;
```

A script in another window, frame, or layer can use the import command to bring that function into its own set of scripts:

```
import fileAccess;
```

Even though the function is now also a part of the second document, it executes within the context of the original document, whose signed script governs the privilege. For example, if you exported a function that did nothing but enable a file access privilege, a Bad Guy who studies your source code could write a page that imports that function into a page that now has unbridled file access.

If you want to share functions from signed scripts in unsigned pages loaded into your own frames or layers, avoid exporting functions that enable privileges. Other kinds of functions, if hijacked, can't do the same kind of damage as a privileged function can.

Locking down your signed pages

Speaking of hijacking scripts, it would normally be possible for someone to download your HTML and JAR archive files and copy them to another site. When a visitor comes to that other site and loads your copied page and JAR file, your signature is still attached to the scripts. Although this may sound good from a copyright point of view, you may not want your signature to appear as coming from someone else's web server. You can, however, employ a quick trick to ensure that your signed scripts work only on your server. By embedding the domain of the document in the code, you can branch execution so that scripts work only if the file comes from your server.

The following script segment demonstrates one way to employ this technique:

```
<script type="text/javascript">
   if (document.URL.match(/^http:\/\/www.myDomain.com\//))
   {
      // privileges statements execute only from my server
   }
</script>
```

Even though this branching code is visible in the HTML file, the hash value of your code is saved and signed in the archive. If someone modifies the HTML, the hash value that is recalculated when a visitor loads the page won't match the JAR file manifest, and the script signature fails.

International characters

Although international characters are fine for HTML content, they should not be used in signed scripts. The problem is that international characters are often converted to other character sets for display. This conversion invalidates the signature, because the signed and recalculated hash values don't match. Therefore, do not put international characters in any signable script item. If you must include such a character, you should escape it.

CHAPTER 50

Cross-Browser Dynamic HTML Issues

IN THIS CHAPTER

Introducing Dynamic HTML

The common denominator of DHTML functionality across browsers

Upgrading to modern compatibility techniques

Version 4 browsers — NN4 and IE4 — were the first to include World Wide Web technologies that gave page authors far more control over the display and interactive behavior of web page content. Lumped together under the heading of Dynamic HTML (DHTML), these technologies dramatically extended the simple formatting of standard HTML that page authors had used for years. These days, scripters and designers coming to web development for the first time take DHTML capabilities for granted; they are probably unaware that plain ol' HTML is little more than a specification to assign context to static text and images on a page.

A lot of what the user gets with DHTML had previously been accomplished only through Java applets and plug-ins, such as early versions of Shockwave (pre-Flash). Not that DHTML eliminates these technologies from the web author's arsenal (DHTML doesn't do sound or video, for example), but because DHTML can accomplish much more of what authors look for in assembling page content and layout, without the long downloads of applets or plug-in content, it becomes an attractive way for nonprogrammers to spice up web applications. At the same time, however, some recent technologies, such as Ajax, have brought legions of experienced programmers onto the scene to create some amazing application-like web pages for mapping, email, and much more.

What Is DHTML?

You can find practically as many definitions of Dynamic HTML as there are people to ask. Our definition covers a broad range, because DHTML is not really any one "thing." Instead, it is an amalgam of several technologies, each of which has a standards effort in varying stages of readiness. The key technologies are as follows: Cascading Style Sheets (CSS);

Document Object Model (DOM); and client-side scripting. It will help your authoring skills if you have a little historical perspective on how the Web arrived at DHTML.

For many years, the HTML standard was intended for the rendering of static content — not much more than an electronic version of a printed page. The most interactive part of a page was a form, which included buttons to click and text boxes to fill in. But for anything to change on the page, the content had to be served up again from the host computer.

Client-side scripting, as first implemented through JavaScript way back in NN2, opened the way for HTML pages to not only contain some smarts, but to also control individual pieces of content on the page without fetching a modified page from the server. At first, only form control elements were scriptable. Soon thereafter, images could be swapped, although the rectangular space for the image was fixed when the page loaded. More dynamism accrued to pages in NN4 by way of the layer, which acted like a borderless, transparent, or opaque window that could contain its own HTML document content and be positioned anywhere on the page — it could even overlap content on the main page or other layers. A layer's entire content could be modified without touching the rest of the page or other layers.

But the real breakthrough in dynamism came in IE4, whose rendering engine permitted any element to be modified, inserted, or removed on the fly, while the rest of the page reflowed its content quickly and automatically in response to the change. At the same time, an accepted standard for style sheets (CSS) opened the way for scripts to modify the look of content already on the page. Text could change colors when a cursor rolled atop it by either adjusting the style sheet property associated with the text or changing the style sheet rule that applies to the text.

Development activity at both Netscape and Microsoft eventually led to a standard for the Document Object Model as a way for scripts to control HTML content directly. Unfortunately, the browser makers frequently implemented first, and only then tried to establish their implementations as standards. Sometimes, the implementations were not as complete as the standards became, leaving the browsers in states that only partially implement the standards, while paying homage to legacy implementations. Netscape used the occasion of developing an entirely new code base for what became the Mozilla browser family to try to sever some ties with the past. In many respects the Mozilla browser represents the state of the standard art as implemented so far, although an argument could certainly be made that Opera is even more standards-compliant than Mozilla browsers.

Unfortunately, IE is still attempting to cater to both the legacy implementation and the standards, creating a massive DOM implementation with significant overlap in functionality with different syntaxes. Thus, the result of proprietary explorations and industry standards is a choice of modern browsers that permit a wide range of dynamic activity on content that reaches the user. Browsers that had started life as sleepy renderers of a tiny HTML vocabulary have grown into powerful front ends for server applications, and even self-contained applications of a sort that execute entirely on the client computer.

Although it's safe to say that version 4 browsers are now officially passé, mobile browsers are becoming increasingly popular, and they share some of the same limitations as legacy browsers. More specifically, many mobile browsers are streamlined to the extent that they have limited or no DOM support, and therefore little or no scripting support. If you're creating a scripted web

application, even if it is targeting modern browsers, it may still make sense to include code that gracefully degrades for mobile browsers.

Standards for CSS, DOM, and JavaScript/ECMA scripting have been well covered earlier in this book. The purpose of this chapter is to help you produce content that runs on DHTML-capable browsers, as well as mobile browsers with no DHTML support at all. Historically, most of the DHTML compatibility problems stem from page authors trying to develop for essentially three different document object models: NN4, IE4+, and W3C DOM (as implemented in IE5+/NN6+/Moz, Gecko, and WebKit- and Presto-based browsers). Nowadays, with the exception of having to accommodate IE's proprietary event model, it's reasonably safe to focus on the latter approach, while taking into consideration the limitations that are often associated with mobile browsers.

Striving for Compatibility

With as many as three object models to support (you can, of course, elect to support only a subset of browsers, if you like) you should look for ways to minimize your pain. If the NN4 object model is in your mix, you will very likely experience moments of sheer torture, as you try to get even the CSS-supported HTML to behave as it does in browsers of the other object models. Thankfully, the NN4 browser's installed base is quickly becoming a remnant of the past.

There are two keys to survival among the object models: knowing each DOM's limitations and finding common denominators.

Between the IE4+ and W3C DOMs, the biggest differences, outside of the conflicting event model syntax, fall along operating system and browser brand lines. For example, Microsoft takes advantage of the integration of the IE browser and the Windows operating system to such an extent that it can provide IE services, such as ActiveX controls and filters, that work only on Windows versions of IE. MacIE users are out of luck, as are users of any other browser brand.

Looking for areas of commonality — or at least gaining a clear understanding of where the models diverge — can be a tedious, yet personally rewarding, pursuit. Because Microsoft implemented some basic W3C DOM element referencing syntax as early as IE5, one compatibility hassle has all but disappeared. That still leaves, however, the W3C DOM's finger-twisting industry standard `document.getElementById()` method to obtain a reference to any ID'd element (implemented in IE5+/NN6+/ Moz, Gecko, and WebKit- and Presto-based browsers).

As soon as your script has a valid reference to an element, the next step is to read or write some property, or invoke some method that governs the element's position (and possibly other style) attributes. Here, again, the object models diverge, but not quite as severely. The W3C DOM works its positioning magic through the `style` property of a positioned element. In some cases the "last-dot" property names are identical across all three models (for example, `document.myLayer.zIndex`, `document.all.myLayer.style.zIndex`, and `document.getElementById("myLayer").style.zIndex`). Building a reference to reach that last dot, however, is where some of your hard work must go. You can also use the W3C `window.getComputedStyle()` method to obtain read-only access to all of the CSS styles associated with a particular element.

Each DOM also has its own event model. Whereas IE5+ overlaps its DOM features with both the IE4+ and, to some extent, the W3C DOM, the event models don't follow the same lines of

implementation. As of WinIE8 and MacIE5, IE does not implement any of the W3C DOM event model, although Mozilla-based browsers and Safari do.

The bottom line, then, is that letting your scripts decide how to perform actions based on the browser version is not a good idea. Instead, the scripts should be smart enough to act based on the capabilities of the browser that is currently running the script. The good news is that with a few exceptions (IE events, for example) it is now possible to forego the version 4 DOMs and focus solely on supporting the W3C DOM in modern browsers.

Working Around Incompatibilities

To create DHTML for multiple DOMs, you must find ways to accommodate incompatible object references and occasionally incompatible property names. Scripting gives you several alternatives to working your way around these potential problems. Some of the approaches you can take are now passé. They are described here partly for the sake of historical reference, but also because you will see many instances of these approaches in legacy DHTML applications from the days when authors only had to worry about two DOMs (NN4 and IE4). The real meat of this discussion comes later, when you learn more about object detection and custom APIs.

Old-fashioned compatibility tricks

Until late 2000, it was necessary to write cross-browser DHTML applications that had to run on two classes of browser: NN4 and IE4. Two approaches to writing code for these two DOMs grew in popularity: inline branching and platform equivalency. They are described here, not for you to apply, but for you to understand what the pioneers did, in case you encounter their code in your web surfing.

Inline branching

The idea behind inline branching is that your scripts use `if...else` decisions to execute one branch of code for one browser and another branch for the other browser. Before you can begin to write code that creates branches for each browser, you should define two global variables at the top of the page that act as Boolean flags for your `if...else` constructions later. Therefore, at the first opportunity for a `<script>` tag in a page, include the following code fragment to set flags named `isNav4` and `isIE4`:

```
var isNav4, isIE4;
if (parseInt(navigator.appVersion) == 4)
{
   if (navigator.appName == "Netscape")
   {
      isNav4 = true;
   }
   else if (navigator.appVersion.indexOf("MSIE") != -1)
   {
      isIE4 = true;
   }
}
```

Version checking here is quite specific. First of all, it intentionally confines access to browsers whose versions come back as version 4. This code, written when the browsers were still at version 4, was remarkably prescient. Our concern at the time was that DHTML was so volatile that it was unknown if future browser versions would be backward compatible with the code to be run inside branches governed by the two global variables. As it turned out, NN6 (whose `navigator.appVersion` reports 5) is not backward compatible with the layer structure of NN4, so that locking the NN4 branches to NN4 became a good thing. On the IE side, the `navigator.appVersion` property continues to report 4, even through IE8, which is backward compatible with IE4. Thus, any branch dedicated to IE4 executes under this scheme and remains syntactically accurate.

Another aspect of the flag-setting script we should mention is that the example provides no escape route for browsers that aren't level 4, and aren't either Navigator or Internet Explorer (should there be a level 4 browser from another brand). In a production environment, we would either prefilter access to the page or redirect ill-equipped users to a page that explains why they can't view the page. In the structure of the preceding script, redirection would have to be made in two places, as follows:

```
var isNav4, isIE4;
if (parseInt(navigator.appVersion) == 4)
{
   if (navigator.appName == "Netscape")
   {
      isNav4 = true;
   }
   else if (navigator.appVersion.indexOf("MSIE") != -1)
   {
      isIE4 = true;
   }
   else
   {
      location = "sorry.html";
   }
}
else
{
   location = "sorry.html";
}
```

Note

In case you're curious, Firefox and other Mozilla-based browsers still report `Netscape` **as their** `appName`**. See for yourself by entering** `navigator.appName` **in the first entry box in The Evaluator Sr. (discussed in Chapter 4, "JavaScript Essentials").** ■

With the global variables defined in the document (and unsupported browsers redirected elsewhere), you can use them as conditional values in branching statements that address an object according to the reference appropriate for each platform. For example, to change the `visibility` property of an object named `instructions`, you use the flags as follows:

```
if (isNav4)
{
   document.instructions.visibility = "hidden";
}
else
{
   document.all.instructions.style.visibility = "hidden";
}
```

As the browser DOMs evolve, expand, and fragment, inline branching becomes increasingly less practical. With so many permutations of the DOM, according to browser brand, browser version, and operating system, you can drive yourself crazy trying to accommodate them all and maintain the code going forward. This approach also eliminates from consideration any non-NN or non-IE browser (such as Opera or WebKit-based browsers, although all of them have reported themselves as something other than they are at one time or another) that may have the capabilities needed to play your DHTML scripts. This approach also limits the possibility that future browsers with higher `navigator.appVersion` values can take advantage of your scripts. In short, this browser-sniffing approach is predominantly a remnant of the scripting days of old, although it does still resonate now on occasion when you need to take advantage of a Microsoft-specific browser feature, or must accommodate a bug in a specific previous version of a browser.

Platform equivalency

Another legacy compatibility technique attempts to limit the concern for the different ways the IE4 and NN4 platforms refer to a positionable element (because cross-browser DHTML were pretty much limited to the properties affecting positionable elements). If you examine the formats for each platform's object references, you see that they all contain a reference to the `document` and to the object name or ID. The IE4+ DOM syntax also includes property words, such as `all` and `style`. If you assign these extra property names to variables for IE4, and leave those variables as empty strings for NN4, you can assemble an object reference for those two platforms in one statement.

Note

If all this talk of version 4 browsers has you thinking we're living in the wrong century, just keep in mind that you have to understand history in order to learn from it. Even modern browsers are far from perfect with regard to compatibility, so unfortunately it still isn't possible to think entirely in terms of JavaScript code that can be written once and run everywhere, without a few compatibility hacks here and there. Understanding how version 4 compatibility problems were solved will help when you inevitably run across a modern day browser problem. ■

To begin using this technique, set two global variables that store reference components for the scope (`all` in IE4) and the `style` object (`style` in IE4):

```
var range = "";
var styleObj = "";
if (parseInt(navigator.appVersion) == 4)
{
```

```
      if (navigator.appVersion.indexOf("MSIE") != -1)
      {
         range = "all.";
         styleObj = ".style";
      }
   }
```

From this point, you can assemble an object reference with the help of the JavaScript `eval()` function, as follows:

```
var instrux = eval("document." + range + "instructions" + styleObj);
instrux.visibility = "hidden";
```

Or, you can use the `eval()` function to handle the entire property assignment in one statement, as follows:

```
eval("document." + range + "instructions" + styleObj +
     ".visibility = 'hidden'");
```

If your page does not have a lot of objects that your scripts will be adjusting, you can use this platform equivalency approach to create global variables holding references to your positionable objects at load time (triggered by the `onload` event handler so that all objects exist and can be referenced by the `eval()` function). Then, use those variables for object references throughout the scripts.

Unfortunately, the platform equivalency methodology breaks down when an NN4 layer object is nested inside another layer. The platform equivalency formulas assume that each object is directly addressable from the outermost `document` object. If your objects have a variety of nested locations, you can use either the inline branching method described earlier, or batch-assign objects to global variables at load time, using platform branching techniques.

Modern approaches to compatibility

Although in-line branching and platform equivalency were suitable for their generations, modern browsers call for better approaches to simplify authoring for multiple DOMs. Browser-sniffing is no longer a workable strategy, except in rare instances. Techniques more suitable for today — object detection and custom APIs — are not really new. But these techniques are the preferred way to build cross-browser scripts with an eye to compatibility, both backward and forward. The good news is that these techniques are becoming less and less necessary as time goes by and browsers continue to adhere more closely to published standards.

Object detection

The subject of object detection has been mentioned in several places in earlier chapters of this book. The technique has been used for a long time to let a browser not equipped to handle image objects gracefully skip over image swapping script segments:

```
if (document.images)
{
   // statements to work with image objects
}
```

If there is no `document.images` property for a browser, the condition evaluates to `undefined`, which the condition treats as being `false`.

But object detection has also been misused in the past, especially in the DHTML realm, to substitute for browser version detection. For example, if a browser supported the `document.all` collection, a global variable was set to indicate that the browser was IE4 or later; the existence of `document.layers` supposedly meant that the browser was NN4. It was a mistake, however, to link a browser version with the existence of an object or property (Opera supports the `document.all` collection, as well as the W3C). Instead, object detection should be used only if your script statements will be addressing that object, just as the `document.images` condition does in the previous example.

To demonstrate this tactic, consider the need to assemble a reference to an object so that it is ready to have one of its DHTML properties adjusted. Each of the three DOMs (NN4, IE4+, W3C) has its own syntax for assembling such a reference, and each syntax relies on the existence of a particular object or property. The function shown in Listing 50-1 (not by itself, but as a part of Listing 50-3) enables you to pass the name or ID of a positioned element (either in string form or object form) to receive back a valid reference to the object with which style-related properties are associated — all without resorting to the `eval()` function in any form.

LISTING 50-1

Using Object Detection to Assemble an Element Object Reference

```
function getObject(obj)
{
   if (document.getElementById)
   {
      if (typeof obj == "string")
      {
         return document.getElementById(obj).style;
      }
      else
      {
         return obj.style;
      }
   }
   if (document.all)
   {
      if (typeof obj == "string")
      {
         return document.all(obj).style;
      }
      else
      {
         return obj.style;
      }
   }
```

continued

LISTING 50-1 *(continued)*

```
   if (document.layers)
   {
      if (typeof obj == "string")
      {
         // just one layer deep
         return document.layers[obj];
      }
      else
      {
         // can be a nested layer
         return obj;
      }
   }
   return null;
}
```

The primary object detection for each of the three sections of this function looks for the presence of categories of objects (`document.layers` and `document.all`) or a particular method (`document.getElementById()`), and then — this is the important part — the script uses those detected objects in the statements. The script doesn't know IE4 from NN7; it *does* know how to derive valid references for three different object models, and employs the syntax of the first one for which the associated object property or method is supported.

In practice, the order of the three sections should have no bearing on your scripts, but you should be aware of one subtlety: IE5+ can work with either of the first two sections, because those browsers detect `document.all` and `document.getElementById` as valid references. In the order shown in the listing, you force IE5+ to use W3C DOM syntax. The results, however, are the same: A valid reference to the `style` object associated with an element.

The code in Listing 50-1 is backward compatible with browsers all the way back to NN4 and IE4, which is fine but probably not necessary at this point in time. If you want to focus solely on modern (W3C) browsers, then the previous code can be easily modified (see Listing 50-2) to accommodate them.

LISTING 50-2

Assembling a W3C DOM Element Object Reference

```
function getObject(obj)
{
   if (document.getElementById)
   {
      if (typeof obj == "string")
      {
         return document.getElementById(obj).style;
```

```
      }
      else
      {
         return obj.style;
      }
   }
   return null;
}
```

In Listing 50-2, only the final `if` clause is kept around since it focuses solely on W3C browsers. You could argue that the function isn't even necessary since it calls the `getElementById()` function regardless of the browser. However, this version of `getObject()` still includes the flexibility of allowing you to pass an object or an ID.

Custom APIs

Notions of object detection and simplifications of your scripts come together in the final approach to building cross-browser DHTML: Writing a custom application programming interface (API). A JavaScript custom API is a library of functions you design to act as an intermediary between your scripts and other scriptable entities. Ideally, an API simplifies access to, or control of, other entities. In the context of designing a cross-browser DHTML page, an API can offer a single function that smoothes over the differences in object references or property names among several platforms. Your custom function provides a single access point that is consistent across all platforms. In essence, you are creating your own meta-vocabulary for methods and property settings.

The element object reference maker in Listings 50-1 and 50-2 is a good start for such an API, because all other functions for moving, hiding, showing, and changing the stacking order of a positionable element need a valid style-oriented reference to the element.

Consider an interface that deals with incompatible ways of adjusting the location of a positionable element. In this case, the act of moving an element has different syntax in different DOMs. One group (NN4 for layers) uses the `moveTo()` method; the rest support `left` and `top` properties of their `style` object:

```
// position an object at a specific pixel coordinate
function shiftTo(obj, x, y)
{
   var theObj = getObject(obj);
   if (theObj.moveTo)
   {
      theObj.moveTo(x,y);
   }
   else if (typeof theObj.left != "undefined")
   {
      theObj.left = x + "px";
      theObj.top = y + "px";
   }
}
```

Note

In reality, it is no longer necessary to write JavaScript code that is backward compatible with version 4 browsers. However, with the track records that browser vendors have established, you are still likely to encounter situations where you may need to employ such branching techniques to fully support popular browsers. Therefore, this example is a good reference for how to develop code that can live happily across browsers with compatibility issues. ■

Notice one workaround, which, on the surface, isn't pretty: The second branch must perform an odd way of object detection. We're stuck with having to make a tradeoff when it comes to checking for the existence of a style property. If the page uses style sheets defined in `<style>` tags (or imported into the page from external style sheet files), the element affected by the rule does not yield the rule's property values through the element's `style` property. The property exists, but its value in this case (or until it is set by script) is an empty string. IE5+ provides a `currentStyle` property to give us the effective values, but that property is not (yet) a part of the DOM standard. It's true that you can derive the effective style property of an element through the W3C DOM (see Chapter 38, "Style Sheet and Style Objects"), but even if you assign the style sheet through the element's `style` attribute (in which case the style property values come through), detecting the presence of the property with the conditional expression

```
if (theObj.left)
```

is not practical here anyway. If the effective value of the `left` and `top` properties were empty strings (or zero for numeric style property values), the conditional expression would evaluate to the equivalent of `false`, making it appear as though the property doesn't exist. To validate the existence of the property, the conditional expression verifies that the value of a named property has a type other than *undefined*. It may seem like a long way to go to prove the existence of a property, but it works, even if the value is an empty string or zero.

It is important that both branches perform object detection. Although it is unlikely (but, as we learned from the transition between NN4 and NN6, not impossible), if a future browser should completely alter its vocabulary, omitting the objects being detected here, the function ends gracefully, without generating script errors.

An API is usually best deployed as an external `.js` file. One such API file is described later in this chapter. Bear in mind, however, that a lengthy API is downloaded to the browser, no matter how much or how little of it your main scripts use. Blindly linking in a big library just to use a few of its functions is a mistake. You serve your users better if you create a subset of the API, and link the subset to the page (or drop the few functions directly into the page's scripts if the combination is not reused on a lot of pages).

Handling non-DHTML browsers

An important question to ask yourself as you embark on a DHTML-enhanced page is how you intend to treat visitors whose browsers aren't up to the task. In many respects, the problem is similar to the problem of treating nonscriptable browsers when your page relies on scripting (see Chapter 4). Both mobile browsers and desktop browsers that have scripting disabled are potentially capable of presenting a problem in this regard. Keep in mind, however, that multiple CSS style sheets can be specified by the page, which is a non-scripting way to handle the look and feel of your web site on a mobile browser, accessible device, or other device.

The moment your page uses DHTML to position an element, you must remember that non-DHTML browsers display the content according to traditional HTML rendering rules. No elements are allowed to overlap. Any block-level tag is rendered at the left margin of the page, unless some other non-DHTML alignment (center or right) is at work. This goes for elements that you design to be DHTML-positioned to sit offscreen (perhaps with a clickable tab) until called by the user. An element defined as being hidden or not displayed in DHTML will be visible. In most cases, your carefully designed DHTML page will look terrible.

However, a page that does not use too radical a layout strategy may still be usable in non-DHTML browsers. You should try to check your DHTML-enabled page in a mobile browser with limited or no scripting support to see how it looks. Perhaps there isn't too much you need to do to degrade the DHTML so that the page is acceptable in limited browsers.

The ultimate responsibility for deciding your compatibility strategy with mobile browsers rests with you and your perceptions about your page visitors. If they are in need of vital information from your site and that information is readable in non-DHTML browsers, that may be enough. Otherwise, you must provide a separate content path for both types of browsers. This may make more sense anyway since mobile web pages often benefit from their own unique graphical content and layout schemes.

A DHTML API Example

Now it's time to get to a real DHTML API that you can use and build upon for your own applications. Listing 50-3 contains the API code, which is most likely to be deployed as an external `.js` library file. In fact, this API is used as-is in a map puzzle game application in Chapter 59, "Application: Cross-Browser DHTML Puzzle Map" (on the CD-ROM). You can see there how it is used to control element positioning, dragging, and layering for W3C DOM browsers, while also making sure that features are supported before summoning them. The code in Listing 50-3 is longer than most listings in this book, so, for your convenience, we interlace commentary within the long listing.

The library starts with the `getObject()` function shown earlier in this chapter. Virtually every other function in this library makes a trip to `getObject()` to convert the name of the object passed as a parameter to an object reference whose positionable (or other style-related property) can be adjusted.

LISTING 50-3

The Custom API

DHTML api.js

```
// convert object name string or object reference
// into a valid object reference ready for style change
function getObject(obj)
{
   if (document.getElementById)
   {
      if (typeof obj == "string")
```

continued

LISTING 50-3 *(continued)*

```
      {
         return document.getElementById(obj).style;
      }
      else
      {
         return obj.style;
      }
   }
      if (document.all)
      {
         if (typeof obj == "string")
         {
            return document.all(obj).style;
         }
         else
         {
            return obj.style;
         }
      }
      if (document.layers)
      {
         if (typeof obj == "string")
         {
            return document.layers[obj];
         }
         else
         {
            return obj;
         }
      }
   return null
}
```

A pair of functions handles all motion of positionable elements. The first function, `shiftTo()` takes three parameters: the ID of the object being moved, and the horizontal and vertical pixel coordinates of the top-left corner of the element. The assumption is that the main page script that invokes this library function performs the calculation of the coordinates. You see that code in Chapter 59. Inside this function, the `left` and `top` style properties for W3C DOM browsers are set. Even though the adjustments are made in separate statements, the action on the screen does not follow the action statement by statement. Between screen buffering and quick execution, the repositioning appears as a single shift.

```
// position an object at a specific pixel coordinate
function shiftTo(obj, x, y)
{
   var theObj = getObject(obj);
      if (theObj.moveTo)
```

```
        {
           theObj.moveTo(x,y);
        }
        else if (typeof theObj.left != "undefined")
        {
           theObj.left = x;
           theObj.top = y;
        }
}
```

The shiftBy() function mimics the NN4 layer.moveBy() method. The second and third parameters represent the number of pixels that the object should be moved on the page. A positive number means a shift to the right or down; a negative number means to the left or up; a value of zero means no change to the axis. The passed values are added to the left and top properties to carry out the shift. Notice that because these properties return strings that include the units for the measurements, the incremental values are added to integer extractions from the current settings. And because the units being used here are the default (pixels), no units have to be assigned with the new values (although they could without penalty). However, the CSS standard requires specification of the units.

```
// move an object by x and/or y pixels
function shiftBy(obj, deltaX, deltaY)
{
   var theObj = getObject(obj);
      if (theObj.moveBy)
      {
         theObj.moveBy(deltaX, deltaY);
      }
      else if (typeof theObj.left != "undefined")
      {
         theObj.left = parseInt(theObj.left) + deltaX;
         theObj.top = parseInt(theObj.top) + deltaY;
      }
}
```

The setZIndex() function does little more than convert the object reference and assign the incoming value to the zIndex property.

```
// set the z-order of an object
function setZIndex(obj, zOrder)
{
   var theObj = getObject(obj);
   theObj.zIndex = zOrder;
}
```

The setBGColor() function utilizes the backgroundColor style property in order to alter the background color.

```
// set the background color of an object
function setBGColor(obj, color)
{
   var theObj = getObject(obj);
      if (theObj.bgColor)
      {
```

```
            theObj.bgColor = color;
         }
         else if (typeof theObj.backgroundColor != "undefined")
         {
            theObj.backgroundColor = color;
         }
      }
```

Allowable values for the `visibility` property are very unprogrammatic in our opinion. We expect a Boolean value rather than strings. To accede to reality while making the process of showing and hiding elements more logical to us, we created API functions called `show()` and `hide()`.

```
      // set the visibility of an object to visible
      function show(obj)
      {
         var theObj = getObject(obj);
         theObj.visibility = "visible";
      }

      // set the visibility of an object to hidden
      function hide(obj)
      {
         var theObj = getObject(obj);
         theObj.visibility = "hidden";
      }
```

Although the `left` and `top` properties of visual objects in modern browsers include unit values, it is still safe to use `parseInt()` on the values returned from the properties. The need for these API functions came from the way the map puzzle application in Chapter 59 works. For a couple of operations, it calculates the destination for an object with respect to the position of one of the other positioned elements. These functions return the values needed for the main program's calculation. This is also an example of how you may need to embellish the API for your own application.

```
      // retrieve the x coordinate of a positionable object
      function getObjectLeft(obj)
      {
         var theObj = getObject(obj);
         return parseInt(theObj.left);
      }

      // retrieve the y coordinate of a positionable object
      function getObjectTop(obj)
      {
         var theObj = getObject(obj);
         return parseInt(theObj.top);
      }
```

The previous API is generalizable enough to be used as a library with any modern cross-platform DHTML application using positioning. Should the API be used in a browser that doesn't support DHTML, the code will fail cleanly and without error.

CHAPTER

51

Internet Explorer Behaviors

IN THIS CHAPTER

- Introducing IE behaviors
- Understanding the structure of behavior XML files
- Exploring behavior samples

Back in the days of Internet Explorer 5 for Windows, Microsoft first introduced a technology known as *DHTML behaviors*, which added enhanced features to the behaviors familiar to standard HTML elements. Microsoft and others have proposed the behaviors concept to the W3C, and it could someday become one of the W3C standard recommendations. Such a standard might not be implemented exactly the way Microsoft has implemented behaviors, but most of the concepts are the same, and the syntax being discussed so far is similar. While there is no guarantee that the W3C will adopt behaviors as a standard, you will see that the concept seems to be a natural extension to the work that has already been adopted for both CSS and XML.

Although the W3C did initiate an effort called Behavioral Extensions to CSS, there hasn't been much progress on the behavior standardization front recently. For the latest document describing the work of the participants of the standards discussions, visit `http://www.w3.org/TR/becss`.

Style Sheets for Scripts

You can best visualize what a behavior is in terms of the way you use style sheets. Consider a style sheet rule whose selector is a tag or a class name. The idea behind the style sheet is that one rule, which can define dozens of rendering characteristics for a chunk of HTML content, can be applied to perhaps dozens, if not hundreds, of elements within the document. A corporation may design a series of rules for the way its web documents will look throughout the web site. If the designer decides to alter the font family or color for, say, `h1` elements, that change is made in one place (the external style sheet file), and the impact is felt immediately across the entire site. Any page that includes an `h1` element renders the header with the newly modified style.

Imagine now that instead of visual styles associated with an element, you want to define a behavioral style for a particular group of elements. A *behavioral style* is the way an element responds to predominantly user interaction with the element. For example, suppose the design specifications for a web site indicate that all links should have their text colored a certain way when at rest, but on mouse rollovers, the text color changes to a more contrasting color, the font weight increases to bold, and the text becomes underlined. Those modifications require scripts to change the style properties of the element in response to the mouse action of the user. The scripts that fire in response to specific user actions (events) are written in an external file known as a behavior, and a behavior is associated with an element, class, or tag through the same CSS syntax that you use for other style properties.

A behavior, of course, assumes that its scripts can work with whatever HTML element is associated with the behavior. Just as it would be illogical to associate the `tableLayout` style property with an element that wasn't a `table`, so, too, would it be illogical to associate a behavior, whose scripts employed `table` object properties and methods, to a `p` element. Even so, a well-designed behavior can obtain details about the element being manipulated through the element object's properties. The better you are at writing generalizable JavaScript functions, the more successful you will be in implementing behaviors.

Embedding Behavior Components

IE treats each behavior as a component, or add-on building block for the browser. WinIE5+ comes equipped with a handful of behaviors built into the browser (the so-called default behaviors, which happen to rely on specific XML elements embedded in a document). Behaviors that you create most likely exist as separate files on the server, just like external `.css` and `.js` files do. The file extension for a behavior file is `.htc` (standing for HTML Component).

Linking in a behavior component

To associate a behavior with any single element, class of elements, or tag as the page loads, use CSS rule syntax and the IE-specific `behavior` attribute. The basic syntax is as follows:

```
selector {behavior:url(componentReference);}
```

As with any style sheet rule, you can combine multiple rule properties, delimiting them with semicolons. The format of the `componentReference` depends on whether you are using one of the IE default behaviors or a behavior you've written to an external file. For default behaviors, the reference is in the format:

```
#default#componentName
```

For example, if you want to associate the `download` behavior with any element of class `downloads` use:

```
.downloads {behavior:url(#default#download);}
```

Relative or absolute URIs to external `.htc` files can also be specified. For example, if your site contains a directory named `behaviors` and a file named `hilite.htc`, the style sheet rule from the root directory is:

```
.hiliters {behavior:url(behaviors/hilite.htc);}
```

As with all Cascading Style Sheet rules, behaviors can be specified in a `style` element of the page, in the `style` attribute of an individual element, or in a rule defined inside an imported `.css` file.

Enabling and disabling behaviors

In Chapter 26, "Generic HTML Element Objects," you can find details of IE methods for all HTML elements that let scripts manage the association of a behavior with an element after the page has loaded. Invoking the `addBehavior()` method on an element assigns an external `.htc` file to that element. When you no longer need that behavior associated with the element, invoke the `removeBehavior()` method.

Component Structure

An `.htc` behavior file is a text file consisting of script statements inside a `<script>` tag set, and some special XML tags that IE knows how to parse. You create `.htc` files in the same kind of plain text editor that you use for external `.js` or `.css` files.

Script statements

Unlike external `.js` files, an `.htc` behavior file includes `<script>` tags, which surround any JavaScript (or VBScript, if you like) statements that control the behavior. Because a behavior most typically is written to control one or more aspects of the HTML element to which it is connected, statements tend to operate only on the associated object element. A special reference — `element` — is used to refer to the element object itself (much like the way the `this` keyword in a custom object's method self-refers to the object associated with the method).

If your behavior will be modifying either the content or style of the element, use the `element` reference as a foundation for the reference to one of that element object's properties or methods. For example, if a statement in a behavior needs to set the `style.visibility` property so that the element hides itself, the statement in the behavior script is:

```
element.style.visibility = "hidden";
```

Any valid reference from the point of view of the element object is fair game, including references to the element's `parentElement`, even though the parent element is not explicitly associated with the behavior.

Variable scope

Except for the special `element` reference, script content of a behavior is completely self-contained. You can define global variables in the behavior that are accessible to any script statement in the behavior. But a global variable in a behavior does not become a global variable for the main document's scripts to use. You can expose variables so that scripts outside of the behavior can get to them (as described later in the chapter), but this exposure is not automatic.

Most of the script content of a behavior consists of functions that usually interact in some fashion with the associated element (via the element's properties and/or methods). Local variables in functions have the same scope and operate just like they do in regular script functions. Global variables you define in a behavior, if any, are usually there for the purpose of preserving values between separate invocations of the functions.

Assigning event handlers

Functions in a behavior are triggered from outside the behavior through two means: event handlers and direct invocation of functions declared as public (described in the next section). Event handler binding is performed in a way that is not used elsewhere in the IE DOM. Each event type (for example, `onmouseover`, `onkeypress`) requires its own special XML tag at the top of the behavior file. The format for the event handler tag is as follows:

```
<public:attach event="eventName" onevent="behaviorFunctionName()" />
```

As the behavior loads, the `public:attach` tag instructs the browser to expose to the "public" (that is, the world outside of the behavior) an event type (whose name always begins with the "on" prefix in the IE4+ event model); whenever an event of that type reaches the behavior's element, the function (defined within the behavior file) is invoked. In XML terminology, the `public:` part of the tag is known as a *namespace*, and IE includes a built-in parser for the `public` namespace. Notice, too, the XML syntax at the end of the tag that allows a single set of angle brackets to act as a start and end tag set (there is no content for this tag, just the attributes and their values).

To demonstrate, imagine that a behavior has a function named `underlineIt()`, which sets the `element.style.textDecoration` property to underline. To get the element to display the underline decoration as the user rolls the mouse atop the element, bind this function to the element's `onmouseover` event handler as follows:

```
<public:attach event="onmouseover" onevent="underlineIt()" />
```

If you examine the wording of the opening part of the tag, you may recognize a connection to the IE event model's `attachEvent()` method, which operates on all HTML elements (see Chapter 26). You can have as many event-binding tags as your element needs. To invoke multiple functions in response to a single event type, simply add the subsequent function invocation statements to the `onevent` attribute, separating the calls by semicolons (the same as with regular JavaScript statement delimiters).

Exposing properties and methods

XML tags with the `public:` namespace are also used (with different attributes) to expose a behavior's global variables as properties of the element and a behavior's functions as methods of the element. The syntax for both types of "public" announcements is as follows:

```
<public:property name="globalVarName" />
<public:method name="functionName" />
```

Values for both items are string versions of references to the variable and function (no parentheses). Again, you can define as many properties and methods for a behavior as you need.

As soon as a property and/or method is made public in a behavior, scripts from outside the behavior can access those items as if they were properties or methods of the element associated with the behavior:

```
document.all.elementID.behaviorProperty
document.all.elementID.behaviorMethod()
```

If you associate a behavior with a style sheet class selector, and several document elements share that class name, each one of those elements gains the public properties and methods of that behavior, accessible through references to the individual elements. That's because a behavior's scripts are written to read or modify properties of whatever element receives a bound event, or is referenced along the way to the public property or method.

Behavior Examples

The following two examples are intentionally simple to help you grasp the concepts of behaviors if they are new to you. The first example interacts with multiple elements strictly through event binding; the second example exposes a property and method that the main page's scripts access to interesting effect.

Example 1: Element dragging behavior

This book contains several examples of how to script a page to let a user drag an element around the browser window (Chapters 43, "Positioned Objects," and 59, "Application: Cross-Browser DHTML Map Puzzle," in particular). In all those examples, the dragging code and event handling was embedded in some fashion into the page's scripts. The first example of a behavior, however, drives home the notion of separating an element's behavior from its content (just as a CSS2 style sheet separates an element's appearance from its content).

Imagine that it's your job to design a page that employs three draggable elements. Two of the elements are images, while the third is a panel layer that also includes a form. If you haven't scripted DHTML before, this may sound like a daunting task at first, one rife with the possibility of including multiple versions of the same scripts to accommodate different kinds of draggable elements.

Now imagine that to the rescue comes a scripter who has built a behavior that takes care of all of the dragging scripting for you. All you do is assign that behavior by way of one attribute of each draggable element's style sheet rule. Absolutely no other scripting is required on the main page to achieve the element dragging.

Listing 51-1 shows the behavior file (`drag.htc`) that controls basic dragging of a positionable element on the page. You may recognize some of the code as an IE version of the cross-browser dragging code used elsewhere in this book (for a blow-by-blow account of these functions, see the description of the map puzzle game in Chapter 59). The names of the three operative functions and the basic way they do their jobs are identical to the other dragging scripts. Event binding, however, follows the behavior format through the XML tags. All interaction with the outside world occurs through the "public" event handlers.

LISTING 51-1

An Element Dragging Behavior

```
<public:attach event="onmousedown" onevent="engage()" />
<public:attach event="onmousemove" onevent="dragIt()" />
<public:attach event="onmouseup"   onevent="release()" />
<public:attach event="onmouseover" onevent="setCursor()" />
<public:attach event="onmouseout"  onevent="release();restoreCursor()" />

<script type="text/javascript">
   // global declarations
   var offsetX = 0;
   var offsetY = 0;
   var selectedObj;
   var oldZ, oldCursor;

   // initialize drag action on mousedown
   function engage()
   {
      selectedObj = (element == event.srcElement) ? element : null;
         if (selectedObj)
         {
            offsetX = event.offsetX - element.document.body.scrollLeft;
            offsetY = event.offsetY - element.document.body.scrollTop;
            oldZ = element.runtimeStyle.zIndex;
            element.style.zIndex = 10000;
            event.returnValue = false;
         }
   }

   // move element on mousemove
   function dragIt()
   {
```

```
      if (selectedObj)
      {
         selectedObj.style.pixelLeft = event.clientX - offsetX;
         selectedObj.style.pixelTop = event.clientY - offsetY;
         event.cancelBubble = true;
         event.returnValue = false;
      }
   }

   // restore state on mouseup
   function release()
   {
      if (selectedObj)
      {
         selectedObj.style.zIndex = oldZ;
      }
      selectedObj = null;
   }

   // make cursor look draggable on mouseover
   function setCursor()
   {
      oldCursor = element.runtimeStyle.cursor;
      element.style.cursor = "hand";
   }

   // restore cursor on mouseout
   function restoreCursor()
   {
      element.style.cursor = oldCursor;
   }
</script>
```

Notice a subtlety in Listing 51-1 that is implied by the element-specific scope of a behavior. Two statements in the `engage()` function need to reference scroll-related properties of the `document.body` object. Because the only connection between the behavior and the document is via the `element` reference, that reference is used along with the `document` property (a property of every HTML element object, as shown in Chapter 26). From there, the `body` object and the required properties can be accessed.

Listing 51-2 is a simple page that contains three elements that are associated with the `drag.htc` behavior through a style sheet rule definition (for the `draggable` class). The document is incredibly uncomplicated. Even the `drag.htc` file isn't very big. But together they produce a far more interesting page for the user than a couple of static images and a form.

LISTING 51-2

Three Draggable Elements Using the Behavior

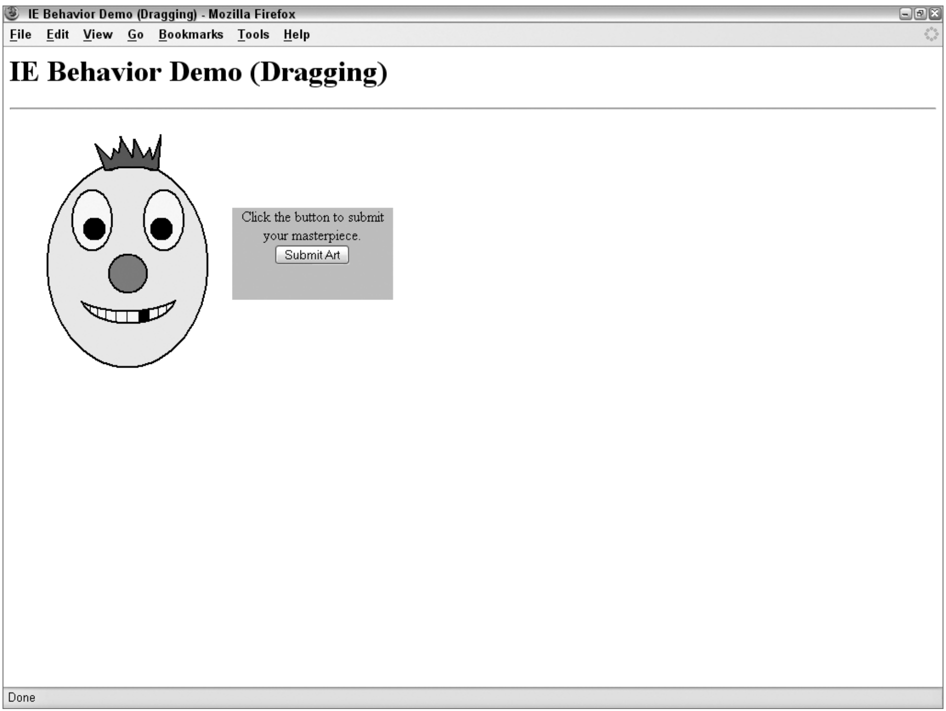
```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>IE Behavior Demo (Dragging)</title>
      <style type="text/css">
         .draggable {position:absolute; behavior:url(drag.htc)}
         #head {left:48px; top:125px}
         #eyes {left:75px; top:155px}
         #nose {left:115px; top:225px}
         #mouth {left:85px; top:275px}
         #hair {left:100px; top:95px}
         #txt1 {left:250px; top:175px; background-color:aqua; width:175px;
                                      height:100px; text-align:center}
      </style>
   </head>
   <body>
      <h1>IE Behavior Demo (Dragging)</h1>
      <hr />
      <img alt="head" class="draggable" id="head" src="head.png" />
      <img alt="eyes" class="draggable" id="eyes" src="eyes.png" />
      <img alt="nose" class="draggable" id="nose" src="nose.png" />
      <img alt="mouth" class="draggable" id="mouth" src="mouth.png" />
      <img alt="hair" class="draggable" id="hair" src="hair.png" />
      <div class="draggable" id="txt1">
         Click the button to submit your masterpiece.
         <form>
            <input type="button" value="Submit Art" />
         </form>
      </div>
   </body>
</html>
```

Obviously, the dragging example here is very rudimentary. It isn't clear from the sample code what the user gets from the page, other than the joy of moving things around and manipulating the smiley face (see Figure 51-1).

If you were designing an application that genuinely benefits from draggable objects (for example, the map puzzle in Chapter 59), you can easily enhance the behavior to perform actions, such as snapping a dragged element into place when it is within a few pixels of its proper destination. For such an implementation, the behavior can be given some extra global variables, akin to the values assigned to the state objects in Chapter 59, including the pixel coordinates of the ideal destination for a dragged element. An `onload` event handler for the page can fire a public `init()` function in each element's behavior to assign those coordinate values. Any event that can bubble (such as mouse events) does so from the behavior to the target. Therefore, you can extend the event action of the behavior by adding a handler for the same event to the element outside of the behavior.

FIGURE 51-1

This draggable JavaScript example is somewhat of an ode to the classic Mr. Potato Head toy.



Example 2: Text rollover behavior

In the second example, you see how a behavior exposes a global variable and function as a public property and method, respectively. The demonstration reinforces the notion that even if a single behavior file is associated with multiple elements (for example, the elements share the same class, and the behavior is assigned to the class), each behavior maintains its own variable values, independent of the other elements and their behaviors.

The nature of this behavior is to set the `color` style property of the associated element to either a default color (red) or to another color that has been passed into the behavior via one of its public methods. The color setting is preserved in one of the behavior's global variables, and that variable is exposed as a public property.

Listing 51-3 shows the `.htc` behavior file's content. Only two events are bound to this behavior: `onmouseover` and `onmouseout` — the typical rollover events. The `onmouseover` event invokes the `makeHot()` function, while the `onmouseout` event invokes the `makeNormal()` function. Before the `makeHot()` function makes any changes to the `color` and `fontWeight` style properties of the element, existing settings are preserved in (non-public) global variables in the behavior. This allows the `makeNormal()` function to restore the original settings, regardless of what document styles may be applied to the element in a variety of pages. That's something to

keep in mind when you design behaviors: they can be deployed in pages controlled by any number of style sheets. Don't assume any basic style setting; instead, use the `currentStyle` property to read and preserve the effective property values before touching them with your behavior's modification scripts.

Neither of the event handler functions are exposed as public methods. This was a conscious decision for a couple of reasons. The most important reason is that both functions rely on being triggered by a known event occurring on the element. If either function were invoked externally, the event object would contain none of the desired information. Another reason behind this comes from a common programming style for components, which protects inner workings, while exposing only those methods and properties that are "safe" for others to invoke. For this code, the public method does little more than set a property. It's an important property, to be sure, and one of the protected functions relies on it. But by allowing the public method little room to do any damage other than execution of the behavior, the design makes the behavior component that much more robust.

Assigning a color value to the public property and passing it as a parameter to the public method accomplishes the same result in this code. As you will see, the property gets used in the demonstration page to retrieve the current value of the global variable. In a production behavior component, the programmer would probably choose to expose this value strictly as a read/write property, or to expose two methods, one for getting and one for setting the value. The choice would be at the whim of the programmer's style and would likely not be both. Using a method, however, especially for setting a value, creates a framework in which the programmer can also perform validation on the incoming value before assigning it to the global variable (something the example here does not do).

LISTING 51-3

Rollover Behavior (makeHot.htc)

```
<public:attach event="onmouseover" onevent="makeHot()" />
<public:attach event="onmouseout" onevent="makeNormal()" />
<public:property name="hotColor" />
<public:method name="setHotColor" />
<script type="text/javascript">
   var oldColor, oldWeight;
   var hotColor = "purple";

   function setHotColor(color)
   {
      hotColor = color;
   }

   function makeHot()
   {
      if (event.srcElement == element)
      {
         oldColor = element.currentStyle.color;
```

```
         oldWeight = element.currentStyle.fontWeight;
         element.style.color = hotColor;
         element.style.fontWeight = "bold";
      }
   }

   function makeNormal()
   {
      if (event.srcElement == element)
      {
         element.style.color = oldColor;
         element.style.fontWeight = oldWeight;
      }
   }
</script>
```

To put the public information and the behavior, itself, to work, a demonstration page includes three spans within a paragraph that are associated with the behavior. Listing 51-4 shows the code for the demo page.

In addition to the text with rollover spans, the page contains two `select` controls, which let you assign a separate color to each of the three elements associated with the behavior. The first `select` element lets you choose one of the three elements. Making that choice invokes the `readColor()` function in the same page. This is the function that reads the `hotColor` public property of the chosen span. That color value is used to select the color name for display in the second `select` element. If you make a choice in the list of colors, the `applyVals()` function invokes the public `setHotColor()` method of the element currently selected from the list of elements. Rolling the mouse over that element now highlights it with the newly selected color, while the other elements maintain their current settings.

LISTING 51-4

Applying the Rollover Behavior

```
<!DOCTYPE html>
<html>
   <head>
      <title>IE Behavior Demo (Styles)</title>
      <style type="text/css">
         .hotStuff
         {
            font-weight:bold; behavior:url(makeHot.htc);
         }
      </style>
      <script type="text/javascript">
         function readColor(choice)
         {
            var currColor = document.all(choice.value).hotColor;
```

continued

LISTING 51-4 *(continued)*

```
            var colorList = choice.form.color;
            for (var i = 0; i < colorList.options.length; i++)
            {
               if (colorList.options[i].value == currColor)
               {
                  colorList.selectedIndex = i;
                  break;
               }
            }
         }
         function applyVals(form)
         {
            var elem = form.elem.value;
            document.all(elem).setHotColor(form.color.value);
         }
      </script>
   </head>
   <body>
      <h1>IE Behavior Demo (Styles)</h1>
      <hr />
      <form>
         Choose Element (the bolded text) to highlight when mousing over:
         <select name="elem" onchange="readColor(this)">
            <option value="elem1">First</option>
            <option value="elem2">Second</option>
            <option value="elem3">Third</option>
         </select>
         Choose Highlight Color:
         <select name="color" onchange="applyVals(this.form)">
            <option value="red" selected="selected">Red</option>
            <option value="blue">Blue</option>
            <option value="green">Green</option>
         </select>
      </form>
      <p>Lorem ipsum dolor sit amet,
         <span id="elem1" class="hotStuff">consectetaur</span>
         adipisicing elit, sed do eiusmod tempor incididunt ut
         <span id="elem2" class="hotStuff">labore et dolore magna aliqua</span>.
         Ut enim adminim veniam, quis nostrud exercitation ullamco laboris
         <span id="elem3" class="hotStuff">nisi ut aliquip ex ea commodo
             consequat</span>.
      </p>
   </body>
</html>
```

Behaviors are not the solution for every scripting requirement. As demonstrated here, they work very well for generic style manipulation, but you are certainly not limited to that sphere. By having a reference back to the element associated with the behavior, and then to the document that contains the element, a behavior's scripts can have free run over the page — provided the actions are either generic among any page, or generic among a design template that is used to build an entire web site or application.

Even if you don't elect to use behaviors now (perhaps because you must support browsers other than IE), they may be in your future. Behaviors are fun to think about, and using them instills good programming practice in the art of creating reusable, generalizable code.

For More Information

In addition to the address of W3C activity on behaviors listed at the beginning of the chapter, you can find much more about behaviors on Microsoft's developer site. Here are some useful pointers.

- Overview:
  `http://msdn.microsoft.com/en-us/library/ms531079%28VS.85%29.aspx`
- Using DHTML Behaviors:
  `http://msdn.microsoft.com/en-us/library/ms532147%28VS.85%29.aspx`
- Default Behaviors Reference:
  `http://msdn.microsoft.com/en-us/library/ms531081%28VS.85%29.aspx`
- Element Behaviors (an extension to the original behaviors):
  `http://msdn.microsoft.com/en-us/library/ms531426%28VS.85%29.aspx`

Each of these locations ends with yet more links to related pages at the Microsoft Developer Network (MSDN) web site.

Part VIII

Applications

IN THIS PART

Chapter 52
Application: Tables and Calendars

Chapter 53
Application: A Lookup Table

Chapter 54
Application: A ''Poor Man's'' Order Form

Chapter 55
Application: Outline-Style Table of Contents

Chapter 56
Application: Calculations and Graphics

Chapter 57
Application: Intelligent ''Updated'' Flags

Chapter 58
Application: Decision Helper

Chapter 59
Application: Cross-Browser DHTML Map Puzzle

Chapter 60
Application: Transforming XML Data

Chapter 61
Application: Creating Custom Google Maps

CHAPTER

52

Application: Tables and Calendars

IN THIS CHAPTER

Accommodating older browsers

Scripted tables

Date calculations

Working with HTML tables is a lot of fun, especially if you are not a born graphic designer. By adding a few tags to your page, you can make your columnar data look more organized, professional, and appealing. Having this power under scripting control is even more exciting, because it means that in response to a user action or other variable information (such as the current date or time), a script can do things to the table as the table is being built. Modern browsers allow scripts to modify the content and structure of a table even after the page has loaded, allowing the page to almost "dance."

You have three options when designing scripted tables for your pages, although only two are compatible with non-DHTML browsers (such as legacy browsers and some mobile browsers):

- Static tables
- Dynamic tables
- Dynamic HTML tables

The design path you choose is determined by whether you need to dynamically update some or all fields of a table (data inside `<td>...</td>` tags), and which browser levels you need to support. To highlight the differences among the three styles, this chapter traces the implementation of a monthly calendar display in all three formats.

About the Calendars

Because the emphasis here is on the way tables are scripted and displayed, we quickly pass over structural issues in the calendar versions described in the following sections. The first two examples are backward compatible to all

but the first-generation scriptable browsers. The final example, however, is a much more modern affair, utilizing table-related DOM objects and methods to simplify the code. It requires a modern browser (WinIE4+/Moz1+/W3C), and is definitely the desired approach unless you are specifically worried about legacy or mobile browser compatibility.

All three calendars follow similar (if not over-simplified) rules for displaying calendar data. English names of the months are coded into the script, so that they can be plugged into the calendar heading as needed. To make some of the other calendar calculations work (such as figuring out which day of the week is the first day of a given month in a given year), we define a method for the month objects. The method returns the JavaScript date object value for the day of the week of a month's first date. Virtually everything we do to implement the month objects is adapted from the custom objects discussion in Chapter 23, "Function Objects and Custom Objects."

Static Tables

The issue of updating the contents of a table's fields is tied to the nature of an HTML document being loaded and fixed in the browser's memory. Recall that for early browsers, you can modify precious few elements of a document and its objects after the document has loaded. This certainly applies for typical data points inside a table's `<td>` tag pair. After a document loads — even if JavaScript has written part of the page — none of its content (except for `text` and `textarea` field contents, and a few limited form element properties) can be modified without a complete reload.

Listing 52-1 contains the static version of a monthly calendar. The scripted table assembly begins in the `body` portion of the document. Figure 52-1 shows the results.

LISTING 52-1

A Static Table Generated by JavaScript

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>JavaScripted Static Table</title>
      <script type="text/javascript">
         // function becomes a method for each month object
         function getFirstDay(theYear, theMonth)
         {
            var firstDate = new Date(theYear,theMonth,1);
            return firstDate.getDay() + 1;
         }
         // number of days in the month
         function getMonthLen(theYear, theMonth)
         {
```

```
            var oneHour = 1000 * 60 * 60;
            var oneDay = oneHour * 24;
            var thisMonth = new Date(theYear, theMonth, 1);
            var nextMonth = new Date(theYear, theMonth + 1, 1);
            var len = Math.ceil((nextMonth.getTime() -
                      thisMonth.getTime() - oneHour)/oneDay);
            return len;
         }
         // create array of month names
         theMonths = new Array("January", "February", "March", "April",
                               "May", "June", "July", "August", "September",
                               "October", "November", "December");
      </script>
   </head>
   <body>
      <h1>A Static Scripted Month at a Glance</h1>
      <hr />
      <script type="text/javascript">
         // initialize some variables for later
         var today = new Date("4/11/1999");
         var theYear = today.getFullYear();
         var theMonth = today.getMonth(); // for index into our array

         // which is the first day of this month?
         var firstDay = getFirstDay(theYear, theMonth);
         // total number of <TD>...</TD> tags needed in for loop below
         var howMany = getMonthLen(theYear, theMonth) + firstDay;

         // start assembling HTML for table
         var content = "<center><table border='1'>";
         // month and year display at top of calendar
         content += "<tr><th colspan='7'>" + theMonths[theMonth];
         content += " " + theYear + "<\/th><\/tr>";
         // days of the week at head of each column
         content += "<tr><th>Sun<\/th><th>Mon<\/th><th>Tue<\/th><th>Wed<\/th>";
         content += "<th>Thu<\/th><th>Fri<\/th><th>Sat<\/th><\/tr>";
         content += "<tr>";

         // populate calendar
         for (var i = 1; i < howMany; i++)
         {
            if (i < firstDay)
            {
               // 'empty' boxes prior to first day
               content += "<td><\/td>";
            }
            else
            {
               // enter date number
               content += "<td align='center'>" + (i - firstDay + 1) + "<\/td>";
            }
```

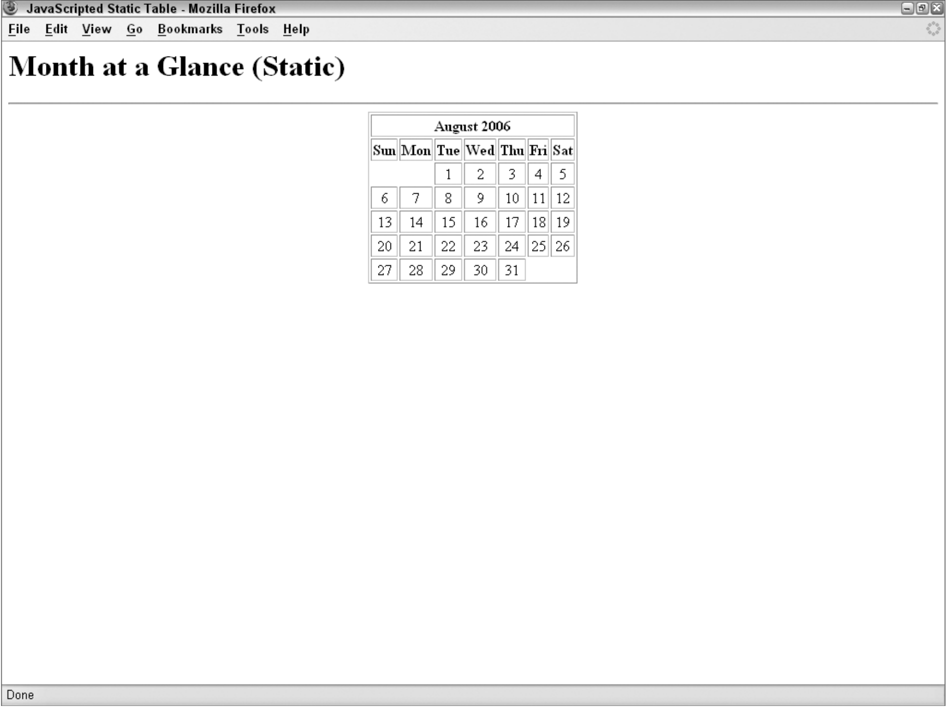
continued

LISTING 52-1 *(continued)*

```
            // start new row after each week
            if (i % 7 == 0 &&  i != howMany)
            {
               content += "<\/tr><tr>";
            }
         }
         content += "<\/table><\/center>";

         // blast entire table's HTML to the document
         document.write(content);
      </script>
   </body>
</html>
```

FIGURE 52-1

The static table calendar generated by Listing 52-1.



In this page, a little bit of the HTML — the `<h1>` heading and `<hr>` divider — is unscripted. The rest of the page consists entirely of the table definition, all of which is constructed in JavaScript. Most of the work for assembling the calendar's data points occurs inside the `for` loop. Because not every month starts on a Sunday, the script determines the day of the week on which the current month starts. For all fields prior to that day, the `for` loop writes empty `<td></td>` tags as placeholders. After the numbered days of the month begin, the `for` loop writes the date number inside the `<td>...</td>` tags. Whatever the script puts inside the tag pair is written to the page as flat HTML. Under script control like that in the example, however, the script can designate what goes into each data point — rather than writing fixed HTML for each month's calendar.

The important point to note in this example is that although the content of the page may change automatically over time (without having to redo any HTML for the next month), after the page is written, its contents cannot be changed. If you want to add controls or links that are to display another month or year, you have to rewrite the entire page. This can be accomplished by passing the desired month and year as a search string for the current page's URL, and then assigning the combination to the `location.href` property. You also have to add script statements to the page that look for a URL search string, extract the passed values, and use those values to generate the calendar while the page loads. (See Chapter 28, "Location and History Objects," for examples of how to accomplish this feat.) But to bring a calendar such as this even more to life (while avoiding page reloading between views), you can implement it as a dynamic table.

Dynamic Tables

The only way to make data points of a table dynamically updatable in backward-compatible browsers is to turn those data points into `text` (or `textarea`) objects. The approach to this implementation is different from the static table because it involves the combination of *immediate* and *deferred* scripting. Immediate scripting facilitates the building of the table framework, complete with fields for every modifiable location in the table. Deferred scripting enables users to make choices from other interface elements, causing a new set of variable data to appear in the table's fields.

Listing 52-2 turns the preceding static calendar into a dynamic one by including controls that enable the user to select a month and year to display in the table. As testament to the support for absolute backward compatibility, a button triggers the redrawing of the calendar contents, rather than `onchange` event handlers in the `select` elements; a bug in an older version of Netscape Navigator caused that event not to work for the `select` object. But don't worry, the last example you see a bit later in the chapter does away with backward compatibility in favor of a sleeker design.

Form controls aside, the look of this version is quite different from the static calendar. Compare the appearance of the dynamic version shown in Figure 52-2 against the static version in Figure 52-1.

LISTING 52-2

A Dynamic Calendar Table

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>JavaScripted Dynamic Table</title>
      <script type="text/javascript">
        // function becomes a method for each month object
        function getFirstDay(theYear, theMonth)
        {
          var firstDate = new Date(theYear,theMonth,1);
          return firstDate.getDay();
        }
        // number of days in the month
        function getMonthLen(theYear, theMonth)
        {
          var oneHour = 1000 * 60 * 60;
          var oneDay = oneHour * 24;
          var thisMonth = new Date(theYear, theMonth, 1);
          var nextMonth = new Date(theYear, theMonth + 1, 1);
          var len = Math.ceil((nextMonth.getTime() -
             thisMonth.getTime() - oneHour)/oneDay);
          return len;
        }
        // create array of month names
        theMonths = new Array("January", "February", "March", "April",
                   "May",  "June", "July", "August", "September",
                   "October", "November", "December");
        // deferred function to fill fields of table
        function populateFields(form)
        {
         // initialize variables for later from user selections
         var theMonth = form.chooseMonth.selectedIndex;
         var theYear = form.chooseYear.options
                      [form.chooseYear.selectedIndex].value;

         // initialize date-dependent variables
         // which is the first day of this month?
         var firstDay = getFirstDay(theYear, theMonth);
         // total number of <TD>...<\/TD> tags needed in for loop below
         var howMany = getMonthLen(theYear, theMonth);
          // set month and year in top field
         form.oneMonth.value = theMonths[theMonth] + " " + theYear;
         // fill fields of table
         for (var i = 0; i < 42; i++)
         {
             if (i < firstDay || i >= (howMany + firstDay))
             {
```

```
            // before and after actual dates, empty fields
            // address fields by name and [index] number
            form.oneDay[i].value = "";
         }
         else
         {
            // enter date values
            form.oneDay[i].value = i - firstDay + 1;
         }
      }
   }
  </script>
</head>
<body>
   <h1>Month at a Glance (Dynamic)</h1>
   <hr />
   <script type="text/javascript">
      // Initialize variable with HTML for each day's field.
      // All will have same name, so we can access via index value.
      // Empty event handler prevents
      //     reverse-loading bug in some platforms.
      var oneField = "<input type='text' name='oneDay' size='2' onfocus=''>";
      // Start assembling HTML for raw table:
      var content = "<form><center><table border='1'>";
      // Field for month and year display at top of calendar:
      content += "<tr><th colspan='7'>";
      content += "<input type='text' name='oneMonth'><\/th><\/tr>";
      // Days of the week at head of each column:
      content += "<tr><th>Sun<\/th><th>Mon<\/th><th>Tue<\/th><th>Wed<\/th>";
      content += "<th>Thu<\/th><th>Fri<\/th><th>Sat<\/th><\/tr>";
      content += "<tr>";

      // layout 6 rows of fields for worst-case month
      for (var i = 1; i < 43; i++)
      {
         content += "<td align='middle'>" + oneField + "<\/td>";
         if (i % 7 == 0)
         {
            content += "<\/tr><tr>";
         }
      }

      content += "<\/table>";
      // blast empty table to the document
      document.write(content);
   </script>
   <select name="chooseMonth">
      <option value="January" selected="selected">January</option>
      <option value="February">February</option>
      <option value="March">March</option>
```

continued

LISTING 52-2 *(continued)*

```
            <option value="April">April</option>
            <option value="May">May</option>
            <option value="June">June</option>
            <option value="July">July</option>
            <option value="August">August</option>
            <option value="September">September</option>
            <option value="October">October</option>
            <option value="November">November</option>
            <option value="December">December</option>
         </select>
         <select name="chooseYear">
            <option value="2008" selected="selected">2008</option>
            <option value="2009">2009</option>
            <option value="2010">2010</option>
            <option value="2011">2011</option>
            <option value="2012">2012</option>
            <option value="2013">2013</option>
            <option value="2014">2014</option>
            <option value="2015">2015</option>
         </select>
         <br />
         <input type="button" name="updater" value="Update Calendar"
                onclick="populateFields(this.form)" />
         </form>
      </body>
   </html>
```

Note

The property assignment event handling technique in this example is a deliberate simplification to make the code more readable. It is generally better to use the more modern approach of binding events using the `addEventListener()` **(NN6+/Moz/W3C) or** `attachEvent()` **(IE5+) methods. A modern cross-browser event handling technique is explained in detail in Chapter 32, "Event Objects." ■**

When you first load Listing 52-2, it creates an empty table, along with numerous text objects. An `onload` event handler in the `body` definition also could easily set the necessary items to load the current month.

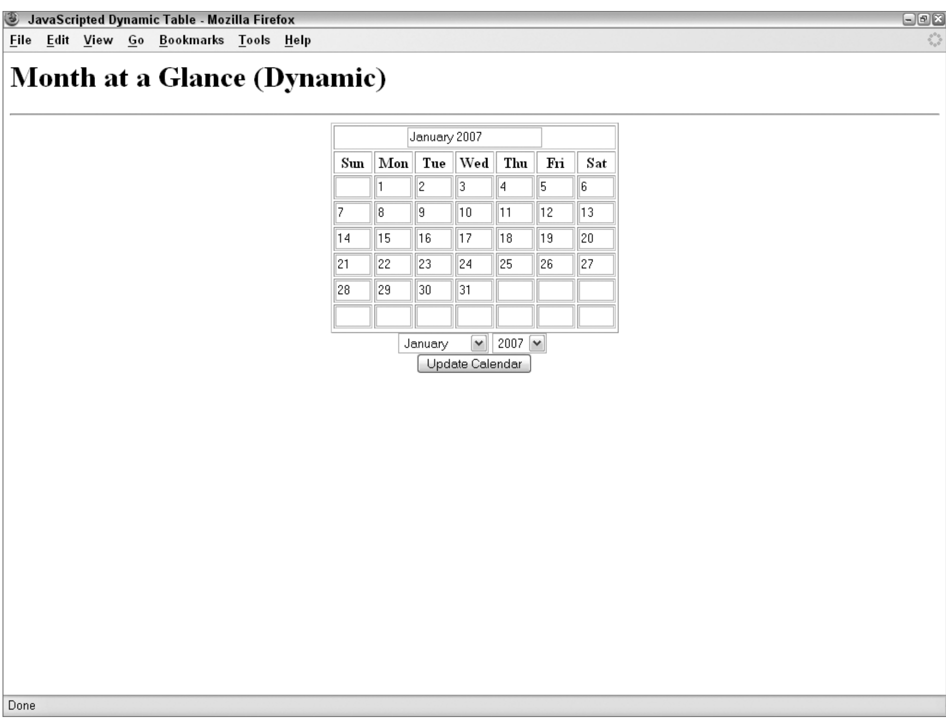
From a cosmetic point of view, the dynamic calendar may not be as pleasing as the static one in Figure 52-1. Several factors contribute to this appearance.

From a structural point of view, creating a table that can accommodate any possible layout of days and dates that a calendar may require is essential. That means a basic calendar consisting of six rows of fields. For many months, the last row remains completely empty. But because the table definition must be fixed when the page loads, this layout cannot change on the fly.

On the other hand, this version includes pop-up menus from which the user can select a month and year of choice. Clicking the Update Calendar button refills the calendar fields with data from the selected month.

FIGURE 52-2

Dynamic calendar generated by Listing 52-2.



A disadvantage to this dynamic table involves the fact that all text objects can be edited by the user. For many applications, this capability may not be a big deal. But if you're creating a table-based application that encourages users to enter values in some fields, be prepared (in other words, have event handlers in place) to either handle calculations based on changes to any field, or to alert users that the fields cannot be changed (and restore the correct value).

Hybrids

It will probably be the rare scripted table that is entirely dynamic. In fact, the one in Figure 52-2 is a hybrid of static and dynamic table definitions. The days of the week at the top of each column are hard-wired into the table as static elements. If your table design can accommodate both styles, implement your tables that way. The fewer the number of text objects defined for a page, the better the performance for rendering the page, and the less confusion for the page's users.

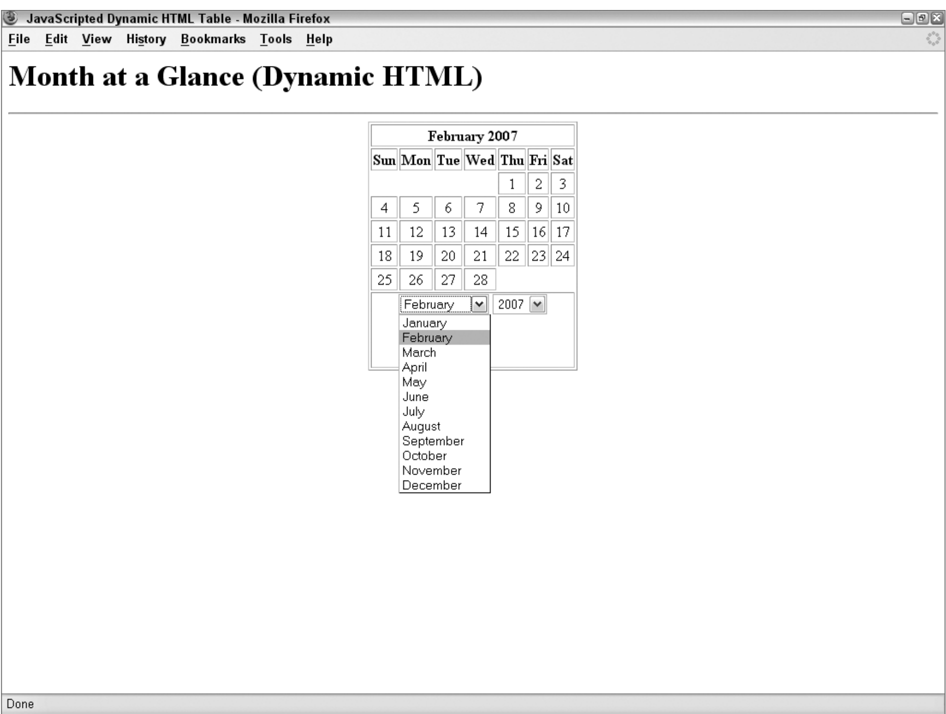
Dynamic HTML Tables

If you're committed to developing JavaScript code solely for modern browsers, which is entirely realistic at this point in time, you have all the resources of the `table` and related element objects, as described in Chapter 41, "Table and List Objects." The resulting application will appear to be much more polished, because not only does your content flow inside a table (which you can style to your heart's delight), but the content is dynamic within the table. The first two calendar examples were really just opening acts for the third and final version.

Listing 52-3 blends the calendar calculations from the earlier two calendar versions with the powers of the modern DOM (IE4+/Moz1+/W3C). A change to a requested calendar month or year instantly redraws the body of the table, without disturbing the rest of the page (see Figure 52-3).

FIGURE 52-3

DHTML table.



Basic date calculations are identical to the other two versions. Because this page has to be viewed with more modern browsers, we can use the shortcut array creation syntax for the month names array. Also, the way the table must be constructed each time is very different from two previous

versions. In this version, the script creates new table rows, creates new cells for those rows, and then populates those cells with the date numbers. Repeat loop logic is quite different, relying on a combination of `while` and `for` loops to get the job done.

Other features made possible by more modern browsers include automatic population of the list of available years. This page will never go out of style (unless browsers in 2050 no longer use JavaScript). There is also more automation in the triggers of the function that populates the table.

LISTING 52-3

Dynamic HTML Calendar

HTML: jsb52-03.html

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>JavaScripted Dynamic HTML Table</title>
      <style type="text/css">
         td, th
         {
            text-align:center;
         }
      </style>
      <script type="text/javascript" src="../jsb-global.js"></script>
      <script type="text/javascript" src="jsb52-03.js"></script>
   </head>
   <body>
      <h1>Month at a Glance (Dynamic HTML)</h1>
      <hr />
      <table id="calendarTable" border="1" align="center">
         <tr>
            <th id="tableHeader" name="tableHeader" colspan="7"></th>
         </tr>
         <tr>
            <th>Sun</th>
            <th>Mon</th>
            <th>Tue</th>
            <th>Wed</th>
            <th>Thu</th>
            <th>Fri</th>
            <th>Sat</th>
         </tr>
         <!-- identify tbody for the calendar dates -->
         <tbody id="tableBody" name="tableBody"></tbody>
         <tr>
            <td colspan="7">
               <form id="dateChooser" name="dateChooser">
                  <select id="chooseMonth" name="chooseMonth">
```

continued

LISTING 52-3 *(continued)*

```
                    <option value="January" selected="selected">January</option>
                    <option value="February">February</option>
                    <option value="March">March</option>
                    <option value="April">April</option>
                    <option value="May">May</option>
                    <option value="June">June</option>
                    <option value="July">July</option>
                    <option value="August">August</option>
                    <option value="September">September</option>
                    <option value="October">October</option>
                    <option value="November">November</option>
                    <option value="December">December</option>
                  </select>
                  <select id="chooseYear" name="chooseYear">
                  </select>
               </form>
               <br />
               <br />
            </td>
         </tr>
      </table>
   </body>
</html>
```

JavaScript: jsb52-03.js

```
// initialize when the page has loaded
addEvent(window, 'load', initialize);
// create array of English month names
var theMonths = ["January","February","March","April","May","June",
   "July","August","September","October","November","December"];
// global variables
   var oDateChooser;
   var oYearChooser;
   var oMonthChooser;
   var oTableHeader;
   var oTableBody;

/*******************
INITIALIZATIONS
*******************/
function initialize()
{
   // get IE4+ or W3C DOM reference for an ID
   // dateChooser is the form element
   if (document.getElementById)
   {
      oDateChooser = document.getElementById("dateChooser");
      oYearChooser = document.getElementById("chooseYear");
      oMonthChooser = document.getElementById("chooseMonth");
```

```
      oTableHeader = document.getElementById("tableHeader");
      oTableBody = document.getElementById("tableBody");
   }
   else
   {
      oDateChooser = document.dateChooser;
      oYearChooser = document.oDateChooser.chooseYear;
      oMonthChooser = document.oDateChooser.chooseMonth;
      oTableHeader = document.tableHeader;
      oTableBody = document.tableBody;
   }
   addEvent(oMonthChooser, 'change', populateTable);
   addEvent(oYearChooser, 'change', populateTable);

   // initialize year choices
   fillYears();
   // initiialize table
   populateTable(oDateChooser);

}

// create dynamic list of year choices
function fillYears()
{
   var today = new Date();
   var thisYear = today.getFullYear();
   for (i = thisYear; i < thisYear + 5; i++)
   {
      oYearChooser.options[oYearChooser.options.length] = new Option(i, i);
   }
   setCurrMonth(today);
}
// set month choice to current month
function setCurrMonth(today)
{
   oMonthChooser.selectedIndex = today.getMonth();
}

/*******************
UTILITY FUNCTIONS
********************/
// day of week of month's first day
function getFirstDay(theYear, theMonth)
{
   var firstDate = new Date(theYear,theMonth,1);
   return firstDate.getDay();
}
// number of days in the month
function getMonthLen(theYear, theMonth)
{
   var oneHour = 1000 * 60 * 60;
   var oneDay = oneHour * 24;
```

continued

LISTING 52-3 *(continued)*

```
   var thisMonth = new Date(theYear, theMonth, 1);
   var nextMonth = new Date(theYear, theMonth + 1, 1);
   var len = Math.ceil((nextMonth.getTime() -
                     thisMonth.getTime() - oneHour) / oneDay);
   return len;
}

/************************
DRAW CALENDAR CONTENTS
*************************/
// clear and re-populate table based on form's selections
function populateTable()
{
   var theMonth = oMonthChooser.selectedIndex;
   var theYear = parseInt(oYearChooser.options
                [oYearChooser.selectedIndex].value);
   // initialize date-dependent variables
   var firstDay = getFirstDay(theYear, theMonth);
   var howMany = getMonthLen(theYear, theMonth);

   // fill in month/year in table header
   while(oTableHeader.firstChild)
   {
      oTableHeader.removeChild(oTableHeader.firstChild);
   }
   oTableHeader.appendChild(document.createTextNode
      (theMonths[theMonth] + " " + theYear));

   // initialize vars for table creation
   var dayCounter = 1;
   // clear any existing rows
   while (oTableBody.rows.length > 0)
   {
      oTableBody.deleteRow(0);
   }
   var newR, newC;
   var done=false;
   while (!done)
   {
      // create new row at end
      newR = oTableBody.insertRow(oTableBody.rows.length);
      for (var i = 0; i < 7; i++)
      {
         // create new cell at end of row
         newC = newR.insertCell(newR.cells.length);
         if (oTableBody.rows.length == 1 && i < firstDay)
         {
            // no content for boxes before first day
            while(newC.firstChild)
```

```
            {
               newC.removeChild(newC.firstChild);
            }
            newC.appendChild(document.createTextNode(""));
            continue;
         }
         if (dayCounter == howMany)
         {
            // no more rows after this one
            done = true;
         }
         // plug in date (or empty for boxes after last day)
         while(newC.firstChild)
         {
            newC.removeChild(newC.firstChild);
         }
         newC.appendChild(document.createTextNode((dayCounter <= howMany)
            ? dayCounter++ : ""));
      }
   }
}
```

Further Thoughts

The best deployment of an interactive calendar requires the kind of Dynamic HTML currently available in modern DOMs. Moreover, the cells in those DOMs can receive mouse events, so that a user can click a cell and it will, perhaps, highlight in a different color or display some related, but otherwise hidden, information.

A logical application for such a dynamic calendar would be a pop-up window or frame that enables a user to select a date for entry into a form date field, such as in a travel web site where you select a date for travel through a calendar interface. It eliminates the need to type in a specific date format, thereby ensuring a valid date entry every time. For the sake of browsers with limited functionality (such as mobile and legacy), you can create a static version of the calendar that renders the numbers in the calendar cells as HTML links. Those links can use a `javascript:` URL to invoke a function call that sets a date field in the main form.

CHAPTER

53

Application: A Lookup Table

IN THIS CHAPTER

Server-less data collection lookup

Data-entry validation

One of the first ideas that intrigued me about JavaScript was the notion of delivering CGI-like functionality along with an HTML document. On the Web, numerous, small data collections currently require CGI scripting and a back-end database engine to drive them. Of course, not everyone who has information to share has access to the server environment (or the expertise) to implement such a solution. JavaScript provides that power.

A Server-Less Database

Before you get too carried away with the idea of letting JavaScript take the place of your SQL database, you need to recognize several limitations that prevent JavaScript from being a universal solution. First, any database that you embed into an HTML document is read-only. Although you can script an interface and lookup routines for the user, no provisions are available for writing revised information back to the server, if that is your intention.

A second consideration is the size of the data collection. Unlike databases residing on servers, the entire JavaScript database (or subset you define for inclusion into a single HTML document) must be downloaded to the user's browser before the user can work with the data. As a point of reference, think about image files. At 56.6 Kbps through a dial-up connection, how large an image file would you tolerate downloading? Granted, a vast majority of users now have broadband access of some kind, but that still doesn't mean that size has no relevance. Whatever that limit may be (based upon your bandwidth assumptions and how long you're willing to make users wait

for the database to download) is what your database size limit should be. For many special-purpose collections, you can get by with databases measured in the tens of kilobytes, assuming one byte per character. Unlike what happens when the user downloads an embedded image file, the user doesn't see special status bar messages about your database: to the browser, these messages are all part of the HTML coming in with the document.

The kind of data we're talking about here is obviously text data. That's not to say you can't let your JavaScript-enhanced document act as a front end to data files of other types on your server. The data in your embedded lookup table can be URLs to images that get swapped into the page as needed.

The Database

While thinking about writing a demonstration of a server-less database, we encountered a small article in the *Wall Street Journal* that related information we had always suspected. The Social Security numbers assigned to virtually every U.S. citizen are partially coded to indicate the state in which you registered for your Social Security number. This information often reveals the state in which you were born (another study indicates that two-thirds of U.S. citizens live their entire lives in the same state). The first three digits of the nine-digit number comprise this code.

When the numbering system was first established, each state was assigned a block of three-digit numbers. Therefore, if the first three digits fall within a certain range, the Social Security Administration has you listed as being registered in the corresponding state or territory. We thought this would be an interesting demonstration for a couple of reasons: first, the database is not that large, so it can be easily embedded into an HTML document without making the document too big to download, even on slow Internet connections; second, it offers some challenges to data-entry validation, as you will see in a moment.

Note

Before young people from populous states write to tell me that their numbers are not part of the database, let us emphasize that we are well aware that several states have been assigned number blocks not reflected in the database. This example is only a demonstration of scripting techniques, not an official Social Security Administration page. ■

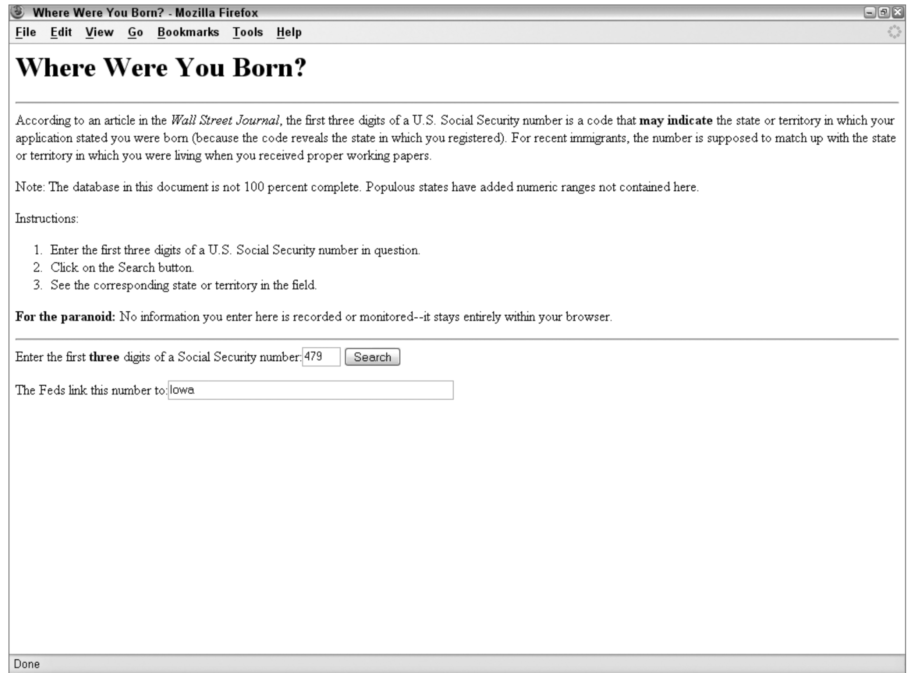
The Implementation Plan

For this demonstration, all we started with was a printed table of data. We figured that the user interface for this application would probably be very plain: a text field in which the user can enter a three-digit number, a clickable button to initiate the search, and a text

field to show the results of the lookup. Figure 53-1 shows the page. Pretty simple by any standards.

FIGURE 53-1

The Social Security number lookup page.



Given that user interface (we almost always start a design from the *interface* — how our page's users will experience the information presented on the page), we next planned the internals. We needed the equivalent of two tables: one for the numeric ranges, and one for the state names. Because most of the numeric ranges are contiguous, we could get by with a table of the high number of each range. This meant that the script would have to trap elsewhere for the occasional numbers that fall outside of the table's ranges — the job of data validation.

Because the two tables were so closely related to each other, we had the option of creating two separate arrays, so that any given index value would correspond to both the numeric and state name entries in both tables (*parallel arrays*, we call them). The other option was to create a two-dimensional array (see Chapter 18, "The Array Object"), in which each array entry has data points for both the number and state name. For purposes of demonstration to first-time database builders, we decided to stay with two parallel arrays. This method makes visualizing how the lookup process works with two separate arrays a little easier.

The Code

The HTML document starts normally through the definition of the document title:

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Where Were You Born?</title>
      <style text="text/css">
         .caution
         {
            font-weight:bold;
         }
      </style>
      <script type="text/javascript" src="../jsb-global.js"></script>
      <script type="text/javascript" src="jsb53-01.js"></script>
   </head>
```

In this application, the script is in a separate file. You'll notice that two files are identified, each in their own `script` element. The first is a global script file that contains the generic cross-browser, event-binding function described in Chapter 32. The second contains the JavaScript specific to this application. The balance of the HTML code is the Body part of the document. The real action takes place within the Form definition:

```
   <body>
      <h1>Where Were You Born?</h1>
      <hr />
      <p>According to an article in the <cite>Wall Street Journal</cite>,
         the first three digits of a U.S. Social Security number is a code
         that <span class="caution">may indicate</span> the state or
         territory in which your application stated you were born
         (because the code reveals the state in which you registered).
         For recent immigrants, the number is supposed to match up with
         the state or territory in which you were living when you received
         proper working papers.
      </p>
      <p>Note: The database in this document is not 100 percent complete.
         Populous states have added numeric ranges not contained here.</p>
      <p>Instructions:</p>
      <ol>
         <li>Enter the first three digits of a U.S. Social Security number
             in question.</li>
         <li>Click on the Search button.</li>
         <li>See the corresponding state or territory in the field.</li>
      </ol>
      <p><span class="caution">For the paranoid:</span> No information
         you enter here is recorded or monitored--it stays entirely
         within your browser.
      </p>
      <hr />
      <form id="searchForm" name="searchForm" onsubmit="return false;">
```

```
            <span>Enter the first <span class="caution">three</span> digits of
               any Social Security number:
            </span>
            <input type="text" id="entry" name="entry" size="3"
                   tabindex="1" />
            <input type="button" id="searchButton" value="Search"
                   tabindex="2" />
            <p>The Feds link this number to:
               <input type="text" id="result" name="result"
                      size="50" />
            </p>
         </form>
      </body>
   </html>
```

The form's `onsubmit` event handler is set to prevent accidental submission (or pseudo-submission, because no `action` attribute is specified for the form) that MacIE does from any form's text box (other browsers submit on Return from only a single-field form). Each of the text objects is sized to fit the expected data.

Now on to the JavaScript file that is specific to this application. In it you will see that we place utility function definitions close to the top of the script sections and put any action-oriented scripts (functions acting in response to event handlers) closer to the bottom of the script sections. My preference is to have all dependencies resolved before the script needs them. This philosophy carries over from the logic that dictates putting as many scripts in the Head or an external script as possible, so that even if the user (or network) should interrupt downloading of a page before every line of HTML reaches the browser, any user interface element relying on scripts will have those scripts loaded and ready to go. The order of functions in this example is not critical, because as long as they all reside in the Head section or an external script, they are defined and loaded by the time the field and button appear at the bottom of the page. But after we develop a style, we find it easier to stick with it — one less matter to worry about while scripting a complex application.

After creating an array (named `ssn`) with 57 empty slots, the script populates all 57 data points of the array, starting with the first entry going into the slot numbered 0. These data numbers correspond to the top end of each range in the 57-entry table. For example, any number greater than 3 but less than or equal to 7 falls into the range of the second data entry of the array (`ssn[1]`).

```
// create array listing all the top end of each numeric range
var ssn = new Array(57);
ssn[0] = 3;
ssn[1] = 7;
ssn[2] = 9;
ssn[3] = 34;
ssn[4] = 39;
ssn[5] = 49;
ssn[6] = 134;
ssn[7] = 158;
ssn[8] = 211;
ssn[9] = 220;
```

```
ssn[10] = 222;
ssn[11] = 231;
ssn[12] = 236;
ssn[13] = 246;
ssn[14] = 251;
ssn[15] = 260;
ssn[16] = 267;
ssn[17] = 302;
ssn[18] = 317;
ssn[19] = 361;
ssn[20] = 386;
ssn[21] = 399;
ssn[22] = 407;
ssn[23] = 415;
ssn[24] = 424;
ssn[25] = 428;
ssn[26] = 432;
ssn[27] = 439;
ssn[28] = 448;
ssn[29] = 467;
ssn[30] = 477;
ssn[31] = 485;
ssn[32] = 500;
ssn[33] = 502;
ssn[34] = 504;
ssn[35] = 508;
ssn[36] = 515;
ssn[37] = 517;
ssn[38] = 519;
ssn[39] = 520;
ssn[40] = 524;
ssn[41] = 525;
ssn[42] = 527;
ssn[43] = 529;
ssn[44] = 530;
ssn[45] = 539;
ssn[46] = 544;
ssn[47] = 573;
ssn[48] = 574;
ssn[49] = 576;
ssn[50] = 579;
ssn[51] = 580;
ssn[52] = 584;
ssn[53] = 585;
ssn[54] = 586;
ssn[55] = 599;
ssn[56] = 728;
```

We do the same for the array containing the states and territory names. Both of these array populators seem long but pale in comparison to what you would have to do with a database of many kilobytes. Unfortunately, JavaScript doesn't give you the power to load existing data files into arrays (but see the recommendations at the end of the chapter), so any time you want to embed a database into an HTML document, you must go through this array-style assignment frenzy:

```
// create parallel array listing all the states/territories
// that correspond to the top range values in the first array
var geo = new Array(57);
geo[0] = "New Hampshire";
geo[1] = "Maine";
geo[2] = "Vermont";
geo[3] = "Massachusetts";
geo[4] = "Rhode Island";
geo[5] = "Connecticut";
geo[6] = "New York";
geo[7] = "New Jersey";
geo[8] = "Pennsylvania";
geo[9] = "Maryland";
geo[10] = "Delaware";
geo[11] = "Virginia";
geo[12] = "West Virginia";
geo[13] = "North Carolina";
geo[14] = "South Carolina";
geo[15] = "Georgia";
geo[16] = "Florida";
geo[17] = "Ohio";
geo[18] = "Indiana";
geo[19] = "Illinois";
geo[20] = "Michigan";
geo[21] = "Wisconsin";
geo[22] = "Kentucky";
geo[23] = "Tennessee";
geo[24] = "Alabama";
geo[25] = "Mississippi";
geo[26] = "Arkansas";
geo[27] = "Louisiana";
geo[28] = "Oklahoma";
geo[29] = "Texas";
geo[30] = "Minnesota";
geo[31] = "Iowa";
geo[32] = "Missouri";
geo[33] = "North Dakota";
geo[34] = "South Dakota";
geo[35] = "Nebraska";
geo[36] = "Kansas";
geo[37] = "Montana";
geo[38] = "Idaho";
geo[39] = "Wyoming";
geo[40] = "Colorado";
geo[41] = "New Mexico";
geo[42] = "Arizona";
geo[43] = "Utah";
geo[44] = "Nevada";
geo[45] = "Washington";
geo[46] = "Oregon";
geo[47] = "California";
geo[48] = "Alaska";
geo[49] = "Hawaii";
```

```
geo[50] = "District of Columbia";
geo[51] = "Virgin Islands";
geo[52] = "Puerto Rico";
geo[53] = "New Mexico";
geo[54] = "Guam, American Samoa, N. Mariana Isl., Philippines";
geo[55] = "Puerto Rico";
geo[56] = "Long-time or retired railroad workers";
```

Then the `onload` event is bound to the `window` via the cross browser function `addEvent`. This is followed by the declaration of global variables:

```
// initialize when the page has loaded
addEvent(window, 'load', initialize);

// global variables
   var oEntry;
   var oResult;
   var oSearchButton;
```

Next, you'll see the initialization function that was bound to the `onload` event. First, the different objects the script will use are assigned to the global variables, and then the rest of the events are bound to the appropriate objects. As you'll recall from Chapters 26 and 32, the balance of the events are bound after the page has loaded to ensure the objects exist before event binding.

```
function initialize()
{
   // get IE4+ or W3C DOM reference for an element
   if (document.getElementById)
   {
      oEntry = document.getElementById("entry");
      oResult = document.getElementById("result");
      oSearchButton = document.getElementById("searchButton");
   }
   else
   {
      oEntry = document.searchForm.entry;
      oResult = document.searchForm.result;
      oSearchButton = document.searchForm.searchButton;
   }
   addEvent(oEntry, 'change', search);
   addEvent(oSearchButton, 'click', search);

}
```

Now comes the beginning of the data validation functions. Under control of a master validation function shown in a minute, the `stripZeros()` function removes any leading zeros that the user may have entered. Notice that the instructions tell the user to enter the first three digits of a Social Security number. For 001 through 099, that means the numbers begin with one or two zeros. JavaScript, however, treats any numeric value starting with 0 as an octal value. Because we have to do some numeric comparisons for the search through the `ssn[]` array, the script must make sure that the entries (which are strings to begin with, coming as they do from text objects)

can be converted to decimal numbers. The `parseInt()` function, with the all-important second parameter indicating Base 10 numbering, does the job. But because the remaining validations assume a string value, the integer is reconverted to a string value before it is returned:

```
// **BEGIN DATA VALIDATION FUNCTIONS**
// JavaScript sees numbers with leading zeros as octal values,
// so strip zeros
function stripZeros(inputStr)
{
   return parseInt(inputStr, 10).toString();
}
```

The next three functions are described in full in Chapter 46, which discusses data validation. In the last function, a copy of the input value is converted to an integer to enable the function to make necessary comparisons against the boundaries of acceptable ranges:

```
// general purpose function to see if an input value has been entered
function isEmpty(inputStr)
{
   if (inputStr == "" || inputStr == null)
  {
      return true;
   }
   return false;
}

// general purpose function to see if a suspected numeric input
// is a positive integer
function isNumber(inputStr)
{
   for (var i = 0; i < inputStr.length; i++)
   {
      var oneChar = inputStr.substring(i, i + 1);
         if (oneChar < "0" || oneChar > "9")
         {                  return false;
         }
   }
   return true;
}

// function to determine if value is in acceptable range
function inRange(inputStr)
{
   num = parseInt(inputStr);
   // Machines don't need parenthesis but humans do:
   if (num < 1 || (num > 586 && num < 596) || (num > 599 && num < 700)
        || num > 728)
   {
      return false;
   }
   return true;
}
```

The master validation controller function (named `isValid()` in this application) is also covered in depth in Chapter 46. A statement that wants to know if it should proceed with the lookup

process calls this function. If any one validation test fails, the function returns `false`, and the search does not proceed:

```
// Master value validator routine
function isValid(inputStr)
{
   var msg = " 3 digit positive number";
   var msgPrefix;
   var msgSuffix;
   if (isEmpty(inputStr))
   {
      msgPrefix = "Please enter a";
      msgSuffix = " into the field before clicking the button.";
      alert(msgPrefix + msg + msgSuffix);
      return false;
   }
   else
   {
      if (!isNumber(inputStr))
      {
         msgPrefix = "Please make sure entries are";
         msgSuffix = "s only.";
         alert(msgPrefix + msg + msgSuffix);
         return false;
      }
      else
      {
         if (!inRange(inputStr))
         {
            msgPrefix = "Sorry, the number you entered is not part
                         of our database. Try a";
            msgSuffix = ".";
            alert(msgPrefix + msg + msgSuffix);
            return false;
         }
      }
   }
   return true;
}      // **END DATA VALIDATION FUNCTIONS**
```

The `search()` function is invoked by two different event handlers. The two direct calls come from the input field's `onchange` event handler and the Search button's `onclick` event handler.

To search the database, the script repeatedly compares each succeeding entry of the `ssn[]` array against the value entered by the user. For this process to work, a little bit of preliminary work is needed. First comes an initialization of a variable, `foundMatch`, which comes into play later. Initially set to `false`, the variable is set to `true` only if there is a successful match — information you need later to set the value of the result text object correctly for all possible conditions:

```
function search()
{
   var foundMatch = false;
   var inputStr = stripZeros(oEntry.value);
```

```
            if (isValid(inputStr))
            {
               inputValue = inputStr;
               for (var i = 0; i < ssn.length; i++)
               {
                  if (inputValue <= ssn[i])
                  {
                     foundMatch = true;
                     break;
                  }
               }
            }
         oResult.value = (foundMatch) ? geo[i] : "";
         oEntry.focus();
         // delay so IE may select
         setTimeout("oEntry.select()", 100);
      }
```

Next comes all the data preparation. After the entry is passed through the zero stripper, a copy is dispatched to the master validation controller, which, in turn, sends copies to each of its special-purpose minions. If the master validator detects a problem from the results of any of those minions, it returns `false` to the condition that wants to know if the input value is valid. Should the value not be valid, processing skips past the `for` loop and proceeds immediately to an important sequence of three statements.

The first is a conditional statement that relies on the value of the `foundMatch` variable that was initialized at the start of this function. If `foundMatch` is still `false`, that means that something is wrong with the entry and it cannot be processed. To prevent any incorrect information from appearing in the result field, that field is set to an empty string if `foundMatch` is `false`. The next two statements set the focus and selection to the entry field, inviting the user to try another number. You'll recall in Chapter 26, "Generic HTML Element Objects," that IE sometimes fails to `select()` immediately after a `focus()` due to an internal timing problem, so `setTimeout()` adds a small delay to get around the problem.

On the other hand, if the entry is a valid number, the script finally gets to perform its lookup task. Looping through every entry of the `ssn[]` array starting with entry 0 and extending until the loop counter reaches the last item (based on the array's length property), the script compares the input value against each entry's value. If the number is less than or equal to a particular entry, the value of the loop counter (`i`) is frozen, the `foundMatch` variable is set to `true`, and execution breaks out of the `for` loop.

This time through the conditional expression, with `foundMatch` being `true`, the statement plugs the corresponding value of the `geo[]` array (using the frozen value of `i`) into the result field. Focus and selection are set to the entry field to make it easy to enter another value.

Further Thoughts

If we were doing this type of application for production purposes, we would turn each pairing of range high number and geographical location into separate objects and store the objects in an

array. Making that technique work requires one extra function and a different way of populating the data. The following is an example using the same variable names as the preceding listing:

```
// specify an array entry with two items
function dataRecord(ssn, geo)
{
   this.ssn = ssn;
   this.geo = geo;
   return this;
}

// initialize basic array
var numberState = new Array(57);

// populate main array with smaller arrays
numberState[0] = new dataRecord(3,"New Hampshire");
numberState[1] = new dataRecord(7,"Maine");
numberState[2] = new dataRecord(9,"Vermont");
```

The other changes (marked in boldface) occur in the `search()` function, which must address this data in a slightly different way than it did before:

```
function search()
{
   var foundMatch = false;
   var inputStr = stripZeros(oEntry.value);
      if (isValid(inputStr))
      {
         inputValue = inputStr;
         for (var i = 0; i < numberState.length; i++)
         {
            if (inputValue <= numberState[i].ssn)
            {
               foundMatch = true;
               break;
            }
         }
      }
   oResult.value = (foundMatch) ? numberState[i].geo : "";
   oEntry.focus();
   // delay so IE may select
   setTimeout("oEntry.select()", 100);
}
```

All references to data are to the `numberState[]` array and properties of its objects (either `ssn` or `geo`). With the data for each record arranged in a comma-delimited fashion, it may be easier to transfer data exported from an existing database to your script with less copying and pasting or dragging and dropping.

Another possibility would be to use JavaScript's capability to load `.js` files that have the arrays already populated or have variables preloaded with comma-delimited values. By using the

`string.split()` method (see Chapter 15, "The String Object"), you can easily assign data in this format to an array.

This application presents its results in a text box form control. But with today's browsers, you have the luxury of replacing segments of regular body content at will. Therefore, you could revise the script and HTML to include a placeholder element whose content is updated with the result of the array lookup.

We truly believe that server-less data lookups offer a great opportunity to many creative JavaScripters.

CHAPTER

54

Application: A Poor Man's Order Form

IN THIS CHAPTER

Live math on table rows and columns

Number formatting

Code reusability

We hesitate to call the application described in this chapter an order form because it is not in any way intended for use as a client-side shopping cart or some of the more advanced e-commerce applications you see on the Web. No, the goal here is to demonstrate how JavaScript can be used to assist users with column-and-row arithmetic, very much like the kinds of arithmetic needed to calculate the total for an order of goods.

Although this order form is not linked to any particular online catalog, some or all of it can be used as a piece for a small e-commerce site. The form in the example here requires that users input product descriptions and prices, but there is no reason that a client-side JavaScript shopping cart can't accumulate the shopper's choices from catalog pages, and then present them in an order form with product descriptions and prices hard-wired into the table. There still are entry boxes for quantity and selecting local sales tax rates. But all the arithmetic products and sums are calculated quickly on the client with JavaScript.

Along the way, you should also discover how to design code — more specifically, JavaScript data structures — in such a way that they are easily editable by non-scripters who are responsible for updating the embedded data. Therefore, even if you prefer to leave professional e-commerce order processing to server applications, you may still pick up a scripting tip or two from this "poor man's" version of an order form.

Defining the Task

We doubt that any two order forms on the Web are executed precisely the same way. Much of the difference has to do with the way an application on the server wants to receive the data on its way to an order-entry system or database. The rest has to do with how clever the HTML programmer is.

To come up with a generalized demonstration, we had to select a methodology and stay with it.

Because the intended goal of this demonstration is to focus on the rows and columns of an order form, we omit the usual name-and-address input elements. Instead, the code deals exclusively with the tabular part of the form, including the footer "stuff" of a form for subtotals, sales tax, shipping, and the grand total.

Another goal is to design the order form with an eye to as much reusability as possible. In other words, we may design the form for one page, but we also want to adapt it to another order form quickly without having to muck around too deeply in complicated HTML and JavaScript code. One giant annoyance that this approach eliminates is the normal HTML repetition of row after row of tags for input fields and table cells. JavaScript can certainly help you out there.

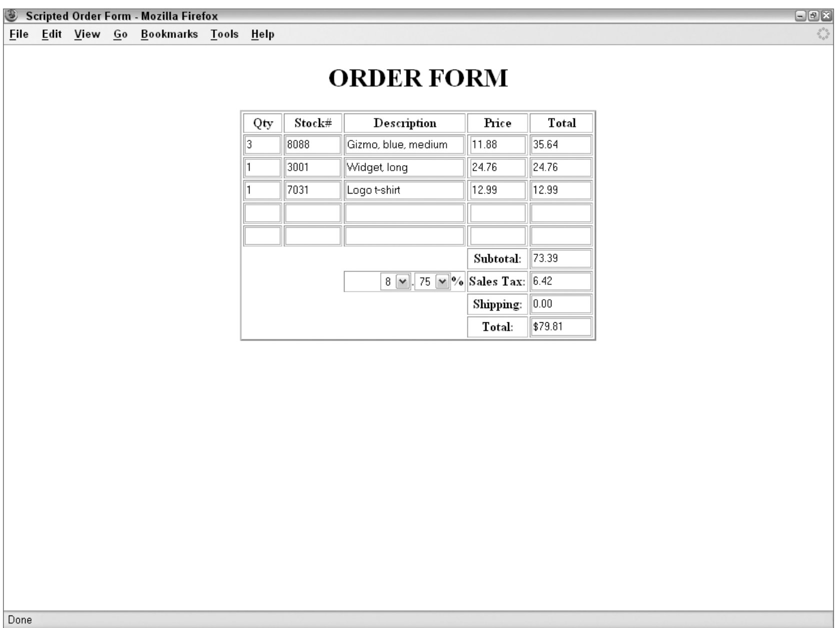
The order form code also demonstrates how to perform math and display results in two decimal places, use the `String.split()` method to make it easy to build arrays of data from comma-delimited lists, and enable JavaScript arrays to handle tons of repetitive work.

The Form Design

Figure 54-1 shows a rather simplified version of an order form as provided in the listings. Many elements of the form are readily adjustable by changing only a few characters near the top of the JavaScript listing. At the end of the chapter, we provide several suggestions for improving the user experience of a form, such as this one.

FIGURE 54-1

The order form display.



Form HTML and Scripting

Because this form is generated as the document loads, JavaScript writes most of the document to reflect the variable choices made in the reusable parts of the script. In fact, in this example, only the document heading is hard-wired in HTML.

The order form example page starts innocently enough:

```
<!DOCTYPE html>
<html>
  <head>
     <meta http-equiv="content-type" content="text/html;charset=utf-8">
<title>Scripted Order Form</title>
```

Global adjustments

The first important section of the example is the start of the JavaScript statements and functions that do most of the work. The script begins by initializing three very important global variables. This location is where the author, defining the details for the order form, also enters information about the column headings, column widths, and number of data entry rows:

```
<script type="text/javascript">
   // ** BEGIN GLOBAL ADJUSTMENTS ** //
   // Order form columns and rows specifications
   // **Column titles CANNOT CONTAIN PERIODS
   var columnHeads = "Qty,Stock#,Description,Price,Total".split(",");
   var columnWidths = "3,7,20,7,8".split(",");
   var numberOfRows = 5;
```

The first two assignment statements perform double duty. Not only do they provide the location for customized settings to be entered by the HTML author, but they use the `string.split()` method to literally create arrays out of their series of comma-delimited strings. At first, this may seem to be a roundabout way to generate an array, because you can also create the array directly with

```
var columnHeads = new Array("Qty","Stock",...);
```

But the way shown here minimizes the possibility of goofing up the quotes and commas when modifying the data, especially if modification might be attempted by a non-scripter.

So much of the repetitive work to come in this application is built around arrays that it will prove to be extraordinarily convenient to have the column title names and column widths in parallel arrays. The number-of-rows value also plays a role in not only drawing the form, but calculating it as well.

Notice the caveat about periods in column heading strings. You will soon see that these column names are assigned as text object names, which, in turn, are used to build object references to text boxes. Object names cannot have periods in them, so for these column headings to perform their jobs, you have to leave periods out of their names.

As part of the global adjustment area, the `extendRow()` method requires knowledge about which columns are to be multiplied to reach a total for any row:

```
// data entry row math
function extendRow(form,rowNum)
{
   // **change 'Qty' and 'Price' to match your column names
   var rowSum = form.Qty[rowNum].value * form.Price[rowNum].value;
   // **change 'Total' to match your corresponding column name
   form.Total[rowNum].value = rowSum.toFixed(2);
}
```

This example uses the `Qty`, `Price`, and `Total` fields for math calculations. Those field names are inserted into the references within this function. To calculate the total for each row, the function receives the form object reference and the row number as parameters. As described later, the order form is generated as a kind of array. Each field in a column intentionally has the same name. This scheme enables scripts to access a given field in that column by row number when using the row number as an index to the array of objects bearing the same name. For example, for the first row (row 0), you calculate the total by multiplying the quantity field of row 0 (`form.Qty[0].value`) times the price field of row 0 (`form.Price[0].value`). You then format that value to two places to the right of the decimal and plug that number into the value of the total field for row 0 (`form.Total[0].value`). More on the `number.toFixed()` method in a moment.

The final place where you have to worry about customized information is in the function that adds up the total columns. The function must know the name that you assigned to the total column:

```
function addTotals(form)
{
   var subTotal = 0;
   for (var i = 0; i < numberOfRows; i++)
   {
      // **change 'Total' in both spots to match your column name
      subTotal += (form.Total[i].value != "") ?
         parseFloat(form.Total[i].value) : 0;
   }
   form.subtotal.value = subTotal.toFixed(2);
   form.tax.value = (getTax(form,subTotal)).toFixed(2);
   form.total.value = "$"
                    + (parseFloat(form.subtotal.value)
                    + parseFloat(form.tax.value)
                    + parseFloat(form.shipping.value)).toFixed(2);
}
// ** END GLOBAL ADJUSTMENTS ** //
```

The `addTotals()` function receives the form reference as a parameter, which it uses to read and write data around the form. The first task is to add up the values of the total fields from each of the data-entry rows. Here you need to be specific about the name you assign to that column. To keep code lines to a minimum, you use a conditional expression inside the `for` loop to make additions to the `subTotal` amount only when a value appears in a row's total field. Because all

values from text fields are strings, you use `parseFloat()` to convert the values to floating-point numbers before adding them to the `subTotal` variable.

Three more assignment statements fill in the subtotal, tax, and total fields. The subtotal is nothing more than a formatted version of the amount reached at the end of the `for` loop. The task of calculating the sales tax is passed off to another function (described in a following section), but its value is also formatted before being plugged into the sales tax field. For the grand total, you add floating-point-converted values of the subtotal, tax, and shipping fields before slapping a dollar sign in front of the result. Even though the three fields contain values formatted to two decimal places, any subsequent math on such floating-point values incurs the minuscule errors that send formatting out to 16 decimal places. Thus, you must reformat the results after the addition.

Do the math

As you can see from Figure 54-1, the user interface for entering the sales tax is a pair of `select` elements. This type of interface minimizes the possibility of users entering the value in all kinds of weird formats that, in some cases, would be impossible to parse. The function that calculates the sales tax of the subtotal looks to these select objects for their current settings:

```
function getTax(form,amt)
{
   var chosenPercent = form.percent[form.percent.selectedIndex].value;
   var chosenFraction = form.fraction[form.fraction.selectedIndex].value;
   var rate = parseFloat(chosenPercent
                      + "."
                      + chosenFraction) / 100;
   return amt * rate;
}
```

After receiving the form object reference and subtotal amount as parameters, the function reads the two values chosen in the `select` elements. The string `value` properties of the `select` objects are temporarily stored in local variables. To arrive at the actual rate, you concatenate the two portions of the string (joined by an artificial decimal point) and `parseFloat()` the string to get a number that you can then divide by 100. The product of the subtotal times the rate is returned to the calling statement (in the preceding `addTotals()` function).

All the calculation that ripples through the order form is controlled by a single `calculate()` function:

```
function calculate(form,rowNum)
{
   extendRow(form,rowNum);
   addTotals(form);
}
```

This function is called by any object that affects the total of any row. Such a request includes both the form object reference and the row number. This information lets the single affected row, and then the totals column, be recalculated. Changes to some objects, such as the sales tax

select objects, affect only the totals column, so they will call addTotals() function directly rather than this function (the rows don't need recalculation).

Number formatting, as explained in Chapter 16, "The Math, Number, and Boolean Objects," is a detail that is addressed in the order form with relative ease thanks to the number.toFixed() method, which is thankfully now supported in modern browsers. In this case you need to trim down all monetary figures to exactly two decimal places. The number.toFixed() method offers precisely this functionality; you simply call the method on a number and pass along the number of fractional digits as the only argument. Here's an example:

```
total2D = total.toFixed(2);
```

You'll end up passing 2 to the toFixed() method for all monetary calculations that require two decimal places, but you may find it helpful to pass other values to the toFixed() method in other situations where fractional precision beyond two decimals is desired.

Cooking up some HTML

As we near the end of the scripting part of the document's Head section, we come to two functions that are invoked later to assemble some table-oriented HTML based on the global settings made at the top. One function assembles the row of the table that contains the column headings:

```
function makeTitleRow()
{
   var titleRow = "<tr>";
   for (var i = 0; i < columnHeads.length; i++)
   {
      titleRow += "<th>"
               + columnHeads[i]
               + "<\/th>";
   }
   titleRow += "<\/tr>";
   return titleRow;
}
```

The heart of the makeTitleRow() function is the for loop, which makes simple <th> tags out of the text entries in the columnHeads array defined earlier. All this function does is assemble the HTML. A document.write() method in the Body puts this HTML into the document.

```
function makeOneRow(rowNum)
{
   var oneRow = "<tr>";
   for (var i = 0; i < columnHeads.length; i++)
   {
      oneRow += "<td align=middle><input type=text size="
             + columnWidths[i]
             + " name=\'"
             + columnHeads[i]
```

```
                + "\' onchange='calculate(this.form,"
                + rowNum
                + ")'><\/td>";
      }
      oneRow += "<\/tr>";
      return oneRow;
   }
```

Creating a row of entry fields is a bit more complex, but not much. Instead of assigning just a word to each cell, you assemble an entire `<input>` object definition. You use the `columnWidths` array to define the size for each field (which therefore defines the width of the table cell in the column). `columnHead` values are assigned to the field's `name` attribute. Each column's fields have the same name, no matter how many rows exist. Finally, the `onchange` event handler invokes the `calculate()` method, passing the form and, most importantly, the row number, which comes into this function as a parameter (see the following section).

The next block of code closes up the head, opens up the body, and prepares the page for a table to hold the order form:

```
   </script>
</head>
<body>
   <center>
      <h1>ORDER FORM</h1>
      <form>
         <table border='2'>
```

Tedium lost

Believe it or not, all of the rows of data-entry fields in the table are defined by the handful of JavaScript statements that follow:

```
<script type="text/javascript">
   document.write(makeTitleRow());
   // order form entry rows
   for (var i = 0; i < numberOfRows; i++)
   {
      document.write(makeOneRow(i));
   }
```

The first function to be called is the `makeTitleRow()` function, which returns the HTML for the table's column headings. Then a very simple `for` loop writes as many rows of the field cells as defined in the global value near the top of the document. Notice how the index of the loop, which corresponds to the row number, is passed to the `makeOneRow()` function, so that it can assign that row number to its relevant statements. Therefore, these few statements generate as many entry rows as you need.

Tedium regained

What follows in the script writes the rest of the form to the screen. To make these fields as intelligent as possible, the scripts must take the number of columns into consideration. A number of empty-space cells must also be defined (again, calculated according to the number of columns). Finally, the code-consuming `select` element definitions must also be in this segment of the code:

```
document.write(makeTitleRow());
// order form entry rows
for (var i = 0; i < numberOfRows; i++)
{
   document.write(makeOneRow(i));
}

// order form footer stuff (subtotal, sales tax, shipping, total)
var colSpacer = "<tr><td colspan="
              + (columnWidths.length - 2)
              + "><\/td>";
document.write(colSpacer);
document.write("<th>Subtotal:<\/th>");
document.write("<td><input type=text size="
               + columnWidths[columnWidths.length - 1]
               + " name=subtotal><\/tr>");
document.write("<tr><td colspan="
               + (columnWidths.length - 3)
               + "><\/td>");
var tax1 = "<select name=percent onchange='addTotals(this.form)'>";
tax1 += "<option value=0>0";
tax1 += "<option value=1>1<option value=2>2<option value=3>3";
tax1 += "<option value=4>4<option value=5>5<option value=6>6";
tax1 += "<option value=7>7<option value=8>8<option value=9>9";
tax1 += "<\/select>";
var tax2 = "<select name=fraction onchange='addTotals(this.form)'>";
tax2 += "<option value=0>00<option value=25>25";
tax2 += "<option value=5>50<option value=75>75<\/select>";
document.write("<th align=right>"
               + tax1
               + "."
               + tax2
               + "\%<\/th>");
document.write("<th align=right>Sales Tax:<\/th>");
document.write("<td><input type=text size="
               + columnWidths[columnWidths.length - 1]
               + " name=tax value=0.00><\/tr>");
document.write(colSpacer);
document.write("<th>Shipping:<\/th>");
document.write("<td><input type=text size="
               + columnWidths[columnWidths.length - 1]
               + " name=shipping value=0.00 "
               + "onchange='addTotals(this.form)'><\/tr>");
document.write(colSpacer);
document.write("<th>Total:<\/th>");
```

```
                document.write("<td><input type=text size="
                               + columnWidths[columnWidths.length - 1]
                               + " name=total><\/tr>");
            </script>
            </table>
          </form>
       </center>
      </body>
   </html>
```

To gain a better understanding of how the script assembles the HTML for this part of the table, start by looking at the `colSpacer` variable. This variable contains a table cell definition that must span all but the rightmost two columns. Thus, the `colspan` attribute is calculated based on the length of the `columnWidths` array (minus two for the columns we need for data). Therefore, to write the line for the subtotal field, you start by writing one of these column spacers, followed by the `<th>` type of cell with the label in it. For the actual field, you must size it to match the fields for the rest of the column. That's why you summon the value of the last `columnWidths` value for the `size` attribute. You use similar machinations for the Shipping and Total lines of the form footer material.

In between these locations, you define the Sales Tax `select` objects (and a column spacer that is one cell narrower than the other one you used). To reduce the risk of data-entry error and to allow for a wide variety of values without needing a 40-item pop-up list, we divided the choices into two components and then display the decimal point and percentage symbol in hard copy. Both `select` objects trigger the `addTotals()` function to recalculate the rightmost column of the form.

Sometimes, it seems odd that you can script 4 lines of code to get 20 rows of a table, yet it takes 20 lines of code to get only 4 more complex rows of a table. Such are the incongruities of the JavaScripter's life.

Further Thoughts

Depending on the catalog of products or services being sold through this order form, the first improvement we would make is to automate the stock number and description entries. For example, if the list of all product numbers isn't that large, you may want to consider dropping a `select` element into each cell of the Description column. Then, after a user makes a selection, the `onchange` event handler performs a lookup through a product array and automatically plugs in the description and unit price. In any version of this form, you also need to perform data validation for crucial calculation fields, such as quantity.

Some of the other online order forms we've seen include reset buttons for every row or a column of checkmarks that lets users select one or more rows for deletion or resetting. Remember that people make mistakes and change their minds while ordering online. Give them plenty of opportunity to recover easily. If getting out of a jam is too much trouble, they will head for the History list or Back button, and that valued order will be, well, history.

CHAPTER

55

Application: Outline-Style Table of Contents

IN THIS CHAPTER

- Multiple frames
- Clickable images
- Persistent data
- Dynamic HTML positioning
- Reading XML data via Ajax

In your web surfing, you may have encountered sites that implement an expandable, outline-type table of contents. I've long thought that these elements were great ideas, especially for sites with lots of information. An outline, such as the Windows Explorer or the text-style Macintosh Finder windows, enables the author to present a large table of contents in a way that doesn't necessarily take up a ton of page space or bandwidth. From listings of top-level entries, a user can drill down to reveal only those items of interest.

No matter how much I like the idea, however, I dislike visiting most of these sites. Usually, a server program responds to each click, chews on my selection, and then sends back a completely new screen, showing my choice expanded or collapsed. After working with outlines in the operating system and outliner programs on personal computers, the delays in this processing seem interminable. It occurred to me that implementing the outline interface as a client-side JavaScript can significantly reduce the delay problem and make outlines a more viable interface to a site's table of contents. This chapter documents the process that went into a JavaScript-powered outliner, which works with all modern browsers.

The Implementation Plan

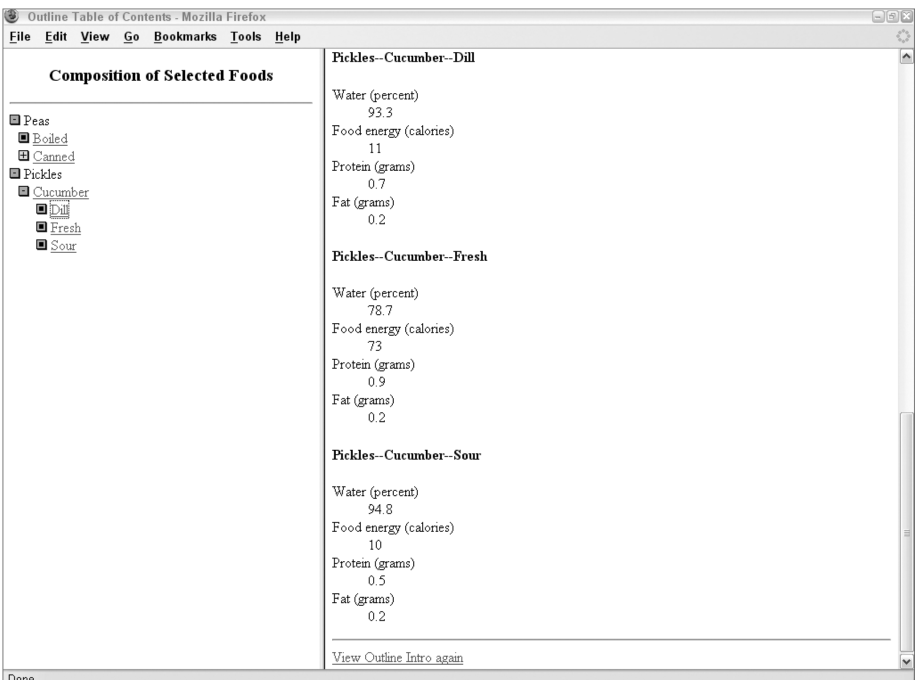
I admit to approaching the outline technique the first time without a specific data-display goal in mind — not always the best way to go about it. In search of some logical and public domain data that I could use as an example, I came upon the tables of information about food composition (grams of protein, fat, calories, and so on) published by the U.S. government. For this demonstration, I created one HTML document containing data for two hierarchical categories of foods: peas and pickles.

At the beginning of each food category, I assigned an anchor to which the text entries of the outline point.

My design for this implementation calls for two frames set up as columns (see Figure 55-1). The narrower left column houses the outline interface. After the frameset loads, the wider right frame initially shows an introductory HTML document. Clicking any of the links in the outline changes the view of the right-hand frame from the introductory document to the food data document. A link at the bottom of the food data document enables the user to view the introductory document again in the same frame, if desired.

FIGURE 55-1

The outline in the left frame is dynamic and local.



Earlier editions of this book included versions of this example that targeted legacy browsers, and were therefore limited in some ways. Modern browsers gave me reason to make some other improvements to the outliner over older versions. They include:

- Adjustable indentation spacing
- Easier specification of widget art files
- Easier way to specify a target frame for the results
- Additional array field for status bar display text

All adapter-adjustable elements appear near the top of the script to make it easy for scripters without a lot of experience to modify the application for their own sites.

The Code

All files for this implementation of the outline are on the CD-ROM accompanying this book, so I display here only the code for the framesetting document (`index.html`) and the outline (`toc.html`).

Setting the frames

To establish the frames, the HTML creates a two-column format, assigning 35 percent of the page as a column to contain the outline:

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
<title>Outline Table of Contents</title>
   </head>
   <frameset cols="35%,*">
      <noframes>
         <body>
            <h1>It's really cool...</h1>
            <h2>...but only if you have a frames-capable browser.</h2>
            <hr />
            <a href="index.html">Back</a>
         </body>
      </noframes>
      <frame name="Frame1" src="toc.html" />
      <frame name="Frame2" src="intro.html" />
   </frameset>
</html>
```

The names that I assign to the two frames aren't very original or clever, but they help me remember which frame is which. Because the nature of the contents of the second frame changes (either the introductory document or the data document), I couldn't think of a good name to reflect its purpose.

Outline code

Now we come to some lengthy code for the outline (in file `toc.html`). Much of the code deals with managing the binary representation of the current state of the outline. For each line of the completely exploded outline, the code designates a `0` for a line that has no nested items showing, and a `1` for a line that has a nested item showing. This sequence of `0`s and `1`s (as one string) is the road map that the script follows when redrawing the outline. Cues from the `0` and `1` settings let the script know whether it should display a nested item (if one exists) or leave that item collapsed.

To help me visualize the inner workings of these scripts, I developed a convention that calls any item with nested items beneath it a *mother*. Any nested item is that mother's *daughter*. A daughter can also be a mother if it has an item nested beneath it. You see how this plays out in the code shortly.

The food outline document starts out simply enough, with three CSS styles for the amount of indentation required by the three indentation levels of my sample outline. If the outline were to include more levels, you could simply add styles accordingly.

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Outline Style JavaScript TOC</title>
      <style type="text/css">
         .heading
         {
            text-align:center;
         }       div.indent0
         {
            margin-left:0px;
         }
         div.indent1
         {
            margin-left:10px;
         }
         div.indent2
         {
            margin-left:20px;
         }
      </style>
```

Scripting begins by setting up a global variable that keeps track of the state of the outline:

```
<script type="text/javascript">
   // global variables
   var CSScurrState;
```

The `toggle()` function is called by the `href` attribute of the links surrounding the widget icon art in the outline. This approach continues some of the legacy implementations, in which `img` elements were not capable of recognizing click events and required a link to handle the event (a version later in the chapter shows a more modern approach). A variable number of parameters are passed to this function, so that the parameters are extracted and analyzed through the `arguments` property of the function. Inside the large `for` loop, the setting of the `style.display` property is dynamically changed. The image of the toggled widget is then swapped. As a final touch, the window is given focus so that WinIE browsers lose the dotted rectangle around the clicked image:

```
// **function that updates persistent storage of state**
// toggles an outline mother entry, storing new value
function toggle()
```

```
{
   var newString = "";
   var expanded, n;
   // get all <DIV> tag objects in W3C DOMs
   var allDivs = document.getElementsByTagName("div");
   var currState = CSScurrState; // of whole outline
   // assemble new state string based on parameters passed from link
   for (var i = 0; i < arguments.length; i++)
   {
      n = arguments[i];
      expanded = currState.charAt(n); // of clicked item
      newString += currState.substring(0,n);
      newString += expanded ^ 1; // Bitwise XOR clicked item
      newString += currState.substring(n+1,currState.length);
      currState = newString;
      newString = "";
      // dynamically change display style without reloading
         if (expanded == "0")
         {
            allDivs[n].style.display = "block";
         }
         else
         {
            allDivs[n].style.display = "none";
         }
   }
   CSScurrState = currState; // store the new state
   // swap images in CSS versions
   var img = document.images["widget" + (arguments[0]-1)];
   img.src = (img.src.indexOf("plus.gif") != -1) ?
         "minus.gif" : "plus.gif";
   window.focus();
}
```

A prerequisite for loading the page to begin with is setting the initial value of the state. This is the only part of the script that must be hard-wired based on the structure of the outline — the string assigned to `initState` will be different with each outline. The goal here is to set each block assigned to the `indent0` style class to `1` while all others are set to `0`. These settings allow the first display of the outline to show all the root nodes, with all other items collapsed:

```
// initialize 'current state' storage field
if (!CSScurrState)
{
   // must be hard-wired to outline structure with "1" for
   // each indent0 class item, "0" for all others
   initState = "1000010000";
   CSScurrState = initState;
}
```

Initial settings of the `display` property can be done programmatically only after the document loads (the tags must exist before their properties can be adjusted). The following `init()`

function is called from the `onload` event handler. The `style.display` properties for hidden blocks are set to `none`:

```
            // initialize flagged tags to style display = "none"
            function init()
            {
               var visState = CSScurrState;
               for (var i = 0; i < visState.length; i++)
               {
                  if (visState.charAt(i) == "0")
                  {
                     document.getElementById("a" + i).style.display = "none";
                  }
               }
            }
         </script>
      </head>
      <body onload="init()">
         <div class="heading">
            <h3>Composition of Selected Foods</h3>
            <hr />
         </div>
```

The `init()` function reveals how the display status of outline entries is initially set. To find out if an entry should be displayed, the script performs a test to see if the visibility state of the entry is `0`.

Assembling outline content

Now begins the HTML that defines the content of the outline. For readability, I have formatted the `<div>` tag sets to follow the indentation of the outline data (this listing looks much better if you open the file from the CD-ROM in your text editor with word-wrap turned off). Each tag includes a `class` attribute pointing to a class defined in the first `<style>` tag of the page. Each tag also includes an `id` attribute whose name begins with the letter *a* and a sequential serial number, starting with zero. The `id` attributes are used to help assign `display` property settings during each reload.

Each outline entry includes an image (surrounded by a clickable link) and a text entry (which may or may not be a link to a document). The link around the image includes a `javascript:` URL for the `href` attribute. When a link is for a widget that is a mother item, the parameters to the `toggle()` function are the serial numbers (IDs) of the immediate children whose display properties are to be adjusted in the `toggle()` function. These passed items only need to be in the immediate children, because any of their children inherit the `display` property of their parents. For example, the first widget toggles items 1 and 2 (IDs `a1` and `a2`). Item 2 happens to be a parent to items 3 and 4. But when the `display` property of item 2 is set to `none`, none of its children (items 3 and 4) are displayed, no matter how their display properties are set.

`img` elements associated with each toggled `div` are named along similar lines, with the name starting with *widget* and the same serial number as the containing `div`. If you look at the end of the `toggle()` function again, you'll see that the name for the `img` element is derived from

the first parameter received by the `toggle()` function. That first parameter will always be one number higher than the serial number for the widget image to swap. To help you visualize the numbering scheme used within the example, the numbered identifiers and methods that relay associated numbers are shown in boldface:

```
<div class="indent0" id="a0">
   <a href="javascript:toggle(1,2)"
      onmouseover="status='Click to expand/collapse nested
                       items';return true"
      onmouseout="status='';return true"><img alt="image" name="widget0"
      src="plus.gif" height="12" width="12" border="0" /></a>
    Peas
   <br />
   <div class="indent1" id="a1">
      <a href="javascript:void(0)"
         onmouseover="status='No further items';return true"
         onmouseout="status='';return true"><img alt="image" src="end.gif"
         height="12" width="12" border="0" /></a>
       <a href="foods.html#boiled" target="Frame2">Boiled</a>
      <br />
   </div>
   <div class="indent1" id="a2">
      <a href="javascript:toggle(3,4)"
        onmouseover="status='Click to expand/collapse nested
                       items';return true"
        onmouseout="status='';return true"><img alt="image" name="widget2"
         src="plus.gif" height="12" width="12" border="0" /></a>
       <a href="foods.html#canned" target="Frame2">Canned</a>
      <br />
      <div class="indent2" id="a3">
         <a href="javascript:void(0)"
            onmouseover="status='No further items';return true"
            onmouseout="status='';return true"><img alt="image"
            src="end.gif" height="12" width="12" border="0" /></a>
          <a href="foods.html#alaska" target="Frame2">Alaska</a>
         <br />
      </div>
      <div class="indent2" id="a4">
        <a href="javascript:void(0)"
            onmouseover="status='No further items';return true"
            onmouseout="status='';return true"><img alt="image"
            src="end.gif" height="12" width="12" border="0" /></a>
         <a href="foods.html#losodium" target="Frame2">Low-Sodium</a>
        <br />
      </div>
   </div>
</div>
<div class="indent0" id="a5">
   <a href="javascript:toggle(6)"
      onmouseover="status='Click to expand/collapse nested
                       items';return true"
      onmouseout="status='';return true"><img alt="image" name="widget5"
```

```
            src="plus.gif" height="12" width="12" border="0" /></a>
          Pickles<br />
         <div class="indent1" id="a6">
            <a href="javascript:toggle(7,8,9)"
              onmouseover="status='Click to expand/collapse nested
                                    items';return true"
              onmouseout="status='';return true"><img alt="image" name="widget6"
              src="plus.gif" height="12" width="12" border="0" /></a>
               <a href="foods.html#cucumber" target="Frame2">Cucumber</a>
            <br />
            <div class="indent2" id="a7">
               <a href="javascript:void(0)"
                 onmouseover="status='Click to expand nested items';return true"
                 onmouseout="status='';return true"><img alt="image"
                 src="end.gif" height="12" width="12" border="0" /></a>
                <a href="foods.html#dill" target="Frame2">Dill</a>
               <br />
            </div>
            <div class="indent2" id="a8">
               <a href="javascript:void(0)"
                  onmouseover="status='No further items';return true"
                  onmouseout="status='';return true"><img alt="image"
                  src="end.gif" height="12" width="12" border="0" /></a>
                <a href="foods.html#fresh" target="Frame2">Fresh</a>
               <br />
            </div>
            <div class="indent2" id="a9">
               <a href="javascript:void(0)"
                  onmouseover="status='No further items';return true"
                  onmouseout="status='';return true"><img alt="image"
                  src="end.gif" height="12" width="12" border="0" /></a>
                <a href="foods.html#sour" target="Frame2">Sour</a>
               <br />
            </div>
         </div>
      </div>
   </body>
</html>
```

Note

The property assignment event handling technique used in this example and the next, is a deliberate simplification to make the code more readable. It is generally better to use the more modern approach of binding events using the `addEventListener()` (NN6+/Moz/W3C) or `attachEvent()` (IE5+) methods. A modern cross-browser event handling technique is explained in detail in Chapter 32, "Event Objects." ■

Note how the `href` attribute of outline entries is set to call the `toggle()` function, passing the number of the entry within the outline database. Perhaps more important is the use of the CSS `class` attribute to set the indentation of each outline entry. By tinkering with this example in a web browser, you should be able to figure out quickly how to modify the outline content to house your own unique outline data.

This example assumes that a site will be using only one outline-style table of contents. Of course, you can have multiple outlines for different sections of a web site or application. But if the outlines all share the same cookie data, the state of the most recent outline will be applied to the next one that loads. Items will be magically opened. And if the number of items between the two outlines is different, the cookie data can get a bit messy.

To solve this problem, assign a different cookie label for each outline. That prevents one outline's state from stepping on another.

An Ajax (XML) Outline

Most modern browsers are now capable of working directly with XML data. Therefore, I've modernized the scriptable outliner described earlier in this chapter to use the `XMLHttpRequest` object described in Chapter 39, "Ajax, E4X, and XML Objects." The data for this Ajax outline version is set apart in a more easily maintainable XML data file. At the same time, I've prettied up the hierarchical interface of the outliner display. Be aware that at the very least, you need to put the XML file on a server in order for the code in this section to render, as discussed later in this chapter.

Birth of an XML specification

Collapsible outlines provide convenient ways to organize hierarchical information all around us. You'd be hard-pressed to find a more active proponent of the outline than Dave Winer, founder of UserLand Software, Inc. (`www.userland.com`). Dave is a veteran software developer, as well as an author and outspoken web publisher. His `www.scripting.com` site is a popular destination if you want to find out the latest Internet and computing technology buzz.

As an outgrowth of development for his company's web tools, Dave looked to the XML structure to assist in representing outline content in a shareable, easily parseable format. The result is a specification called Outline Processor Markup Language, or OPML for short. You can read all about the formal specification at `http://www.opml.org/spec`. Like virtually all XML, OPML is intended to be written by software, not humans (although humans input the data via a user-friendly front-end provided by the software). Even so, the format of an OPML outline is extremely readable by humans, and, with little more trouble than it would take to write basic HTML tags manually, you can represent an outline in this format yourself.

A plain OPML file, saved as an `.xml` file, can be viewed through the native XML parsers of modern browsers such as IE5+ and NN6+/Moz1+. These parsers automatically render XML tags in the same hierarchical fashion in which OPML encourages outlines to be structured. But such rendering is under strict control of the browser, unless you also get involved with XML style sheets (the XSL and XSLT standards), at which point, browser implementation incompatibilities can make the going tough.

I liked the OPML data format when I first saw it, and I think it's a convenient way to convey an outline's data to the client, at which point JavaScript and the browser's DOM can take over to

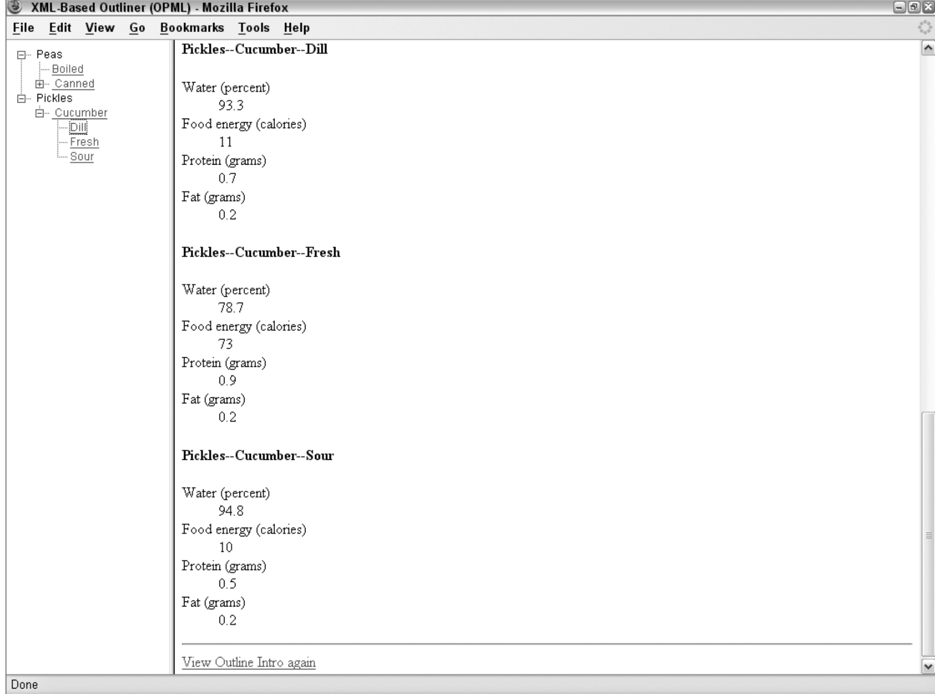
provide interesting visuals for, and interaction with, the content. Thus was born this last example of the chapter, in which the outliner's data is delivered, but not in the form of scripted arrays or hard-wired HTML `div` elements. Instead, the data arrives in its native XML (OPML) format from a separate file. Scripts take over to do the rest.

OPML outliner prep

The appearance of widgets and text for the new outliner has changed to more closely emulate the kinds of outline presentations that you see in some Windows programs (see Figure 55-2). For demonstration purposes, the same frameset structure and outline content from the earlier example is used for the OPML version so that you can more easily see the differences in implementations and grasp the new concepts presented here.

FIGURE 55-2

OPML-based outliner style.



Each OPML outline entry is nothing more than a tag. An entry that does not have any nested child nodes can use the XML shortcut of combining a start and end tag inside one set of angle brackets:

```
<tagName attribute="value" ... />
```

And any entry that has nested nodes contains the nested nodes between its start and end tags, as shown here, with the actual tag names used in OPML (indentation is optional, but increases readability and is therefore considered a best practice):

```
<outline text="text">
   <outline text="text" />
   <outline text="text" />
</outline>
```

If you want to associate more information about an entry, simply add an attribute. For example, if an entry is to behave as a link, you can convey that information with an attribute whose name you determine. When the time comes for your scripts to render the content in HTML, the scripts access the attribute values and generate the associated HTML for the attributes (you see an example of this in the code).

The true beauty of the OPML structure (and XML in general) is that the parent–child relationships are automatically implied by the element containment. The XML containment hierarchy implicitly specifies how many levels deep an entry is, and whether it has any child nodes. Suddenly, all of the W3C DOM gobbledygook about nodes, child nodes, and attributes becomes your friend, as your scripts convert the element hierarchy into a renderable hierarchy of your design.

The XML and HTML code

Because our focus is so tight on the outliner content, the first code to view is that of the XML document, named `outlineData.xml`:

```
<?xml version="1.0" ?>
<opml version="1.0">
   <head>
      <title>A Modern Outline</title>
      <dateCreated>Thu, 13 Nov 2003 02:40:00 GMT</dateCreated>
      <dateModified>Fri, 19 Dec 2003 19:35:00 GMT</dateModified>
      <ownerName>Danny Goodman</ownerName>
      <ownerEmail>dannyg@dannyg.com</ownerEmail>
      <expansionState></expansionState>
      <vertScrollState>1</vertScrollState>
      <windowTop></windowTop>
      <windowLeft></windowLeft>
      <windowBottom></windowBottom>
      <windowRight></windowRight>
   </head>
   <body>
      <outline text="Peas">
         <outline text="Boiled" uri="foods.html#boiled"/>
         <outline text="Canned" uri="foods.html#canned">
            <outline text="Alaska" uri="foods.html#alaska"/>
            <outline text="Low-Sodium" uri="foods.html#losodium"/>
         </outline>
      </outline>
```

```
         <outline text="Pickles">
            <outline text="Cucumber" uri="foods.html#cucumber">
               <outline text="Dill" uri="foods.html#dill"/>
               <outline text="Fresh" uri="foods.html#fresh"/>
               <outline text="Sour" uri="foods.html#sour"/>
            </outline>
         </outline>
      </body>
   </opml>
```

This file is a textbook OPML version 1.0 form. Notice that the OPML syntax reuses element names that are found in all HTML files (for example, `head`, `title`, `body`). This element name duplication makes it essential to isolate the XML from any HTML rendering area (such as an `iframe`), which would automatically add its own `body` element and thus destroy the OPML node tree.

The HTML body of the `toc.html` file is sparse, to say the least:

```
<body onload="init('outlineData.xml')">
   <div id="content"></div>
</body>
```

The only other HTML delivered in the document body is an empty `div` element, which is used as the container for the outline HTML that the scripts generate as a result of the `onload` event handler's invocation of the `init()` function.

Setting the scripted stage

All scripts for this page are in the `head` (although they could also be linked in from an external `.js` file). First on the docket is establishing several global variables that get used a lot within the rest of the code and make it easy to customize important visible properties, especially widget art. Due to the art choices made for this version, there are separate versions for items that appear as first, middle, and end items for different nesting states:

```
<script type="text/javascript">
   // global variables
   // art files and sizes for widget styles and spacers
   // (all images must have same height/width)
   var collapsedWidget = "oplus.gif";
   var collapsedWidgetStart = "oplusStart.gif";
   var collapsedWidgetEnd = "oplusEnd.gif";
   var expandedWidget = "ominus.gif";
   var expandedWidgetStart = "ominusStart.gif";
   var expandedWidgetEnd = "ominusEnd.gif";
   var nodeWidget = "onode.gif";
   var nodeWidgetEnd = "onodeEnd.gif";
   var emptySpace = "oempty.gif";
   var chainSpace = "ochain.gif";
   var widgetWidth = "20";
   var widgetHeight = "16";
```

```
var currState = "";
var displayTarget = "Frame2";
```

The `init()` function, invoked by the `onload` event handler, sets the content creation in motion. In actuality, all this function does is call the `loadXMLDoc()` function to handle the details of loading the XML document and building the outline:

```
// initialize first time
function init(outlineDataURL)
{
   loadXMLDoc(outlineDataURL);
}
```

Loading the XML file and populating the outline is performed in the `loadXMLDoc()` function. The function also handles alerts to users due to a missing XML file, connection problems, or lack of support for the feature:

```
// XML document
var xDoc;

// retrieve XML document as document object
function loadXMLDoc(url)
{
   var req = null;
   try
   {
      req = new XMLHttpRequest();
      req.overrideMimeType("text/xml");
   }
   catch(e)
   {
      req = new ActiveXObject("Msxml2.XMLHTTP");
   }
   if (req)
   {
      xDoc = null;
      req.open("GET", url, true);
      req.onreadystatechange = function()
         {
            if (req.readyState == 4)
            {
               if (req.status == 200)
               {
                xDoc = req.responseXML;
                if (xDoc &&
                    typeof xDoc.childNodes != "undefined" &&
                    xDoc.childNodes.length == 0)
                {
                   xDoc = null;
                }
                else
                {
```

```
                  // demo how to get outline head elements
                  var hdr = xDoc.getElementsByTagName("head")[0];
                  // get outline body elements for iteration
                  // and conversion to HTML
                  var ol = xDoc.getElementsByTagName("body")[0];
                  // wrap whole outline HTML in a span
                  var olHTML = "<span id='renderedOL'>"
                             + makeHTML(ol)
                             + "</span>";
                 // throw HTML into 'content' div for display
                 document.getElementById("content").innerHTML =
olHTML;

                 initExpand();
               }
            }
            else
            {
              alert("There was a problem retrieving the XML data:\n"
                    + req.statusText);
            }
         }
      }
     req.send(null);
   }
  }
```

The XMLHttpRequest object is available in IE as the Msxml2.XMLHTTP object, so the first thing you'll see is a try/catch block that instantiates the appropriate object. Both objects support the same properties and methods (see Chapter 39). If the XML file is not found on the server, the status property will not return a 200 (it will return a 0), and the responseXML property will be null. So, the code execution falls through with an error message to the user.

The loadXMLDoc() function uses the standard Ajax approach of establishing a callback function that waits for the XML document to be loaded, and then does something meaningful with it once the load has succeeded. The external XML data is loaded into a scripted document object global variable (named xDoc). If the data loads successfully, a reference to the Body portion of the outline data is retrieved so that many other functions are able to dive into the outliner hierarchy. The reference to the OPML body element is passed to the makeHTML() function, which returns the entire outline HTML, to be assigned to the innerHTML property of the empty div element, which is delivered with the document.

Note

As handy as it may be, in a strict W3C approach to JavaScript, you wouldn't use the innerHTML property since it isn't officially part of the W3C standard. However, it is often too powerful a convenience property to ignore, as much of the code throughout this book is a testament. The book does show the W3C node manipulation alternative to innerHTML in some examples. Refer to Chapter 29, "The Document and Body Objects," for a thorough explanation, and examples of the W3C alternative to innerHTML. ■

Accumulating the HTML

From the `loadXMLDoc()` function, a call to the `makeHTML()` function starts the most complex actions of the scripts on this page. This function walks the node hierarchy of the outline's `body` elements, deciphering which ones are containers and which ones are end points.

Two global variables are used to keep track of how far the node walk progresses, because this function calls itself from time to time to handle nested branches of the node tree. Because a reflexive call to a function starts out with new values for local variables, the global variables operate as pointers to let statements in the function know which node is being accessed. The numbers get applied to an `id` attribute assigned to the `div` elements holding the content.

One of the fine points of the design of this outline is the way space to the left of each entry is assembled. Unlike the earlier outlines in this chapter, this one displays vertical dotted lines connecting nodes at the same level. There isn't a vertical line for every clickable node appearing above the item, because a clickable node may have no additional siblings — meaning that the space is blank. To see what I mean, open the OPML example, and expand the Peas and Canned nodes (or refer to Figure 55-2). The Canned node is the end of the second column, so the space beneath the icon is blank. That's what some of the code in the `makeHTML()` named prefix is dealing with: Accumulating just the right combination of dotted line (`chain.gif`) and blank (`empty.gif`) images in sequence before the outline entry.

Another frequent construction throughout this function is a three-level conditional expression. This construction is used to determine whether the image just to the left of the item's text should be a start, middle, or end version of the image. The differences among them are subtle (having to do with how the vertical dotted line extends above or below the widgets). All of these decisions are made from information revealed by the inherent structure of the OPML element nesting. The listing in the book looks longer than it truly is because so many long or deeply nested lines must be wrapped to the next line. Viewing the actual file in your text editor should calm your fears a bit.

```
// counters for reflexive calls to makeHTML()
var currID = 0;
var blockID = 0;
// generate HTML for outline
function makeHTML(ol, prefix)
{
   var output = "";
   var nestCount, link, nestPrefix;
   prefix = (prefix) ? prefix : "";
   for (var i = 0; i < ol.childNodes.length ; i++)
   {
      if (ol.childNodes[i].nodeType != 1)
      {
         continue;
      }
      nestCount = ol.childNodes[i].childNodes.length;
      output += "<div class='row' id='line" + currID++ + "'>\n";
      if (nestCount > 0)
```

```
{
   output += prefix;
   output += "<img id='widget"
          + (currID-1)
          + "' src='"
          + ((i== ol.childNodes.length-1) ?
               collapsedWidgetEnd : (blockID==0) ?
                   collapsedWidgetStart : collapsedWidget);
   output += "' height="
          + widgetHeight
          + " width="
          + widgetWidth;
   output += " title='Click to expand/collapse nested items.'"
          + " onclick='toggle(this,"
          + blockID
          + ")'>";
   link =  (ol.childNodes[i].getAttribute("uri")) ?
               ol.childNodes[i].getAttribute("uri") : "";
   if (link)
   {
      output += " <a href='"
             + link
             + "' class='itemTitle' title='"
             + link
             + "' target='"
             + displayTarget
             + "'>";
   }
   else
   {
      output += " <a class='itemTitle' title='"
             + link
             + "'>";
   }
   output += " "
          + ol.childNodes[i].getAttribute("text")
          + "</a>";
   currState += calcBlockState(currID-1);
   output += "<span class='OLBlock' blocknum='"
          + blockID
          + "' id='OLBlock"
          + blockID++
          + "'>";
   nestPrefix = prefix;
   nestPrefix += (i == ol.childNodes.length - 1) ?
      "<img src='" + emptySpace + "' height=16 width=20>" :
      "<img src='" + chainSpace + "' height=16 width=20>";
   output += makeHTML(ol.childNodes[i], nestPrefix);
   output += "</span></div>\n";
}
else
{
   output += prefix;
```

```
            output += "<img id='widget"
                  + (currID-1)
                  + "' src='"
                  + ((i == ol.childNodes.length - 1) ?
                             nodeWidgetEnd : nodeWidget);
            output += "' height="
                  + widgetHeight
                  + " width="
                  + widgetWidth
                  + ">";
            link =  (ol.childNodes[i].getAttribute("uri")) ?
                     ol.childNodes[i].getAttribute("uri") : "";
            if (link)
            {
               output += " <a href='"
                     + link
                     + "' class='itemTitle' title='"
                     + link
                     + "' target='"
                     + displayTarget
                     + "'>";
            }
            else
            {
               output += " <a class='itemTitle' title='"
                     + link
                     + "'>";
            }
            output += ol.childNodes[i].getAttribute("text")
                  + "</a>";
            output += "</div>\n";
         }
      }
      return output;
   }
```

As with the HTML assembly code of the first outliner, if you were to add attributes to `outline` elements in an OPML outline (for example, a URL for an icon to display in front of the text), it is in `makeHTML()` that the values would be read and applied to the HTML being created.

The only other function invoked by the `makeHTML()` function is `calcBlockState()`. This function looks into one of the OPML outline's `head` elements, called `expansionstate`. This element's values can be set to a comma-delimited list of numbers, corresponding to nodes that are to be shown expanded when the outline is first displayed. The `calcBlockState()` function is invoked for each parent element. The element's location is compared against values in the `expansionstate` element, if there are any, and returns the appropriate `1` or `0` value for the state string being assembled for the rendered outline:

```
// apply default expansion state from outline's header
// info to the expanded state for one element to help
// initialize currState variable
function calcBlockState(n)
```

```
{
   var ol = xDoc.getElementsByTagName("body")[0];
   var outlineLen = ol.getElementsByTagName("outline").length;
   // get OPML expansionState data
   var expandElem = xDoc.getElementsByTagName("expansionState")[0];
   var expandedData = (expandElem.childNodes.length) ?
         expandElem.firstChild.nodeValue.split(",") : null;
   if (expandedData)
   {
      for (var j = 0; j < expandedData.length; j++)
      {
         if (n == expandedData[j] - 1)
         {
            return "1";
         }
      }
   }
   return "0";
}
```

The final act of the initialization process is a call to the `initExpand()` function. This function loops through the `currState` global variable (whose value was written in `makeHTML()` with the help of `calcBlockState()`) and sets the `display` property to `block` for any element designed to be expanded at the outset. HTML element construction in `makeHTML()` is performed in such a way that each parent `div` has a `span` nested directly inside of it; and inside that `span` are all the child nodes. The `display` property of the `span` determines whether all of those children are seen or not.

```
// expand items set in expansionState XML tag, if any
function initExpand()
{
   for (var i = 0; i < currState.length; i++)
   {
      if (currState.charAt(i) == 1)
      {
         document.getElementById("OLBlock" + i).style.display =
"block";
      }
   }
}
```

By the time the `initExpand()` function has run — a lot of setup code that executes pretty quickly — the rendered outline is in a steady state. Users can now expand or collapse portions by clicking the widget icons.

Toggling node expansion

All of the widget images in the outline have `onclick` event handlers assigned to them. The handlers invoke the `toggle()` function, passing parameters consisting of a reference to the `img`

element object receiving the event, and the serial number of the span block nested just inside the div that holds the image. With these two pieces of information, the toggle() function sets in motion the act of inverting the expanded/collapsed state of the element and the plus or minus version of the icon image. The blockNum parameter corresponds to the position within the currState string of 1s and 0s holding the flag for the expanded state of the block. With the current value retrieved from currState, the value is inverted through swapState(). Then, based on the new setting, the display property of the block is set accordingly, and widget art is changed through two special-purpose functions:

```
// toggle an outline mother entry, storing new state value;
// invoked by onclick event handlers of widget image elements
function toggle(img, blockNum)
{
   var newString = "";
   var expanded, n;
   // modify state string based on parameters from img
   expanded = currState.charAt(blockNum);
   currState = swapState(currState, expanded, blockNum);
   // dynamically change display style
   if (expanded == "0")
   {
     document.getElementById("OLBlock" + blockNum).style.display =
"block";
     img.src = getExpandedWidgetState(img.src);
   }
   else
   {
     document.getElementById("OLBlock" + blockNum).style.display = "none";
     img.src = getCollapsedWidgetState(img.src);
   }
}
```

Swapping the state of the currState variable utilizes the same XOR operator employed by the first outliner in this chapter. The entire currState string is passed as a parameter. The indicated digit is segregated and inverted, and the string is reassembled before being returned to the calling statement in toggle():

```
// invert state
function swapState(currState, currVal, n)
{
   var newState = currState.substring(0,n);
   newState += currVal ^ 1; // Bitwise XOR item n
   newState += currState.substring(n+1,currState.length);
   return newState;
}
```

As mentioned earlier, each of the clickable widget icons (plus and minus) can be one of three states, depending on whether the widget is at the start, middle, or end of a vertical-dotted chain. The two image-swapping functions find out (by inspecting the URLs of the images currently occupying the img element) which version is currently in place so that, for instance, a starting

"plus" (collapsed) widget is replaced with a starting "minus" (expanded) widget. This is a case of going the extra mile for the sake of an improved user interface:

```
          // retrieve matching version of 'minus' images
          function getExpandedWidgetState(imgURL)
          {
             if (imgURL.indexOf("Start") != -1)
             {
                return expandedWidgetStart;
             }
             if (imgURL.indexOf("End") != -1)
             {
                return expandedWidgetEnd;
             }
             return expandedWidget;
          }

          // retrieve matching version of 'plus' images
          function getCollapsedWidgetState(imgURL)
          {
             if (imgURL.indexOf("Start") != -1)
             {
                return collapsedWidgetStart;
             }
             if (imgURL.indexOf("End") != -1)
             {
                return collapsedWidgetEnd;
             }
             return collapsedWidget;
          }
        return collapsedWidget;
   }
```

Wrap up

There's no question that the amount and complexity of the code involved for the OPML version of the outliner are significant. The pain associated with developing an application such as this is all up front. After that, the outline content is easily modifiable in the OPML format (or perhaps by some future editor that produces OPML output).

Even if you don't plan to implement an OPML outline, the explanation of how this example works should drive home the importance of designing data structures that assist not only the visual design, but also the scripting that you use to manipulate the visual design.

Further Thoughts

The advent of CSS and element positioning has prompted numerous JavaScripters to develop another kind of hierarchical system of pop-up or drop-down menus that don't require as much scripting because of their reliance of CSS. You can find examples of this interface at many of the JavaScript source web sites listed in Appendix E.

Most of the code you find, however, will require a fair amount of tweaking to blend the functionality into the visual design that you have, or are planning, for your web site. Finding two implementations on the Web that look or behave the same way is rare. As long as you're aware of what you'll be getting yourself into, you are encouraged to check out these interface elements. By hiding menu choices except when needed, valuable screen real estate is preserved for more important static content.

CHAPTER

56

Application: Calculations and Graphics

IN THIS CHAPTER

Pre-cached images

Math calculations

CGI-like image assembly

When the scripting world had its first pre-release peeks at JavaScript (while Netscape was still calling the language LiveScript), the notion of creating interactive HTML-based calculators captured the imaginations of many page authors. Somewhere on the World Wide Web, you can find probably every kind of special-purpose calculation normally done by scientific calculators and personal computer programs — leaving only weather-modeling calculations to the supercomputers of the world.

In the search for my calculator gift to the JavaScript universe, I looked around for something more graphical. Numbers, by themselves, are pretty boring; so any way the math could be enlivened was fine by me. Having been an electronics hobbyist since I was a kid, I recalled the color-coding of electronic resistor components. The values of these gizmos aren't printed in plain numbers anywhere. You have to know the code and the meaning of the location of the colored bands to arrive at the value of each one. I think that this calculator will be fun to play with, even if you don't know what a resistor is.

The Calculation

To give you an appreciation for the calculation that goes into determining a resistor's value, here is the way the system works. Colored bands are printed on the surface of a resistor to encode its resistance value in ohms. The first (leftmost) band is the tens digit; the second (middle) band is the ones digit. Each color has a number from 0 through 9 assigned to it (black = 0, brown = 1, and so on). Therefore, if the first band is brown and the second band

is black, the number you start off with is 10. The third band is a multiplier. Each color determines the power of ten by which you multiply the first digits. For example, the red color corresponds to a multiplier of 10^2, so that $10 \infty 10^2$ equals 1,000 ohms.

A fourth band, if present, indicates the tolerance of the component — how far, plus or minus, the resistance measurement can fluctuate due to variations in the manufacturing process. Gold means a tolerance of plus-or-minus 5 percent; silver means plus-or-minus 10 percent; and no band means a 20 percent tolerance. A pinch of extra space typically appears between the main group of three-color bands and the one tolerance band.

User Interface Ideas

The quick-and-dirty, non-graphical approach for a user interface was to use a single frame with four `select` elements defined as pop-up menus (one for each of the four color bands on a resistor), a button to trigger calculation, and a field to show the numeric results.

How dull.

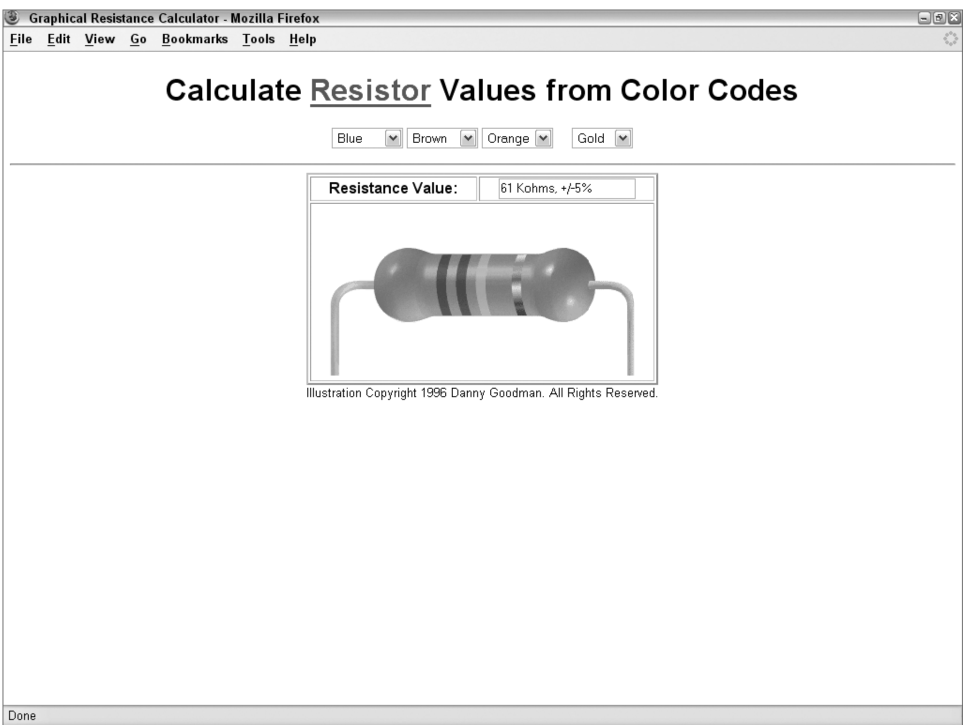
It occurred to me that if I design the art carefully, I can have JavaScript assemble an updated image of the resistor consisting of different slices of art: static images for the resistor's left and right ends, and variable slivers of color bands for the middle. Rather than use the brute force method of creating an image for every possible combination of colors (3,600 images total!), a far more efficient approach is to have one image file for each color (12 colors plus 1 empty) and enable JavaScript to call them from the server, as needed, in the proper order. If not for client-side JavaScript, a script on the server would have to handle this kind of intelligence and user interaction. But with this system, any dumb server can dish up the image files as called by the JavaScript script with no special server-side processing.

The first generation of this resistor calculator used two frames, primarily because I needed a second frame to update the calculator's art dynamically while keeping the pop-up color menus stationary. Images couldn't be swapped back in those frontier days, so the lower frame had to be redrawn for each color choice. Fortunately, modern browsers enabled me to update individual image objects in a loaded document without any document reloading. Moreover, with all the images precached in memory, page users experience no (or virtually no) delay in making a change from one value to another.

The design for the modern version is a simple, single-document interface (see Figure 56-1). Four pop-up menus let you match colors of a resistor, whereas the `onchange` event handler in each menu automatically triggers an image and calculation update. To hold the art together on the page, a table border surrounds the images on the page, whereas the numeric value of the component appears in a text field.

FIGURE 56-1

The Resistor Calculator with images inside a table border.



The Code

All the action takes place in the file named `resistor.html`. A second document is an introductory HTML text document that explains what a resistor is and why you need a calculator to determine a component's value. The article, called *The Path of Least Resistance*, can be viewed in a secondary window from a link in the main window. Here you will be looking only at `resistor.html`.

The document begins in the traditional way without any surprises other than a CSS style to change the font of the body text. Notice that the `onload` event is bound to the window using the `addEvent` function (as described in Chapter 32, "Event Objects").

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Graphical Resistance Calculator</title>
```

```
<style type="text/css">
   body
   {
      font-family:Arial, Helvetica, serif;
   }
   .copyright
   {
      font-size:small;
   }
</style>
<script type="text/javascript" src="../jsb-global.js"></script>
<script type="text/javascript">
   // initialize when the page has loaded
   addEvent(window, "load", calcOhms);
```

Basic arrays

In calculating the resistance, the script needs to know the multiplier value for each color. If not for the last two multiplier values actually being negative multipliers (for example, 10-1 and 10-2), I could have used the index values without having to create this array. But the two out-of-sequence values at the end mean that it's easier to work with an array than to try special-casing these instances in later calculations:

```
// create array listing all the multiplier values
var multiplier = new Array();
multiplier[0] = 0;
multiplier[1] = 1;
multiplier[2] = 2;
multiplier[3] = 3;
multiplier[4] = 4;
multiplier[5] = 5;
multiplier[6] = 6;
multiplier[7] = 7;
multiplier[8] = 8;
multiplier[9] = 9;
multiplier[10] = -1;
multiplier[11] = -2;

// create array listing all tolerance values
var tolerance = new Array();
tolerance[0] = "+/-5%";
tolerance[1] = "+/-10%";
tolerance[2] = "+/-20%";
```

Although the script doesn't do any calculations with the tolerance percentages, it needs to have the strings corresponding to each color for display in the pop-up menu. The `tolerance` array is there for that purpose. This is in contrast with the `multiplier` array, which contains numeric values instead of strings; the `multiplier` array is used to calculate the resistor value.

Calculations and formatting

Before the script displays the resistance value, it needs to format the numbers into values that are meaningful to those who know about these values. Just like measures of computer storage bytes, high quantities of ohms are preceded with "kilo" and "meg" prefixes, commonly abbreviated with the "K" and "M" letters. The `format()` function determines the order of magnitude of the final calculation (from another function shown in the following section) and formats the results with the proper unit of measure:

```
// format large values into kilo and meg
function format(ohmage)
{
   if (ohmage >= 1e6)
   {
      ohmage /= 1e6;
      return "" + ohmage + " Mohms";
   }
   else
   {
      if (ohmage >= 1e3)
      {
         ohmage /= 1e3;
         return "" + ohmage + " Kohms";
      }
      else
      {
         return "" + ohmage + " ohms";
      }
   }
}
```

The selections from the pop-up menus meet the calculation formulas of resistors in the `calcOhms()` function. Because this function is triggered indirectly by each of the `select` objects, sending any of their parameters to the function is a waste of effort. Moreover, the `calcOhms()` function is invoked by the `onload` event handler, which is not tied to the form or its controls. Therefore, the function obtains the reference to the form and then extracts the necessary values of the four `select` objects by using explicit (named) references. Each value is stored in a local variable for convenience in completing the ensuing calculation.

Recalling the rules used to calculate values of the resistor bands, the first statement of the calculation multiplies the "tens" pop-up value times 10 to determine the tens digit, and then adds the ones digit. From there, the combined value is multiplied by the exponent value of the selected multiplier value. Notice that the expression first assembles the value as a string to concatenate the exponent factor and then evaluates it to a number. Although I try to avoid the `eval()` function because of its slow performance, the one call per calculation is not a performance issue. The evaluated number is passed to the `format()` function for proper formatting (and setting of order of magnitude). In the meantime, the tolerance value is extracted from its array, and the

combined string is plugged into the `result` text field (which is in a separate form, as described later):

```
// calculate resistance and tolerance values
function calcOhms()
{
   var form = document.forms["input"];
   var d1 = form.tensSelect.selectedIndex;
   var d2 = form.onesSelect.selectedIndex;
   var m = form.multiplierSelect.selectedIndex;
   var t = form.toleranceSelect.selectedIndex;
   var ohmage = (d1 * 10) + d2;
   ohmage = eval("" + ohmage + "e" + multiplier[m]);
   ohmage = format(ohmage);
   var tol = tolerance[t];
   document.forms["output"].result.value = ohmage + ", " + tol;
}
```

Preloading images

As part of the script that runs when the document loads, the next group of statements precaches all possible images that can be displayed for the resistor art. For added scripting convenience, the color names are loaded into an array. With the help of that just-created array of color names, you then create another array (`imageDB`), which both generates `Image` objects for each image file, and assigns a URL to each image. Notice an important subtlety about the index values being used to create each entry of the `imageDB` array: Each index is a `colorArray` entry, which is the name of the color. As you found out in Chapter 18, "The Array Object," if you create an array element with a named index, you must use that style of index to retrieve the data thereafter; you cannot switch arbitrarily between numeric indexes and names. As you see in a moment, this named index provides a critical link between the choices a user makes in the pop-up lists and the image objects being updated with the proper image file.

```
// pre-load all color images into image cache
var colorArray = new Array("Black","Blue","Brown","Gold","Gray",
   "Green","None","Orange","Red","Silver","Violet","White","Yellow");
var imageDB = new Array();
for (i = 0; i < colorArray.length; i++)
{
   imageDB[colorArray[i]] = new Image(21,182);
   imageDB[colorArray[i]].src = colorArray[i] + ".gif";
}
```

The act of assigning a URL to the `src` property of an `Image` object instructs the browser to pre-load the image file into memory. This preloading happens as the document is loading, so another few seconds of delay won't be noticed by the user.

Changing images on-the-fly

The next four functions are invoked by their respective `select` object's `onchange` event handler. For example, after a user makes a new choice in the first `select` object (the "tens" value

color selector), that `select` object reference is passed to the `setTens()` function. Its job is to extract the text of the choice and use that text as the index into the `imageDB` array. Alternatively, the color name can also be assigned to the `value` attribute of each `option`, and the `value` property read here. You need this connection to assign the `src` property of that array entry to the `src` property of the image that you see on the page (defined in the following section). This assignment is what enables the images of the resistor to be updated instantaneously — just the one image "slice" affected by the user's choice in a `select` object:

```
function setTens(choice)
{
   var tensColor = choice.options[choice.selectedIndex].value;
   document.getElementById("tens").src = imageDB[tensColor].src;
   calcOhms();
}
function setOnes(choice)
{
   var onesColor = choice.options[choice.selectedIndex].value;
   document.getElementById("ones").src = imageDB[onesColor].src;
   calcOhms();
}
function setMult(choice)
{
   var multColor = choice.options[choice.selectedIndex].value;
   document.getElementById("mult").src = imageDB[multColor].src;
   calcOhms();
}
function setTol(choice)
{
   var tolColor = choice.options[choice.selectedIndex].value;
   document.getElementById("tol").src = imageDB[tolColor].src;
   calcOhms();
}
```

The rest of the script for the Head portion of the document merely provides the statements that open the secondary window to display the introductory document:

```
        function showIntro()
        {
            window.open("resintro.htm","","width=400,height=320,
                left=100,top=100");
        }
      </script>
   </head>
```

Creating the select objects

A comparatively lengthy part of the document is consumed with the HTML for the four `select` objects. Remember that the `onload` event was bound to the window object at the beginning of the script. This handler calculates the results of the currently selected choices whenever the document loads or reloads. If it weren't for this event handler, you would not see the resistor

art when the document first appears. Also, because many browsers maintain their form controls' settings while the page is in history, a return to the page later must display the images previously selected in the form:

```
<body>
   <center>
      <h1>Calculate
         <a href="javascript:showIntro()"
            onmouseover="status='An intro to the resistor electronic
                               component...';return true">
            Resistor
         </a> Values from Color Codes         <form name="input">
         <select name="tensSelect" onchange="setTens(this)">
            <option value="Black" selected="selected">Black</option>
            <option value="Brown">Brown</option>
            <option value="Red">Red</option>
            <option value="Orange">Orange</option>
            <option value="Yellow">Yellow</option>
            <option value="Green">Green</option>
            <option value="Blue">Blue</option>
            <option value="Violet">Violet</option>
            <option value="Gray">Gray</option>
            <option value="White">White</option>
         </select>
         <select name="onesSelect" onchange="setOnes(this)">
            <option value="Black" selected="selected">Black</option>
            <option value="Brown">Brown</option>
            <option value="Red">Red</option>
            <option value="Orange">Orange</option>
            <option value="Yellow">Yellow</option>
            <option value="Green">Green</option>
            <option value="Blue">Blue</option>
            <option value="Violet">Violet</option>
            <option value="Gray">Gray</option>
            <option value="White">White</option>
         </select>
         <select name="multiplierSelect" onchange="setMult(this)">
            <option value="Black" selected="selected">Black</option>
            <option value="Brown">Brown</option>
            <option value="Red">Red</option>
            <option value="Orange">Orange</option>
            <option value="Yellow">Yellow</option>
            <option value="Green">Green</option>
            <option value="Blue">Blue</option>
            <option value="Violet">Violet</option>
            <option value="Gray">Gray</option>
            <option value="White">White</option>
            <option value="Gold">Gold</option>
            <option value="Silver">Silver</option>
         </select>
             
         <select name="toleranceSelect" onchange="setTol(this)">
            <option value="Gold" selected="selected">Gold</option>
```

```
      <option value="Silver">Silver</option>
      <option value="None">None</option>
   </select>
</form>
<hr />
```

Note

The property assignment event handling technique in this example is a deliberate simplification to make the code more readable. It is generally better to use the more modern approach of binding events using the `addEventListener()` (NN6+/Moz/W3C) or `attachEvent()` (IE5+) methods, as was done earlier in the code for the `onload` event. The modern cross-browser event handling technique used earlier is explained in detail in Chapter 32. ■

Drawing the initial images

The balance of the document, mostly in JavaScript, is devoted to creating the table and image objects whose `src` properties will be modified with each choice of a `select` object. The act of assembling the HTML for the image table occurs right after the `select` objects have rendered. References to those `select` elements are required in order to extract the currently selected values. If the `form` element that holds the `select` elements is not closed, you can't build a valid reference to the `select` elements. Therefore, the page contains two forms: One for the `select` elements; one for the output text box inside the table:

```
<script type="text/javascript">
   var form = document.forms["input"];
   var tensDigit = form.tensSelect.selectedIndex;
   var tensColor = form.tensSelect.options[tensDigit].value;
   var onesDigit = form.onesSelect.selectedIndex;
   var onesColor = form.onesSelect.options[onesDigit].value;
   var multDigit = form.multiplierSelect.selectedIndex;
   var multColor = form.multiplierSelect.options[multDigit].value;
   var tolDigit = form.toleranceSelect.selectedIndex;
   var tolColor = form.toleranceSelect.options[tolDigit].value;

   var table ="<table border='2'><form name='output'>";
   table += "<tr><th align='middle'>Resistance Value:<\/th>";
   table +=" <th align='middle'>";
   table +="<input type='text' name='result' size=20
            onfocus='this.blur()'>";
   table +="<\/th><\/tr><tr><td colspan=2> ";
   table +="<img SRC='resleft.gif' width=127 height=182><img"
         + " SRC='" + tensColor
         + ".gif' id='tens' width=21 height=182><img"
         + " SRC='" + onesColor
         + ".gif' id='ones' width=21 height=182><img"
         + " SRC='" + multColor
         + ".gif' id='mult' width=21 height=182><img"
         + " SRC='spacer.gif' width=17 height=182><img"
         + " SRC='" + tolColor
         + ".gif' id='tol' width=21 height=182><img"
         + " SRC='resright.gif' width=127 height=182>";
```

```
            table += " <\/td><\/tr><\/form><\/table>";
            document.write(table);
         </script>
         <div class="copyright">Illustration Copyright 1996 Danny Goodman.
            All Rights Reserved.
         </div>
      </center>
   </body>
</html>
```

As you can see, the resistor images appear in one table cell (in the second row) that contains all seven image objects smashed against each other. To keep the images flush, there can be no spaces or carriage returns between `<img>` tags.

Further Thoughts

Although this example works quite well, annoying differences still exist among different platforms. At one point in the design process, I considered trying to align the pop-up menus with images of the resistor (or callout line images), but the differences in platform rendering of pop-up menus made that idea impractical. At best, I now separate the three left `select` objects from the right one by way of hard-coded spaces (` `).

You should notice from this exercise that I look for ways to blend JavaScript object data structures with my own data's structure. For example, the `select` objects serve multiple duties in these scripts. Not only does the text of each option point to an image file of the same name, but the index values of the same options are applied to the calculations. Things don't always work out that nicely, but whenever your scripts bring together user interface elements and data elements, look for algorithmic connections involving names or index integers that you can leverage to create elegant, concise code.

One last thought regarding this example involves the possible usage of the `canvas` object, which you learned about in Chapter 31, "Image, Area, Map, and Canvas Objects." The `canvas` object makes it possible to forego images altogether and render the resistor purely through graphics methods. In other words, you could use the `canvas` object to draw the static parts of the resister via lines and circles, and then draw each colored stripe as a rectangle. Granted, it would be tough getting the nifty shaded 3D effect, but you could eliminate the entire issue of relying on images.

CHAPTER

57

Application: Intelligent "Updated" Flags

IN THIS CHAPTER

Temporary and persistent cookies

World time calculations

CGI-like intelligence

It happens to every active web user all the time: You visit a site periodically and never know for sure what material is new since your last visit. Often, web page authors may flag items with "New" or "Updated" iconic images after they update those items themselves. But if you fail to visit the site over a few modification sessions, the only items you find flagged are those that are new as of the most recent update by the page's author. Several new items from a few weeks back may be of vital interest to you, but you won't have the time to look through the whole site in search of material that is more recent than *your* last visit. Even if the items display their modification dates, do you remember for sure the date and time of your last visit to the page?

As much as we might expect a server-side program and database on a web site to keep track of my last visit, that really is asking a great deal of the web site. Besides, not every web site has the wherewithal to build such a database system — if it can even utilize server applications. Plus, some users won't visit sites if they need to identify themselves or register.

After surveying the way scriptable browsers store cookie information and how time calculations are performed under popular web browsers, I found that a feasible alternative is to build this functionality into HTML documents and let the scripting manage the feature for users. The goal is to save the date and time of the last visit to a page in the visitor's cookie file, and then use that point as a measure against items that have an authorship time stamp in the HTML document.

The Cookie Conundrum

Managing the cookie situation in this application is a bit more complicated than you may think. The reason is that you have to take into account the possible ways visitors may come and go from your site while surfing the

Web. You cannot use just one cookie to store the last time a user visits the site, because you cannot predict when you should update that information with today's date and time. For example, if you have a cookie with the previous visit in it, you eventually need to store today's visit. But you cannot afford to overwrite the previous visit immediately (say, in `onload`) because your scripts need that information to compare against items on the page not only right now, but even after the visitor vectors off from a link and comes back later. That also means you cannot update that last visit cookie solely via an `onunload` event handler, because that, too, would overwrite information that you need when the visitor comes back a minute later.

To solve the problem, I devised a system of two cookies. One is written to the cookie that is given an expiration date of some time out in the future — the *hard cookie,* I call it. The other is a temporary cookie — the *soft cookie* — which stays in cookie memory but is never written to the file. Such temporary cookies are automatically erased as the browser quits.

The hard cookie stores the time stamp when a visitor first loads the page since the last launch of the browser. In other words, the hard cookie contains a time stamp of the current visit. Before the previous entry is overwritten, however, it is copied into the soft cookie. That soft cookie maintains the time stamp of the previous visit and becomes the measure against which author time stamps in the HTML document are compared. To guard against inadvertent overwriting of both cookies, a function triggered by the document's `onload` event handler looks to see if the soft cookie has any data in it. If so, the function knows that the visitor has been to this page in the current session and leaves the current settings alone. Thus, the visitor can come to the site and see what's new, vector off to some other location, come back to see the same new items flagged, and pick up from there.

One potential downside to this system is that if a user never quits the browser (or if the browser quits only by crashing), the cookies will never be updated. If you discover that a great deal of your users keep their computers and browsers running all the time, you can build in a kind of timeout that invalidates the soft cookie if the hard cookie is more than, say, 12 hours old.

Time's Not on Your Side

Thanks to more than 17 years' experience programming applications that involve tracking time, I am overly sensitive to the way computers and programming languages treat time on a global basis. This issue is a thorny one, what with the vagaries of Daylight Savings Time and time zones in some parts of the world that differ from their neighbors by increments other than whole hours.

To accomplish a time tracking scheme for this application, I had to be aware of two times: the local time of the visitor and the local time of the page author. Making times match up in what can be widely disparate time zones, I use one time zone — GMT — as the reference point.

When a visitor arrives at the page, the browser needs to save that moment in time so that it can be the comparison measure for the *next* visit. Fortunately, whenever you create a new date object in JavaScript, it does so internally as the GMT date and time. Even though the way you attempt

to read the date and time created by JavaScript shows you the results in your computer's local time, the display is actually filtered through the time zone offset as directed by your computer's time control panel. In other words, the millisecond value of every date object you create is maintained in memory in its GMT form. That's fine for the visitor's cookie value.

For the page author, however, I was presented with a different problem. Rather than force the author to convert the time stamps throughout the document to GMT, I wanted to let the author enter dates and times in local time. Besides the fact that many people have trouble doing time zone conversions, it is much easier on the scripter's side to look at an existing item in the HTML with a local time stamp and recognize when it was last updated.

The problem, then, is how to let the visitor's browser know what time the author's time stamp is in GMT terms. To solve the issue, the author's time stamp needs to include a reference to the author's time zone relative to GMT. An Internet convention provides a couple of ways to do this: specifying the number of hours and minutes difference from GMT or, where supported by the browser, the abbreviation of the time zone. In JavaScript, you can create a new date object out of one of the specially formatted strings containing the date, time, and time zone. Three examples follow for the Christmas Eve dinner that starts at 6 p.m. in the Eastern Standard Time zone of North America:

```
var myDate = new Date("24 Dec 2010 23:00:00 GMT");
var myDate = new Date("24 Dec 2010 18:00:00 GMT-0500");
var myDate = new Date("24 Dec 2010 18:00:00 EST");
```

The first assumes you know the Greenwich Mean Time for the date and time that you want to specify. But if you don't, you can use the GMT designation and offset value. The syntax indicates that the date and time is in a time zone exactly five hours west of GMT (values to the east would be positive numbers) in `hhmm` format. Browsers also know all of the time zone abbreviations for North America (EST, EDT, CST, CDT, MST, MDT, PST, and PDT, where "S" is for standard time and "D" is for daylight time).

When a user visits a page with this application embedded in it, the visitor's browser converts the author's time stamp to GMT (with the help of the author's zone offset parameter), so that both the author time stamp and last visit time stamp (in the soft cookie) are comparing apples to apples.

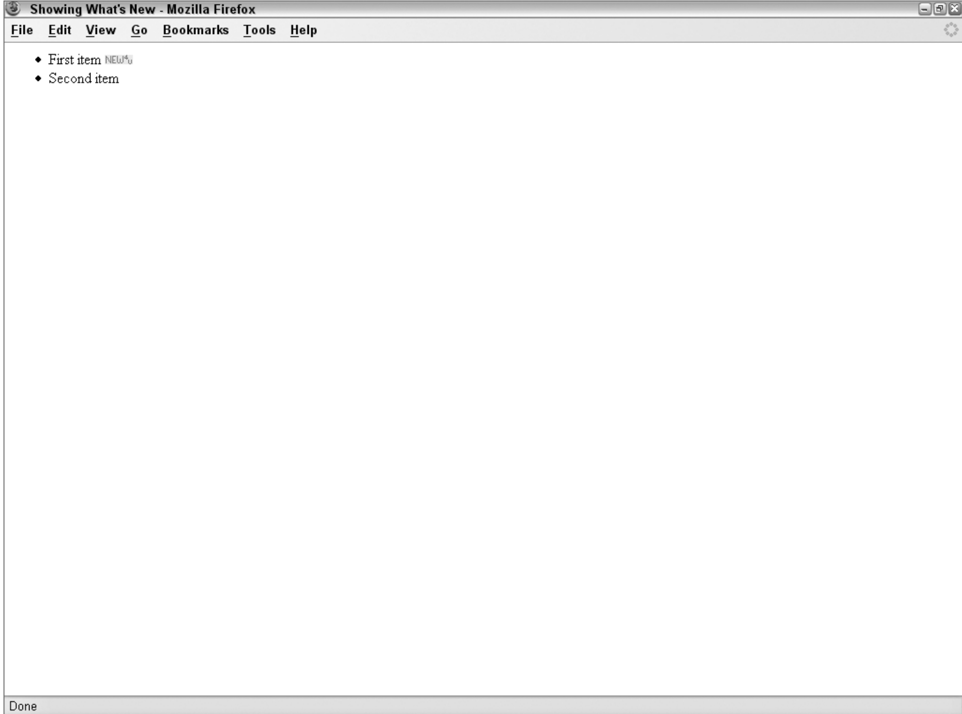
The Application

All of this discussion may make the application sound complicated. That may be true, internally. But the goal, as in most of the samples in this book, is to make the application easy to use on your site, even if you're not sure how everything works inside.

The sample page described in this chapter and on the CD-ROM (`whatsnew.html`) is pretty boring to look at, because the power all lies in the scripting that users don't see (see Figure 57-1). Though this figure may be the most uninspired graphic presentation of the book, the functionality may be the most valuable addition that you make to your web site.

FIGURE 57-1

An item flagged as being new since my last visit to the page.



When you first open the document (do so from a copy on your hard disk so that you can modify the author time stamp in a moment), all you see are the two items on the page without any flags. Although both entries have author time stamps that pre-date the time you're viewing the page, a soft cookie does not yet exist against which to compare those times. But the act of making the first visit to the page has created a hard cookie of the date and time.

Note

Testing the sample page requires that cookies are enabled in your browser. Also, be aware that if you are testing with Google Chrome, your code will not work unless the files are coming from an HTTP server. Chrome intentionally disables cookies on `file:///` documents. See Chapter 29, "The Document and Body Objects," for a discussion of cookies. ■

Quit the browser to get that hard cookie officially written to the cookie file. Then open the `whatsnew.html` file in your script editor. Scroll to the bottom of the document, where you see the `<body>` tag and the interlaced scripts that time stamp anything that you want on the page. This application is designed to display a special `.gif` image that says "NEW 4U" whenever an item has been updated since your last visit.

Each interlaced script looks like this:

```
<script type="text/javascript">
   document.write(newAsOf("12 Oct 2010 13:36:00 PDT"))
</script>
```

The `document.write()` method writes to the page whatever HTML comes back from the `newAsOf()` function. The parameter to the `newAsOf()` function is what holds the author time stamp and zone offset information. The time stamp value must be in the string format, as shown in the preceding example, with the date and time following the exact order "`dd mmm yyyy hh:mm:ss`". Month abbreviations are in English (Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec).

As you see in the code that follows, the `newAsOf()` function returns an `<img>` tag with the "NEW 4U" image if the author time stamp (after appropriate conversion) is later than the soft cookie value. This image can be placed anywhere in a document. For example, at my web site, I sometimes place the image before a contents listing rather than at the end. This means, too, that if part of your page is written by `document.write()` methods, you can just insert the `newAsOf()` function call as a parameter to your own `document.write()` calls.

If you want to see the author time stamping work, edit one of the time stamps in the `whatsnew.html` file to reflect the current time. Save the document and relaunch the browser to view the page. The item whose author time stamp you modified should now show the bright "NEW 4U" image.

The Code

The HTML file for the sample page is simple enough. I have you create a simple bulleted list of two entries, imaginatively called "First item" and "Second item." Interlaced into the HTML are scripts that are ready to insert the "NEW 4U" image if the author time stamp is new enough:

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Showing What's New</title>
      <script type="text/javascript" src="whatsnew.js"></script>
   </head>
   <body>
      <ul>
         <li>First item
            <script type="text/javascript">
               document.write(newAsOf("19 Jan 2010 19:55:00 PST"))
            </script>
         </li>
         <li>Second item
            <script type="text/javascript">
               document.write(newAsOf("15 Jan 2010 13:36:00 PST"))
            </script>
```

```
         </li>
      </ul>
   </body>
</html>
```

The JavaScript file for the sample page starts by initializing three global variables that are used in the statements that follow. One variable is a Boolean value indicating whether the visitor has been to the page before. Another variable, `lastVisit`, holds the value of the soft cookie whenever the visitor is at this page:

```
// globals
var repeatCustomer = false;
var lastVisit = 0; // to hold date & time of previous access in GMT milliseconds
```

For reading and writing cookie data, I'm including an expiration date, because I want this cookie to hang around in the cookie file for a while — at least until the next visit, whenever that may be:

```
// shared cookie functions
var mycookie = document.cookie;
// read cookie data
function getCookieData(name)
{
   var label = name + "=";
   var labelLen = label.length;
   var cLen = mycookie.length;
   var i = 0;
   while (i < cLen)
   {
      var j = i + labelLen;
         if (mycookie.substring(i,j) == label)
         {
            var cEnd = mycookie.indexOf(";",j);
               if (cEnd == -1)
               {
                  cEnd = mycookie.length;
               }
            return unescape(mycookie.substring(j,cEnd));
         }
      i++;
   }
   return "";
}

// write cookie data
function setCookieData(name,dateData,expires)
{
   mycookie = document.cookie = name + "=" + dateData
                              + "; expires=" + expires;
}
```

Notice that the `setCookieData()` function still maintains a level of reusability by requiring a name for the cookie to be passed as a parameter, along with the data and expiration date. I could have hard-wired the name into this function, but that goes against my philosophy of designing for reusability.

Setting the stage

Functions in the next part of the script get your cookies all in a row. The first function (`saveCurrentVisit()`) deals with the visitor's local time, converting it to a form that will be useful on the next visit. Although one of the local variables is called `nowGMT`, all the variable does is take the new date object and convert it to the GMT milliseconds value (minus any `dateAdjustment` value) by invoking the `getTime()` method of the date object. I use this name in the variable to help me remember what the value represents:

```
// write date of current visit (in GMT time) to cookie
function saveCurrentVisit()
{
   var visitDate = new Date();
   var nowGMT = visitDate.getTime();
   var expires = (nowGMT + (180 * 24 * 60 * 60 *1000));
   expires = new Date(expires);
   expires = expires.toGMTString();
   setCookieData("lastVisit", nowGMT, expires);
}
```

From the current time, I create an expiration date for the cookie. The example shows a date roughly six months (180 days, to be exact) from the current time. I leave the precise expiration date up to your conscience and how long you want the value to linger in a user's cookie file.

The final act of the `saveCurrentVisit()` function is to pass the relevant values to the function that actually writes the cookie data. I assign the name `lastVisit` to the cookie. If you want to manage this information for several different pages, assign a different cookie name for each page. This setup can be important in case a user gets to only part of your site during a visit. On the next visit, the code can point to page-specific new items.

The "temporary" cookie also needs to set an expiration date for itself. Normally, temporary cookies disappear when the user quits the browser. But these days, it's not uncommon for users to keep their computers and applications running for days (with fewer crashes than not too long ago). To force the browser to visit the site after awhile, as if coming fresh (to look for new flags), the `nextPrevVisit` cookie is set to expire one hour after it is set:

```
// set cookie with next previous visit with one-hour expiration
function saveNextPrevVisit(lastStoredVisit)
{
   var visitDate = new Date();
   var nowGMT = visitDate.getTime();
   var expires = (nowGMT + (60*1000));
   expires = new Date(expires);
```

```
      expires = expires.toGMTString();
      setCookieData("nextPrevVisit", lastStoredVisit, expires);
   }
```

The bulk of what happens in this application takes place in an initialization function. All the cookie swapping occurs there, as well as the setting of the `repeatCustomer` global variable value:

```
// set up global variables and establish whether user is repeat customer
function initialize()
{
   var lastStoredVisit = getCookieData("lastVisit");
   var nextPrevStoredVisit = getCookieData("nextPrevVisit");

   if (!lastStoredVisit)
   { // never been here before
      saveCurrentVisit();
      repeatCustomer = false;
   }
   else
   { // been here before...
      if (!nextPrevStoredVisit)
      {  // but first time this session
         saveNextPrevVisit(lastStoredVisit); // expires in one hour
         lastVisit = parseFloat(lastStoredVisit);
         saveCurrentVisit();
         repeatCustomer = true;
      }
      else
      { // back again during this session (perhaps reload or Back)
         lastVisit = parseFloat(nextPrevStoredVisit);
         repeatCustomer = true;
      }
   }
}

initialize();
```

The first two statements retrieve both hard (`lastVisit`) and soft (`nextPrevVisit`) cookie values. After calling the function that sets any necessary date adjustment, the script starts examining the values of the cookies to find out where the visitor stands upon coming to the page.

The first test is whether the person has ever been to the page before. You do this by checking whether a hard cookie value (which would have been set in a previous visit) exists. If no such cookie value exists, the current visit time is written to the hard cookie and `repeatCustomer` is set to `false`. These actions prepare the visitor's cookie value for the *next* visit.

Should a user already be a repeat customer, you have to evaluate whether this visit is the user's first visit since launching the browser. You do that by checking for a value in the soft cookie. If that value doesn't exist, it means the user is here for the first time "today." You then grab the hard cookie and drop it into the soft cookie before writing today's visit to the hard cookie.

For repeat customers who have been here earlier in this session, you update the `lastVisit` global variable from the cookie value. The variable value will have been destroyed when the user left the page just a little while ago, whereas the soft cookie remains intact, enabling you to update the variable value now.

Outside of the function definition, the script automatically executes the `initialize()` function by that single statement. This function runs every time the page loads.

The date comparison

Every interlaced script in the body of the document calls the `newAsOf()` function to find out if the author's time stamp is after the user's last visit to the page. This function is where the time zone differences between visitor and author must be neutralized so that a valid comparison can be made:

```
function newAsOf(authorDate)
{
   authorDate = new Date(authorDate);
   var itemUpdated = authorDate.getTime();
   return ((itemUpdated > lastVisit) && repeatCustomer)
           ? "<img src='updated.gif' height=10 width=30 \/>" : "";
}
```

As you saw earlier, calls to this function require one parameter: a specially formatted date string that includes time zone information. The first task in the function is to re-cast the author's date string to a date object. You reuse the variable name (`authorDate`) because its meaning is quite clear. The date object created here is stored internally in the browser in GMT time, relative to the time zone data supplied in the parameter. To assist in the comparison against the `lastVisit` time (stored in milliseconds), the `getTime()` method converts `authorDate` to GMT milliseconds.

The last statement of the function is a conditional expression that returns the `<img>` tag definition for the "NEW 4U" image only if the author's time stamp is later than the soft cookie value, and the visitor has been here before. Otherwise, the function returns an empty string. Any `document.write()` method that calls this function and executes via this branch writes an empty string — nothing — to the page.

Further Thoughts

You can, perhaps, go overboard with the way that you use this technique on a web site. Like most features in JavaScript, I recommend using it in moderation and confining the flags to high-traffic areas that repeat visitors frequent. For example, it makes a great callout for new entries in a blog, since blog updates are often time sensitive. One hazard is that you can exceed the number of cookies a browser allows if you have too many page-specific listings. Although some modern browsers (Safari and Chrome) don't have a limit, others do. The fewest number

of cookies allowed on some browsers (IE6 and legacy) is 20. Other browsers allow a varying number of cookies.

You can share the same cookie among documents in related frames. Copy all the functions from the `head` section of the script in this chapter into a `head` section of a framesetting document. Then, modify the call to the `newAsOf()` function by pointing it to the parent:

```
document.write(parent.newAsOf("18 Nov 2006 17:40:00 PDT"));
```

That way, one cookie can take care of all documents that you display in that frameset.

CHAPTER

58

Application: Decision Helper

IN THIS CHAPTER

Multiple frames

Multiple-document applications

Multiple windows

Persistent storage (cookies)

Scripted image maps

Scripted charts

The list of key concepts for this chapter's application looks like the grand finale to a fireworks show. As JavaScript implementations go, this application is, in some respects, over the top, yet not out of the question for presenting a practical interactive application on a web site without any server programming.

The Application

I wanted to implement a classic application often called a *decision support system*. My experience with the math involved here goes back to the first days of Microsoft Excel. Rather than design a program that had limited appeal (covering only one possible decision tree), I set out to make a completely user-customizable decision helper. All the user has to do is enter values into fields on a series of screens; the program performs the calculations to let the user know how the various choices rank against each other.

Although I won't be delving too deeply into the math inside this application, you will find it helpful to understand how a user approaches this program and what the results look like. The basic scenario is a user who is trying to evaluate how well a selection of choices measure up to his or her expectations of performance. User input includes:

- The name of the decision
- The names of up to five alternatives (people, products, ideas, and so on)
- The factors or features of concern to the user
- The importance of each of the factors to the user
- A user ranking of the performance of every alternative in each factor

What makes this kind of application useful is that it forces the user to rate and weigh a number of often-conflicting factors. By assigning hard numbers to these elements, the computer calculates a decision based on the rates and weights of the various factors.

Results come in the form of floating-point numbers between 0 and 100. As an extra touch, I've added a graphical charting component to the results display.

The Design

With so much user input necessary for this application, conveying the illusion of simplicity is important. Rather than lump all text objects onto a single scrolling page, I decided to break them into five pages, each consisting of its own HTML document. As an added benefit, I could embed information from early screens into the HTML of later screens, which would help to establish a more intuitive user interface. This "good idea" presented one opportunity and one rather large challenge.

The opportunity was to turn the interface for this application into something resembling a multimedia application using multiple frames. The largest frame would contain the forms the user fills out, as well as the results page. Another frame would contain a navigation panel with arrows for moving forward and backward through the sequence of screens, plus buttons for going back to a home page and getting information about the program. I also thought a good idea would be to add a frame that provides instructions or suggestions for the users at each step. And so, the three-frame window was born, as shown in the first entry screen in Figure 58-1.

Using a navigation bar also enables me to demonstrate how to script a client-side image map — not an obvious task with JavaScript. Also notice in the figure how there is a helpful text message appearing just above the navigation bar. This "tooltip"-style message appears thanks to the use of several handy `onmouseover` and `onmouseout` event handlers, which you learn about a bit later in the chapter.

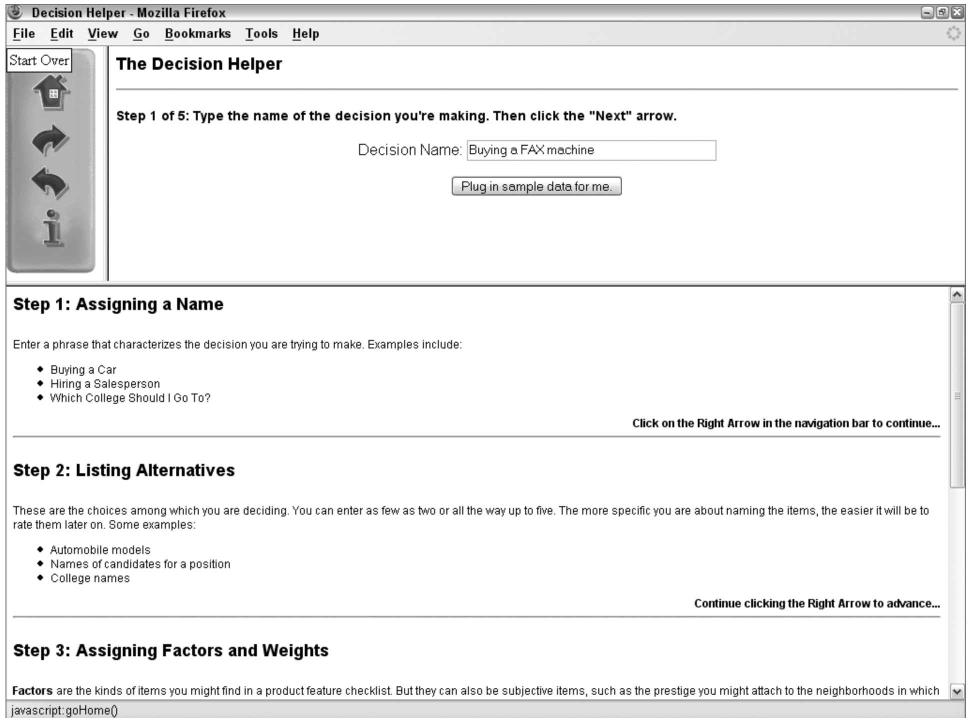
On the challenge side of this design, finding a way to maintain data globally, as the user navigates from screen to screen, was necessary. Every time one of the entry pages unloads, none of its text fields is available to a script. My first attack at this problem was to store the data as global variable data (mostly arrays) in the parent document that creates the frames. Because JavaScript enables you to reference any parent document's object, function, or variable (by preceding the reference with `parent`), I thought this task would be a snap. But a hazard exists in that a reload of the frameset could erase the current state of those variables.

My next hope was to use the `document.cookie` as the storage bin for the data. A major problem I faced was that this program needs to store a total of 41 individual data points, yet no more than 20 uniquely-named cookies can be allotted to a given domain. (While some modern browsers don't have a limit, other modern browsers and legacy browsers allow between 20 and 50 cookies.) But the cookie proved to be the primary solution for this application. (See the "Further Thoughts" section at the end of the chapter about a non-cookie version on the

CD-ROM.) For some of the data points (which are related in an array-like manner), I fashioned my own data structures so that one named cookie could contain up to five related data points. That reduced my cookie demands to 17.

FIGURE 58-1

The Decision Helper window consists of three frames.



Note

Testing the sample application obviously requires that cookies be enabled in your browser. Be aware that if you are testing with Google Chrome, your code will not work unless the files are coming from an HTTP server. Chrome intentionally disables cookies on `file:///` documents. See Chapter 29, "The Document and Body Objects," for a discussion of cookies. ■

The Files

Before I get into the code, let me explain the file structure of this application. Table 58-1 gives a rundown of the files used in the Decision Helper.

TABLE 58-1

Files Comprising the Decision Helper Application

File	Description
index.html	Framesetting parent document
dhNav.html	Navigation bar document for the upper-left frame that contains some scripting
dhNav.gif	Image displayed in dhNav.html
dh1.html	First Decision Helper entry page
dh2.html	Second Decision Helper entry page
dh3.html	Third Decision Helper entry page
dh4.html	Fourth Decision Helper entry page
dh5.html	Results page
chart.gif	Tiny image file used to create bar charts in dh5.html
dhHelp.html	Sample data and instructions document for the lower frame
dhAbout.html	Document that loads into a second window

A great deal of interdependence exists among these files. As you see later, assigning the names to some of these files was strategic for the implementation of the image map.

The Code

With so many JavaScript-enhanced HTML documents in this application, you can expect a great deal of code. To best grasp what's going on here, first try to understand the structure and interplay of the documents, especially the way the entry pages rely on functions defined in the parent document. My goal in describing this structure is not to teach you how to implement this application, but rather how to apply the lessons I learned while building this application to the more complex ideas that may be aching to get out of your head and into JavaScript.

index.html

Taking a top-down journey through the JavaScript and HTML of the Decision Helper, start at the document that loads the frames. Unlike a typical framesetting document, however, this one contains JavaScript code in its head section — code that many other documents rely on:

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Decision Helper</title>
```

An important consideration to remember is that in a multiple-frame environment, the title of the parent window's document is the name that appears in the window's title bar, no matter how many other documents are open inside its subframes.

The first items of the script control a global variable, `currTitle`, which is set by a number of the subsidiary files as the user navigates through the application. This variable ultimately helps the navigation bar buttons do their jobs correctly. Because this application relies on the `document.cookie` so heavily, the cookie management functions (slightly modified versions of Bill Dortch's cookie functions — see Chapter 29) are located in the parent document so they load only once. I simplified the cookie-writing function because this application uses default settings for pathname and expiration. With no expiration date, the cookies don't survive the current browser session, which is perfect for this application:

```
<script type="text/javascript">
   // global variable settings of current dh document number
   var currTitle = "";
   function setTitleVar(titleVal)
   {
      currTitle = "" + titleVal;
   }
   function getCookieVal(offset)
   {
      var endstr = document.cookie.indexOf (";", offset);
         if (("" + endstr) == "" || endstr == -1)
         {
            endstr = document.cookie.length;
         }
      return unescape(document.cookie.substring(offset, endstr));
   }

   function getCookie(name)
   {
      var arg = name + "=";
      var alen = arg.length;
      var clen = document.cookie.length;
      var i = 0;
      while (i < clen)
      {
         var j = i + alen;
            if (document.cookie.substring(i, j) == arg)
            {
               return getCookieVal (j);
            }
         i = document.cookie.indexOf(" ", i) + 1;
            if (i == 0)
            {
               break;
            }
      }
      return null;
   }
```

```
function setCookie(name, value)
{
   document.cookie = name + "=" + escape (value) + ";";
}
```

When this application loads (or a user elects to start a new decision), it's important to grab the cookies you need and initialize them to the basic values that the entry screens will use to fill fields when the user first visits them. All statements inside the `initializeCookies()` function call the `setCookie()` function, defined in the preceding listing. The parameters are the name of each cookie and the initial value — mostly empty strings. Before going on, study the cookie labeling structure carefully. I refer to it often in discussions of other documents in this application:

```
function initializeCookies()
{
   setCookie("decName","");
   setCookie("alt0","");
   setCookie("alt1","");
   setCookie("alt2","");
   setCookie("alt3","");
   setCookie("alt4","");
   setCookie("factor0","");
   setCookie("factor1","");
   setCookie("factor2","");
   setCookie("factor3","");
   setCookie("factor4","");
   setCookie("import","0");
   setCookie("perf0","");
   setCookie("perf1","");
   setCookie("perf2","");
   setCookie("perf3","");
   setCookie("perf4","");
}
```

The following functions should look familiar to you. They were borrowed either wholesale or with minor modification from the data-entry validation section of the Social Security number database lookup in Chapter 53, "Application: A Lookup Table." I'm glad I wrote these as generic functions, making them easy to incorporate into this script. Because many of the entry fields on two screens must be integers between 1 and 100, I brought the data validation functions to the parent document rather than duplicating them in each of the subdocuments:

```
// JavaScript sees numbers with leading zeros
// as octal values, so strip zeros
function stripZeros(inputStr)
{
   var result = inputStr;
   while (result.substring(0,1) == "0")
   {
      result = result.substring(1,result.length);
   }
   return result;
}
```

```
// general purpose function to see if a suspected numeric
// input is a positive integer
function isNumber(inputStr)
{
   for (var i = 0; i < inputStr.length; i++)
   {
      var oneChar = inputStr.substring(i, i + 1);
         if (oneChar < "0" || oneChar > "9")
         {
            return false;
         }
   }
   return true;
}

// function to determine if value is in acceptable
// range for this application
function inRange(inputStr)
{
   num = parseInt(inputStr);
   if (num < 1 || num > 100)
   {
      return false;
   }
   return true;
}
```

To control the individual data entry validation functions in the master controller, I again was able to borrow heavily from the application in Chapter 53:

```
// Master value validator routine
function isValid(inputStr)
{
   if (inputStr != "" )
   {
      inputStr = stripZeros(inputStr);
      if (!isNumber(inputStr))
      {
         alert("Please make sure entries are numbers only.");
         return false;
      }
      else
      {
         if (!inRange(inputStr))
         {
            var msg="Entries must be numbers between 1 and 100.";
            msg += " Try another value.";
            alert(msg);
            return false;
         }
      }
   }
   return true;
}
```

Each of the documents containing entry forms retrieves and stores information in the cookie. Because all cookie functions are located in the parent document, it simplifies coding in the subordinate documents to have functions in the parent document acting as interfaces to the primary cookie functions. For each category of data stored as cookies, the parent document has a pair of functions for getting and setting data. The calling statements pass only the data to be stored when saving information; the interface functions handle the rest, such as storing or retrieving the cookie with the correct name.

In the following pair of functions, the decision name (from the first entry document) is passed back and forth between the cookie and the calling statements. Not only must the script store the data, but if the user returns to that screen later for any reason, the entry field must retrieve the previously entered data:

```
function setDecisionName(str)
{
   setCookie("decName",str);
}
function getDecisionName()
{
   return getCookie("decName");
}
```

The balance of the storage and retrieval pairs does the same thing for their specific cookies. Some cookies are named according to index values (`factor1`, `factor2`, and so on), so their cookie-accessing functions require parameters for determining which of the cookies to access, based on the request from the calling statement. Many of the cookie retrieval functions are called to fill in the data in tables of later screens during the user's trip down the decision path:

```
function setAlternative(i,str)
{
   setCookie("alt" + i,str);
}
function getAlternative(i)
{
   return getCookie("alt" + i);
}
function setFactor(i,str)
{
   setCookie("factor" + i,str);
}
function getFactor(i)
{
   return getCookie("factor" + i);
}
function setImportance(str)
{
   setCookie("import",str);
}
function getImportance(i)
{
```

```
      return getCookie("import");
   }
   function setPerformance(i,str)
   {
      setCookie("perf" + i,str);
   }
   function getPerformance(i)
   {
      return getCookie("perf" + i);
   }
```

One sequence of code that runs when the parent document loads is the one that looks to see if a cookie structure is set up. If no such structure is set up (the retrieval of a designated cookie returns a `null` value), the script initializes all cookies via the function described earlier:

```
   if (getDecisionName() == null)
   {
      initializeCookies();
   }
```

The last part of the script is a function that's called if the user wants to plug in a canned example by clicking a button in the file (`dh1.html`) that is the first data entry screen in the upper-right frame:

```
   // plug in sample code if user wants
   function doSample(form)
   {
      setCookie("decName","Buying a FAX machine");
      setCookie("alt0","Fax-O-Matic 1000");
      setCookie("alt1","InkyFax 300");
      setCookie("alt2","LazyFax LX");
      setCookie("alt3","Loose Cannon M-200");
      setCookie("alt4","");
      setCookie("factor0","Cost");
      setCookie("factor1","Size");
      setCookie("factor2","Paper Handling");
      setCookie("factor3","Warranty");
      setCookie("factor4","");
      setCookie("import","80.40.60.70..");
      setCookie("perf0","60.80.40.20..");
      setCookie("perf1","80.60.50.30..");
      setCookie("perf2","80.55.75.70..");
      setCookie("perf3","70.70.80.70..");
      setCookie("perf4","");
      parent.entryForms.document.forms[0].decName.value
                               = "Buying a FAX machine";
   }
 </script>
</head>
```

The balance of the parent document source code defines the frameset for the browser window. It establishes some hard-wired pixel sizes for the navigation panel. This ensures that the entire

`.gif` file is visible whenever the frameset loads, without a ton of unwanted white space if the browser window is large:

```
      <frameset rows="250,*">
         <frameset cols="104,*">
            <frame name="navBar" src="dhNav.html" scrolling="no" marginheight="2"
                   marginwidth="1" />
            <frame name="entryForms" src="dh1.html" />
         </frameset>
         <frameset rows="100%">
            <frame name="instructions" src="dhHelp.html" />
         </frameset>
         <noframes>
            <body>
               <h1>It's really cool...</h1>
               <h2>...but only if your browser handles frames</h2>
               <hr />
               <a href="/index.html">Back</a>
            </body>
         </noframes>
      </frameset>
   </html>
```

It's worth pointing out that some older browsers do not respond to changes in framesetting size attributes through a simple reload of the page. Modern browsers appear to have resolved this problem; however if you seem to be making changes, but reloading the frameset doesn't make the changes appear, try reopening the browser window or — as a last resort — restarting the browser.

dhNav.html

Because of its crucial role in controlling the activity around this program, look into the navigation bar's document next. To accomplish the look and feel of a multimedia program, this document was designed as a client-side image map that has four regions scripted, corresponding to the locations of the four buttons (see Figure 58-1). The trick to the tooltip mechanism is the showing and hiding of a `div` element that contains the text to be displayed. This element has its own style class, `tipStyle`, which is responsible for establishing the white rectangular text box with a thin black border.

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Navigation Bar</title>
      <style type="text/css">
         body
         {
            background-color:white;
         }
         .tipStyle
         {
            position:absolute;
```

```
        background-color:#FFFFFF;
        border:solid 1px #000000;
        padding:2px;
        visibility: hidden;
    }
</style>
```

One function is connected to each button. The first function is linked to the graphical Home button. For the listing here, I just present an alert dialog box replicating the action of navigating back to a real web site's home page:

```
<script type="text/javascript">
   function goHome()
   {
      alert("Navigate back to home page on a real site.");
   }
```

Each of the arrow navigation buttons brings the user to the next or previous entry screen in the sequence. To facilitate this without building tables of document titles and names, you call upon the `currTitle` global variable in the parent document. This value contains an integer in the range between 1 and 5, corresponding to the main content documents `dh1.html`, `dh2.html`, and so on. As long as the offset number is no higher than the next-to-last document in the sequence, the `goNext()` function increments the index value by one and concatenates a new location for the frame. At the same time, the script advances the help document (in the bottom frame) to the anchor corresponding to the chosen entry screen by setting the `location.hash` property of that frame. Similar action navigates to the previous screen of the sequence through the `goPrev()` function. This time, the index value is decremented by one, and an alert warns the user if the current page is already the first in the sequence:

```
   function goNext()
   {
      var currOffset = parseInt(parent.currTitle);
         if (currOffset <= 4)
         {
            ++currOffset;
            parent.entryForms.location.href = "dh"
                                            + currOffset
                                            + ".html";
            parent.instructions.location.hash = "help"
                                              + currOffset;
         }
         else
         {
            alert("This is the last form.");
         }
   }
   function goPrev()
   {
      var currOffset = parseInt(parent.currTitle);
         if (currOffset > 1)
```

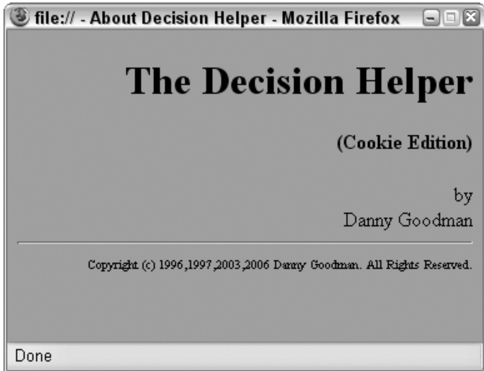
```
            {
               --currOffset;
               parent.entryForms.location.href = "dh"
                                                + currOffset
                                                + ".html";
               parent.instructions.location.hash = "help"
                                                + currOffset;
            }
            else
            {
               alert("This is the first form.");
            }
      }
```

Clicking the Info button displays a smaller window containing typical About-box data for the program (see Figure 58-2):

```
function goInfo()
{
   var newWindow =
       window.open("dhAbout.html","","height=250,width=380");
}
```

FIGURE 58-2

The About Decision Helper screen appears in a separate window.



The last pair of functions, `showTip()` and `hideTip()`, are used to show and hide, respectively, the tooltip text that appears when the mouse hovers over one of the navigation buttons. The `showTip()` function accepts the text to be shown in the tooltip as its only argument:

```
function showTip(tipMsg)
{
   var obj = document.getElementById("tip");
   obj.style.visibility = "visible";
```

```
      while(obj.firstChild)
         obj.removeChild(obj.firstChild);
      obj.appendChild(document.createTextNode(tipMsg));
   }
   function hideTip()
   {
      var obj = document.getElementById("tip");
      obj.style.visibility = "hidden";
   }
   </script>
```

The body of the navigation document contains the part that enables you to script a client-side image map. Mouse click events weren't available to area elements in legacy browsers and are not available in all modern browsers, so to inject a bit of backward compatibility, mouse action is converted to script action by assigning a javascript: pseudo-URL to the HREF attribute for each area element. Instead of pointing to an entirely new URL (as area elements usually work), the attributes point to the JavaScript functions defined in the head portion of this document. After a user clicks the rectangle specified by an <area> tag, the browser invokes the function instead.

```
<body>
      <div id="tip" class="tipStyle"></div>
      <map id="navigation" name="navigation">
         <area shape="rect" coords="23,22,70,67"
               href="javascript:goHome()"
               onmouseover="showTip('Start Over')"
               onmouseout="hideTip()"/>
         <area shape="rect" coords="25,80,66,116"
               href="javascript:goNext()"
               onmouseover="showTip('Next Step')"
               onmouseout="hideTip()"/>
         <area shape="rect" coords="24,125,67,161"
               href="javascript:goPrev()"
               onmouseover="showTip('Previous Step')"
               onmouseout="hideTip()"/>
         <area shape="rect" coords="35,171,61,211"
               href="javascript:goInfo()"
               onmouseover="showTip('Get Info')"
               onmouseout="hideTip()"/>
      </map>
      <img alt="image" src="dhNav.gif" height="240" width="96"
          border="0" usemap="#navigation" />
   </body>
</html>
```

Revealed here is the other component of the tooltip feature you learned about earlier (revisit the upper-left corner of Figure 58-1). Each area element within the image map has two mouse event handlers that take care of showing and hiding the tooltip text, based upon the mouse hovering above the navigation area.

Note

The property assignment event handling technique used throughout this example is a deliberate simplification to make the code more readable. It is generally better to use the more modern approach of binding events using the `addEventListener()` **(NN6+/Moz/W3C) or** `attachEvent()` **(IE5+) methods. A modern cross-browser event handling technique is explained in detail in Chapter 32.** ■

dh1.html

Of the five documents that display in the main frame, `dh1.html` is the simplest (refer to Figure 58-1). It contains a single entry field in which the user is invited to enter the name for the decision.

The function `loadDecisionName` summons one of the cookie interface functions in the parent window. A test is located here in case there is a problem with initializing the cookies. Rather than show `null` in the field, the conditional expression substitutes an empty string:

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>DH1</title>
      <style type="text/css">
         body
         {
            font-family:Arial, Helvetica, serif;
         }
         h2
         {
            font-size:18px;
         }
         h4
         {
            font-size:14px;
         }
         .form
         {
            text-align:center;
         }
      </style>
      <script type="text/javascript" src="../../jsb-global.js"></script>
      <script type="text/javascript">
         // initialize when the page has loaded
         addEvent(window, "load", initialize);

         function initialize()
         {
            parent.setTitleVar(1);
            loadDecisionName();
            document.forms[0].decName.focus();
         }
         function loadDecisionName()
```

```
      {
         var result = parent.getDecisionName();
         result = (result == null) ? "" : result;
         document.forms[0].decName.value = result;
      }
   </script>
</head>
```

After the document loads, it performs three tasks (in the `onload` event handler bound to the `window` object in the first part of the script). The first task is to set the global variable in the parent to let it know which number of the five main documents is currently loaded. Next, the script must fill the field with the decision name stored in the cookie. This task is important because users will want to come back to this screen to review what they entered previously. A third statement sets the focus of the entire browser window to the one text object. This task is especially important in a multi-frame environment, such as this design. After a user clicks on the navigation panel, that frame has the focus. To begin typing into the field, the user has to tab (repeatedly) or click the text box to give it focus for typing. By setting the focus in the script when the document loads, you save the user time and aggravation:

```
<body>
   <h2>The Decision Helper</h2>
   <hr />
   <h4>Step 1 of 5: Type the name of the decision you're making. Then
      click the "Next" arrow.
   </h4>
```

In the text field itself, an `onchange` event handler saves the value of the field in the parent's cookie for the decision name. No special Save button or other instruction is necessary here because any navigation that the user does via the navigation bar automatically causes the text field to lose focus and triggers the `onchange` event handler:

```
<div class="form">
   <form>
      Decision Name:
      <input type="text" name="decName" size="40"
             onchange="parent.setDecisionName(this.value)" />
```

The balance of this file has a (commented out) `textarea` object that you can use for debugging cookie data. This is followed by a button that, when clicked, calls a function in the parent (`index.html`). This allows the user to choose to plug in sample data:

```
<p>
   <script type="text/javascript">
      // un-comment next lines to show textarea that
      // monitors cookie values
      // document.write("<textarea rows='6' cols='50'"
      //                + " name='showcookie' wrap='HARD'>"
      //                + parent.document.cookie
      //                + "<\/textarea>");
   </script>
</p>
```

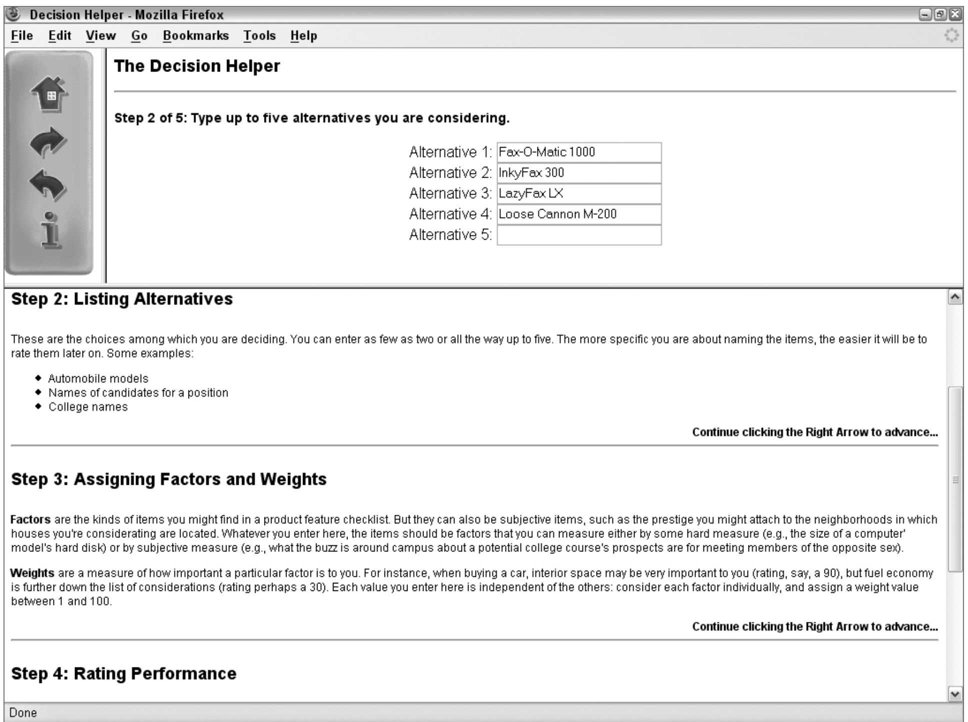
```
            <p><input type="button" value="Plug in sample data for me."
                      onclick="parent.doSample()" />
            </p>
         </form>
      </div>
   </body>
</html>
```

dh2.html

For the second data entry screen (shown in Figure 58-3), five fields invite the user to enter descriptions of the alternatives under consideration. As with the decision name screen, the scripting for this page must both retrieve and save data displayed or entered in the fields.

FIGURE 58-3

The second data entry screen.



In one function, the script retrieves the alternative cookies (five total) and stuffs them into their respective text fields (as long as their values are not `null`). This function uses a `for` loop to cycle through all five items — a common process throughout this application. Whenever a cookie is one of a set of five, the parent function has been written (in the following example) to

store or extract a single cookie, based on the index value. Text objects holding like data (defined in the following listing) are all assigned the same name, so that JavaScript lets you treat them as array objects — greatly simplifying the placement of values into those fields inside a `for` loop:

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>DH2</title>
      <style type="text/css">
         body
         {
            font-family:Arial, Helvetica, serif;
         }
         h2
         {
            font-size:18px;
         }
         h4
         {
            font-size:14px;
         }
         .form
         {
            text-align:center;
         }
      </style>
      <script type="text/javascript" src="../../jsb-global.js"></script>
      <script type="text/javascript">
         // initialize when the page has loaded
         addEvent(window, "load", initialize);

         function initialize()
         {
            parent.setTitleVar(2);
            loadAlternatives();
            document.forms[0].alternative[0].focus();
         }
         function loadAlternatives()
         {
            for (var i = 0; i < 5; i++)
            {
               var result = parent.getAlternative(i);
               result = (result == null) ? "" : result;
               document.forms[0].alternative[i].value = result;
            }
         }
      </script>
   </head>
```

After the document loads, the `onload` event (bound to the `window` object at the beginning of the script) does three things: the document number is sent to the parent's global variable, its

fields are filled by the function defined in the `head`, and the first field is handed the focus to assist the user in entering data the first time.

Any change that a user makes to a field is stored in the corresponding cookie. Each `onchange` event handler passes its indexed value (relative to all like-named fields), plus the value entered by the user, as parameters to the parent's cookie-saving function:

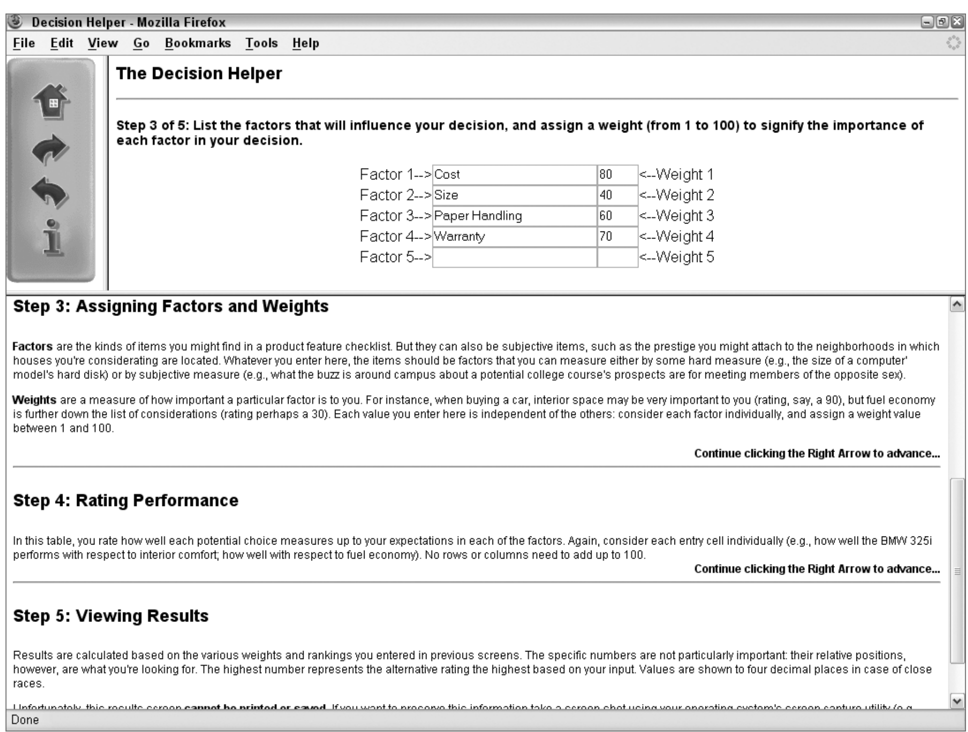
```
   <body>
      <h2>The Decision Helper</h2>
      <hr />
      <h4>Step 2 of 5: Type up to five alternatives you are considering.</h4>
      <div class="form">
         <form>
            Alternative 1:
            <input type="text" name="alternative" size="25"
                   onchange="parent.setAlternative(0,this.value)" />
            <br />
            Alternative 2:
            <input type="text" name="alternative" size="25"
                   onchange="parent.setAlternative(1,this.value)" />
            <br />
            Alternative 3:
            <input type="text" name="alternative" size="25"
                   onchange="parent.setAlternative(2,this.value)" />
            <br />
            Alternative 4:
            <input type="text" name="alternative" size="25"
                   onchange="parent.setAlternative(3,this.value)" />
            <br />
            Alternative 5:
            <input type="text" name="alternative" size="25"
                   onchange="parent.setAlternative(4,this.value)" />
            <br />
         </form>
      </div>
   </body>
</html>
```

dh3.html

With the third screen, the complexity increases a bit. Two factors contribute to this increase in difficulty. One is that the limitation on the number of cookies available for a single domain forces you to combine into one cookie the data that might normally be distributed among five cookies. Second, with the number of text objects on the page (see Figure 58-4), it becomes more efficient (from the standpoint of tedious HTML writing) to let JavaScript deploy the fields. The fact that two sets of five related fields exist facilitates using `for` loops to lay out and populate them.

FIGURE 58-4

Screen for entering decision factors and their weights.



The `onload` event is bound to the `window` object. The only tasks that the `onload` event handler needs to do are to update the parent global variable about the document number, and to set the focus on the first entry field of the form. The `getdh3Factor()` function is reminiscent of `head` functions in previous entry screens. This function retrieves a single factor cookie from the set of five cookies:

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>DH3</title>
      <style type="text/css">
         body
         {
            font-family:Arial, Helvetica, serif;
         }
```

```
      h2
      {
         font-size:18px;
      }
      h4
      {
         font-size:14px;
      }
   </style>
   <script type="text/javascript" src="../../jsb-global.js"></script>
   <script type="text/javascript">
      // initialize when the page has loaded
      addEvent(window, "load", initialize);

      function initialize()
      {
         parent.setTitleVar(3);
         document.forms[0].factor[0].focus();
      }

      function getdh3Factor(i)
      {
         var result = parent.getFactor(i);
            if (result == null)
            {
            return "";
            }
         return result;
      }
```

Values for the five possible weight entries are stored together in a single cookie. To make this work, I had to determine a data structure for the five "fields" of a single cookie "record." Because all entries are integers, any nonnumeric character would work. I arbitrarily selected the period:

```
function setdh3Importance()
{
   var oneRecord = "";
   for (var i = 0; i < 5; i++)
   {
      var dataPoint = document.forms[0].importance[i].value;
         if (!parent.isValid(dataPoint))
         {
            document.forms[0].importance[i].focus();
            document.forms[0].importance[i].select();
            return;
         }
      oneRecord += dataPoint + ".";
   }
   parent.setImportance(oneRecord);
   return;
}
```

The purpose of the `setdh3Importance()` function is to assemble all five values from the five Weight entry fields (named `importance`) into a period-delimited record that is ultimately sent to the cookie for safekeeping. Another of the many `for` loops in this application cycles through each of the fields, checking for validity and then appending the value with its trailing period to the variable (`oneRecord`) that holds the accumulated data. As soon as the loop finishes, the entire record is sent to the parent function for storage.

Although the function shows two `return` statements, the calling statement does not truly expect any values to be returned. Instead, I use the `return` statement inside the `for` loop as a way to break out of the `for` loop without any further execution whenever an invalid entry is found. Just prior to that, the script sets the `focus` and `select` to the field containing the invalid entry. JavaScript, however, is sensitive to the fact that a function with a `return` statement in one possible outcome doesn't have a `return` statement for other outcomes (an error message to this effect appears in some browsers if you try the function without balanced returns). By putting a `return` statement at the end of the function, all other possibilities are covered to the browser's satisfaction.

The inverse of storing the weight entries is retrieving them. Because the `parent.getImportance()` function returns the entire period-delimited record, this function must break apart the pieces and distribute them into their corresponding Weight fields. A combination of string methods determines the offset of the period and how far the data extraction should go into the complete record. Before the `for` loop repeats each time, it is shortened by one "field's" data. In other words, as the `for` loop executes, the copy of the cookie data returned to this function is pared down one entry at a time, as each entry is stuffed into its text object for display:

```
         function getdh3Importance()
         {
            var oneRecord = parent.getImportance();
               if (oneRecord != null)
               {
                  for (var i = 0; i < 5; i++)
                  {
                     var recLen = oneRecord.length;
                     var offset = oneRecord.indexOf(".");
                     var dataPoint = (offset >= 0 )
                                   ? oneRecord.substring(0,offset) : "";
                     document.forms[0].importance[i].value = dataPoint;
                     oneRecord = oneRecord.substring(offset+1,recLen);
                  }
               }
         }
      </script>
   </head>
```

Assembling the HTML for the form and its 10 data entry fields needs only a few lines of JavaScript code. Performed inside a `for` loop, the script assembles each line of the form, which consists of a label for the Factor (and its number), the factor input field, the importance input

field, and the label for the Weight (and its number). A `document.write()` method writes each line to the document:

```
      <body>
         <h2>The Decision Helper</h2>
         <hr />
         <h4>Step 3 of 5: List the factors that will influence your decision, and
            assign a weight (from 1 to 100) to signify the importance of each
            factor in your decision.
         </h4>
         <p>
            <script type="text/javascript">
               var output = "<center><form>";
               for (i = 0; i < 5; i++)
               {
                  output += "Factor "
                         + (i+1)
                         + "--><input type='text' name='factor' size='25' ";
                  var eHandler = " onchange=\'parent.setFactor("
                               + i
                               + ",this.value)\'";
                  output += eHandler
                         + "value='"
                         + getdh3Factor(i)
                         + "'>";
                  output += "<input type='text' name='importance' size='3' ";
                  var eHandler = " onchange=\'setdh3Importance()\'";
                  output += eHandler + "value=''>";

                  output += "<--Weight " + (i+1) + "<br \/>";
                  document.write(output);
                  output = "";
               }
               document.write("<\/form><\/center>");
               getdh3Importance();
            </script>
         </p>
      </body>
   </html>
```

Each of the scripted text objects has an event handler. Notice that each event handler is first defined as a variable on a statement line just above its insertion into the string being assembled for the `input` object definition. One reason for this is that the nested quote situation gets quite complex when you are doing these tasks all in one massive assignment statement. Rather than mess with matching several pairs of deeply nested quotes, I found it easier to break out one portion (the event handler definition) as a variable value and then insert that preformatted expression into the concatenated string for the `input` definition.

Notice, too, how the different ways of storing the data in the cookies influence the ways the existing cookie data is filled into the fields as the page draws itself. For the factors, which have one cookie each, the `value` attribute of the field is set with a specific indexed call to the parent factor cookie retriever, one factor at a time. But, for the importance values, which are stored

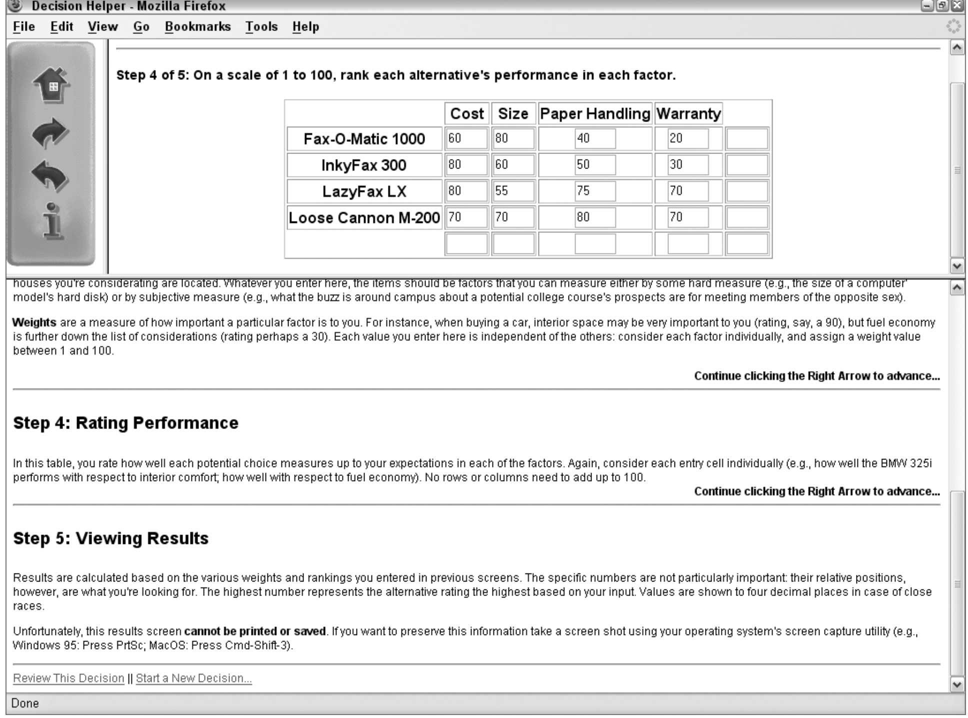
together in the period-delimited chunk, a separate function call (`getdh3Importance()`) executes after the fields are already drawn (with initial values of empty strings) and fills all the fields in a batch operation.

dh4.html

Step 4 of the decision process (shown in Figure 58-5) is the most complex step because of the sheer number of entry fields: 25 in all. Notice that this screen retrieves data from two of the previous screens (or rather, from the cookies preserving the entries) and embeds the values into the fixed parts of the table. All these tasks are possible when you create those tables with JavaScript.

FIGURE 58-5

A massive table includes label data from earlier screen entries.



Functions for getting and setting performance data are complex because of the way I was forced to combine data into five "field" records. In other words, one parent cookie exists for each row of data cells in the table. To extract cell data for storage in the cookie, I use nested `for` loop constructions. The outer loop counts the rows of the table, whereas the inner loop (with the `j` counter variable) works its way across the columns for each row.

Because all cells are named identically, they are indexed with values from 0 to 24. Calculating the row (`i * 5`), plus the column number, establishes the cell index value. After you check for

validity, each cell's value is added to the row's accumulated data. Each row is then saved to its corresponding cookie. As in the code for `dh3.html`, the `return` statement is used as a way to break out of the function if an entry is deemed invalid.

Retrieving the data and populating the cells for the entire table requires an examination of each of the five performance cookies — and, for each labeled cookie's data, a parsing for each period-delimited entry. After a given data point is in hand (one entry for a cell), it must go into the cell with the proper index:

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>DH4</title>
      <style type="text/css">
         body
         {
            font-family:Arial, Helvetica, serif;
         }
         h2
         {
            font-size:18px;
         }
         h4
         {
            font-size:14px;
         }
      </style>
      <script type="text/javascript" src="../../jsb-global.js"></script>
      <script type="text/javascript">
         // initialize when the page has loaded
         addEvent(window, "load", initialize);

         function initialize()
         {
            parent.setTitleVar(4);
            document.forms[0].ranking[0].focus();
         }

         function getdh4Performance()
         {
            var oneRecord = "";
            var recLen = 0;
            var offset = 0;
            var dataPoint = "";
            var cellNum = 0;
            for (var i = 0; i < 5; i++)
            {
               oneRecord = parent.getPerformance(i);
                  if (oneRecord == null)
                  {
                     continue;
                  }
```

```
            for (var j = 0; j < 5; j++)
            {
               recLen = oneRecord.length;
               offset = oneRecord.indexOf(".");
               dataPoint = oneRecord.substring(0,offset);
               cellNum = j + (i * 5);
               document.forms[0].ranking[cellNum].value = dataPoint;
               oneRecord = oneRecord.substring(offset+1,recLen);
            }
         }
      }
      function setdh4Performance()
      {
         var oneRecord = "";
         var cellNum = 0;
         var dataPoint = "";
         for (var i = 0; i < 5; i++)
         {
            oneRecord = "";
            for (var j = 0; j < 5; j++)
            {
               cellNum = j + (i * 5);
               dataPoint = document.forms[0].ranking[cellNum].value;
               if (!parent.isValid(dataPoint))
               {
                  document.forms[0].ranking[cellNum].focus();
                  document.forms[0].ranking[cellNum].select();
                  return;
               }
               oneRecord += dataPoint + ".";
            }
            parent.setPerformance(i,oneRecord);
         }
         return;
      }
   </script>
</head>
```

After the document is loaded, the `onload` event handler (bound to the `window` object at the beginning of the script) sends the document number to the parent global variable and brings focus to the first field of the table.

To lessen the repetitive HTML for all tables, JavaScript again assembles and writes the HTML that defines the tables. In the first batch, the script uses yet another `for` loop to retrieve the factor entries from the parent cookie, so that the words can be embedded into `<th>` tags of the first row of the table. If every factor field is not filled in, the table cell is set to empty:

```
<body>
   <h2>The Decision Helper</h2>
   <hr />
   <h4>Step 4 of 5: On a scale of 1 to 100, rank each alternative's
      performance in each factor.
   </h4>
```

```
<p>
   <script type="text/javascript">
      var output = "<div align='center'>";
      output += "<form name='perfRankings'>";
      output += "<table border='1'>";
      output += "<tr><td><\/td>";
      for (var i = 0; i < 5; i++)
      {
         var oneFactor = parent.getFactor(i);
         oneFactor = (oneFactor == null) ? "" : oneFactor;
         output += "<th>"
                + oneFactor
                + "<\/th>";
      }
```

Next comes the assembly of subsequent rows of the table. The first column displays the name of each alternative (within `<th>` tags). The remaining columns are text boxes, all with the same name and event handler. As each row of table definition is completed, it is written to the document. After the table and form closing tags are also written, the `getdh4Performance()` function retrieves all cookie data for the fields and distributes it accordingly:

```
            for (var i = 0; i < 5; i++)
            {
               var oneAlt = parent.getAlternative(i);
               oneAlt = (oneAlt == null) ? "" : oneAlt;
               output += "<tr><th>" + oneAlt + "<\/th>";
               for (var j = 0; j < 5; j++)
               {
                  output += "<td align='center'>";
                  output += "<input type='text' size='3'"
                         + " name='ranking' value=''"
                         + " onchange='setdh4Performance()'>";
                  output += "<\/td>";
               }
               output += "<\/tr>";
               document.write(output);
               output = "";
            }
            document.write("<\/table><\/form><\/div>");
            getdh4Performance();
         </script>
      </p>
   </body>
</html>
```

dh5.html

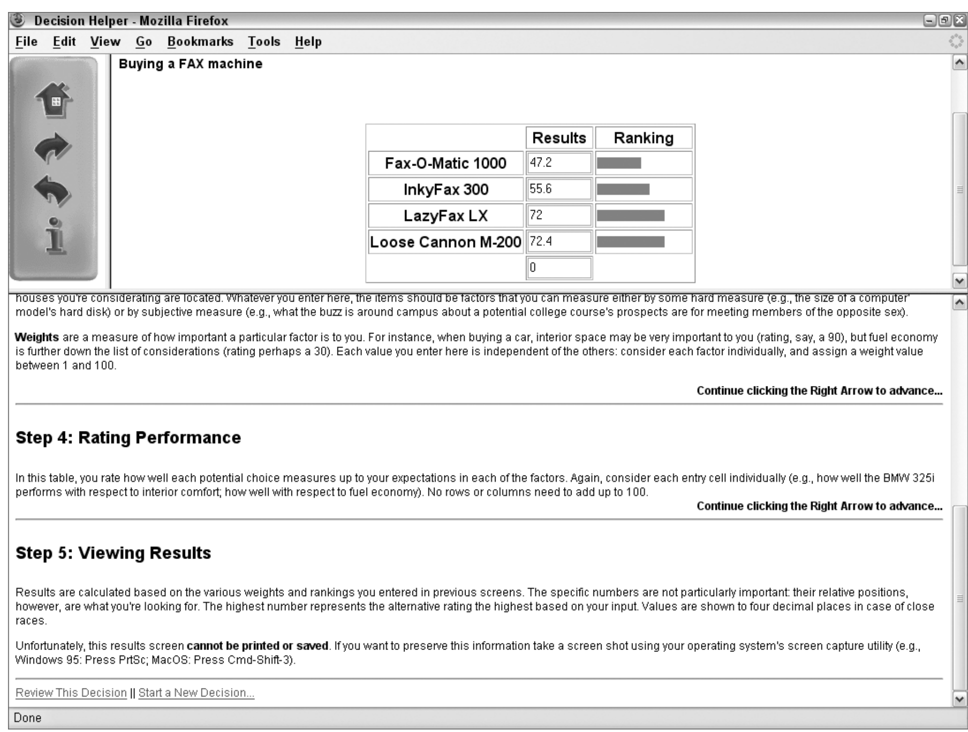
From a math standpoint, dh5.html's JavaScript gets pretty complicated. But because the complexity is attributed to the decision support calculations that turn the user's entries into results,

I treat the calculation script shown here as a black box. You're free to examine the details, if you're so inclined.

Results appear in the form of a table (see Figure 58-6) with columns showing the numeric results and an optional graphical chart.

FIGURE 58-6

The results screen for a decision.



For the purposes of this example, you only need to know a couple of things about the `calculate()` function. First, this function calls all the numeric data stored in parent cookies to fulfill values in its formulas. Second, results are tabulated and placed into a five-entry indexed array called `itemTotal[i]`. This array is defined as a global variable, so that its contents are available to scripts coming up in the `body` portion of the document:

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>DH5</title>
      <style type="text/css">
```

```
   body
   {
      font-family:Arial, Helvetica, serif;
   }
   h2
   {
      font-size:18px;
   }
   h4
   {
      font-size:14px;
   }
</style>
<script type="text/javascript" src="../../jsb-global.js"></script>
<script type="text/javascript">
   // initialize when the page has loaded
   addEvent(window, "load", initialize);
   function initialize()
   {
      parent.setTitleVar(5);
   }

   var itemTotal = new Array();

   function calculate()
   {
      var scratchpad = "";
      var importanceSum = 0;
      var oneRecord = parent.getImportance();
      var weight = new Array();
      for (i = 0; i < 5; i++)
      {
         var recLen = oneRecord.length;
         var offset = oneRecord.indexOf(".");
         scratchpad = oneRecord.substring(0,offset);
         importanceSum += (scratchpad == "" || scratchpad == "NaN")
               ? 0 : parseInt(scratchpad);
         oneRecord = oneRecord.substring(offset+1,recLen);
      }
      oneRecord = parent.getImportance();
      for (i = 0; i < 5; i++)
      {
         recLen = oneRecord.length;
         offset = oneRecord.indexOf(".");
         scratchpad = oneRecord.substring(0,offset);
         weight[i] = (scratchpad == "" || scratchpad == "NaN")
               ? 0 : parseInt(scratchpad)/importanceSum * 100;
         oneRecord = oneRecord.substring(offset+1,recLen);
      }

      for (i = 0; i < 5; i++)
      {
         oneRecord = parent.getPerformance(i);
```

```
                if (oneRecord == null)
                {
                   continue;
                }
             scratchpad = 0;
             for (var j = 0; j < 5; j++)
             {
                var recLen = oneRecord.length;
                var offset = oneRecord.indexOf(".");
                var dataPoint = oneRecord.substring(0,offset);
                scratchpad += (dataPoint != "" || dataPoint == "NaN")
                   ? parseInt(dataPoint) * weight[j] / 100 : 0;
                oneRecord = oneRecord.substring(offset+1,recLen);
             }
             itemTotal[i] = scratchpad;
          }
       }
       calculate();
    </script>
 </head>
```

Constructing this function served up many reminders about keeping data types straight. Because the data stored in cookies was in the form of strings, when it comes time to do some real math with those values, careful placement of the `parseInt()` function is essential for getting the math operators to work.

An `onload` event handler (bound to the `window` object at the beginning of the script) sends the document number to the global variable, as usual. The results displayed in this document rely heavily on stored and calculated values, so the table is constructed entirely out of JavaScript. This also means it can redisplay the decision name as part of the page:

```
<body>
   <h2>The Decision Helper</h2>
   <hr />
   <script type="text/javascript">
      document.write("<h4>"
                   + parent.getDecisionName()
                   + "<\/h4><br \/><br \/>");
      var output = "<div align='center'>";
      output += "<form name='Results'><table border='1'>";
      output += "<tr><td><\/td><th>Results<\/th><th>Ranking<\/th>";
      document.write(output);
      output = "";
```

I need to break up the discussion of the `for` loop that produces the results because there are two distinct parts of this HTML assembly. The first, shown in the following script segment, assembles the first two cells of each row of the table. The first cell contains an embedded listing of the alternative name (in `<th>` tags). To highlight the calculated values — and enable the `size` attribute to do the artificial job of truncating the floating-point number — the results are shown in text boxes. For each row, the corresponding result in `itemTotal[i]` is inserted as the `value`

attribute of the text box. The `size` attribute is set to 7, which allows the typical double-digit results, a decimal point, and four digits to the right of the decimal:

```
        for (var i = 0; i < 5; i++)
        {
           var oneAlt = parent.getAlternative(i);
           oneAlt = (oneAlt == null || oneAlt == "") ? "" : oneAlt;
           itemTotal[i] = (oneAlt == "") ? 0 : itemTotal[i];
           output += "<tr><th>" + oneAlt + "<\/th>";
           output += "<td align='center'>";
           output += "<input type='text' size='7' name='ranking' value="
                  + itemTotal[i]
                  + "><\/td>";
```

For extra pizzazz, a third column "draws" a bar chart within a 100-pixel-wide cell. The bars are actually scalings of a one-pixel-wide `.gif` file (an orange line, 12 pixels tall). A single-color `.gif` image scales to fill whatever width is assigned in the `width` attribute. This method is faster, and far better, than a more tedious method (tedious from the web page author's point of view) of creating 100 different `.gif` files, one for each possible width of the bar. I also could have used a one-pixel square `.gif` file with equal ease:

```
           output += "<td width='100'>";
           chartWidth = Math.round(itemTotal[i]);
              if (chartWidth > 0)
              {
                 output += "<img src='chart.gif' height='12' width='"
                        + chartWidth
                        + "' \/>";
              }
           output += "<\/td><\/tr>";
           document.write(output);
           output = "";
        }
        document.write("<\/table><\/form><\/div>");
     </script>
  </body>
</html>
```

dhHelp.html

The only other code worth noting in this application is in the `dhHelp.html` document, which appears in the lower frame of the window. At the end of this document are two links that call separate JavaScript functions in this document's `head` section. The `head` functions are as follows:

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Decision Helper Help</title>
      <style type="text/css">
```

```
        body
        {
           font-family:Arial, Helvetica, serif; font-size:12px;
        }
        h4
        {
           font-size:18px;
        }
        .continue, .term, .warning
        {
           text-align:right; font-weight:bold;
        }
     </style>
     <script type="text/javascript">
        function goFirst()
        {
           parent.entryForms.location = "dh1.html";
           self.location.hash = "help1";
        }
        function restart()
        {
           if (confirm("Erase current decision and start a new one?"))
           {
              parent.initializeCookies();
              parent.entryForms.location = "dh1.html";
              self.location.hash = "help1";
           }
        }
     </script>
  </head>
```

One function merely returns the user to the beginning of the sequences for both the entry screens and the help screen. The second function is a rare instance in which a confirm dialog box makes sense: It is about to erase all entered data. If the user says it's okay to go ahead, the parent window's function for initializing all cookies is called, and the navigation for both the entry and help screens goes back to the beginning.

The links at the bottom of the document (see Figure 58-6) are coded to trigger JavaScript functions (rather than navigate to URLs) and include `onmouseover` event handlers to provide more information about the link in the status bar:

```
     <a href="javascript:goFirst()"
        onmouseover="window.status='Go back to beginning
                                    to review data...';return true">
        Review This Decision</a>
     ||
     <a href="javascript:restart()"
        onmouseover="window.status='Erase current data and
                                    start over...';return true">
        Start a New Decision...</a>
```

Further Thoughts

If you've managed to follow through with this application's discussions, you will agree that it's quite a JavaScript workout. But this application proves that, without a ton of code, JavaScript provides enough functionality to add a great deal of interactivity and pseudo-intelligence to an otherwise flat HTML document.

As an alternative to using cookies for data storage, I have also implemented a version of the application that uses text boxes defined in a frame with a row height of 0. This technique further challenges the synchronization of frames during reloading when a user either resizes the browser window or navigates with the Back or Forward browser buttons. This alternate version is located on the CD-ROM for your own investigation and comparison.

Dynamic HTML also offers some possibilities for this application. The entire program can be presented in a no-frame window, with the navigation, interactive content, and instructions frames incorporated into individual positionable objects. The interactive content area can be treated almost like a slide show, with successive pages flying in from one edge.

Another Dynamic HTML possibility for added intelligence would be to create the tables in the fourth and fifth screens (`dh4.html` and `dh5.html`) with HTML, but to use JavaScript to dynamically add however many rows and columns would be needed. This would avoid the display of unneeded rows and columns for the user's decision management process.

Not only is this application instructive for many JavaScript techniques, it is also fun to play with as a user. Some financial web sites have adapted it to assist visitors with investment decisions. You can use it to ponder where to go on a dream vacation, or to help you decide the most ethical of a few paths confronting you in a personal dilemma. There's something satisfying about putting in data, turning a crank, and watching results (with a bar chart to boot!) magically appear on the screen.

CHAPTER 59

Application: Cross-Browser DHTML Map Puzzle

IN THIS CHAPTER

Applying a DHTML API

Scripting, dragging, and layering multiple elements

Accounting for incompatibilities among modern browsers

Dynamic HTML (DHTML) allows scripts to position, overlap, and hide or show elements under the control of style sheets and scripting. To demonstrate modern cross-browser DHTML development techniques, this chapter describes the details of a jigsaw puzzle game using pieces of a map of the lower 48 United States. (I think everyone would guess where Alaska and Hawaii go on a larger map of North America.) I chose this application because it enables me to demonstrate several typical tasks you might want to script in DHTML: hiding and showing elements; handling events for multiple elements; tracking the position of an element with the mouse cursor; absolute positioning of elements; changing the z-order of elements; changing element colors; and animating the movement of elements.

You may wonder why it is necessary to deal with cross-browser issues in the modern browser era, when so many things have now been standardized thanks to the W3C DOM. Although the W3C DOM has certainly bridged many compatibility gaps among browsers, it still isn't quite yet feasible to build highly interactive JavaScript applications without accounting for subtle differences between browsers. The puzzle game examined in this chapter reveals many of these subtle issues and how to go about handling them as cleanly as possible.

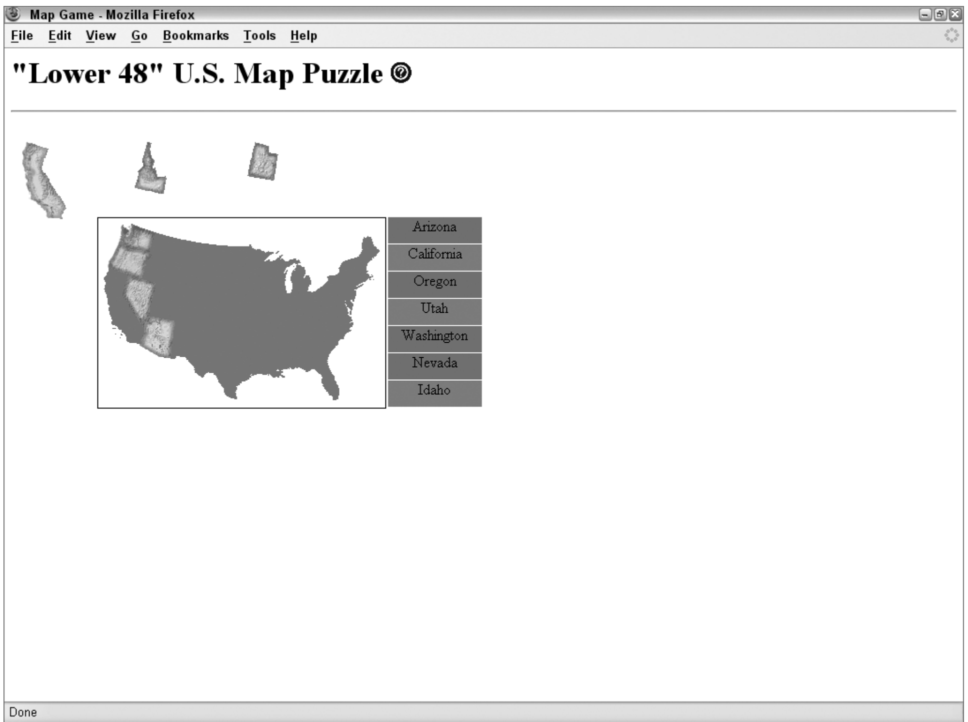
As with virtually any programming task, the example code here is not laid out as the quintessential way to accomplish a particular task. Each author brings his or her own scripting style, experience, and implementation ideas to a design. Very often, you have available several ways to accomplish the same end. If you find other strategies or tactics for the operations performed in these examples, it means you are gaining a good grasp of both JavaScript and DHTML.

The Puzzle Design

Figure 59-1 shows the finished map puzzle with the game in progress. To keep the code to a reasonable length, the example provides positionable state maps for only seven western states; you are free to expand the example to include additional states. Also, the overall design is intentionally Spartan so as to place more emphasis on the positionable elements and their scripting, rather than on fancy design.

FIGURE 59-1

The puzzle map game DHTML example.



(Images courtesy Map Resources — `www.mapresources.com`)

When the page initially loads, all the state maps are presented across the top of the puzzle area. The state labels all have a red background, and the silhouette of the continental United States has no features in it. To the right of the title is a question mark icon. A click of this icon causes a panel of instructions to glide to the center of the screen from the right edge of the browser window. A button on the panel hides it.

To play the game (no scoring or time keeping is in this simplified version), a user clicks and drags a state, with the goal of moving it into its rightful position on the silhouette. While the user drags the state, the background of its label to the right of the main map turns yellow to

highlight the name of the state being worked on. To release the state into its trial position, the user releases the mouse button. If the state is within a 4-pixel square region around its true location, the state snaps into its correct position, and the corresponding label background color turns green. If the state is not dropped close enough to its destination, the label background reverts to red, meaning that the state still needs to be placed.

After the last state map is dropped into its proper place, all the label backgrounds will be green, and a congratulatory message is displayed where the state map pieces originally lay. Should a user then pick up a state and drop it out of position, the congratulatory message disappears.

Implementation Details

Due to the number of different scripted properties being changed in this application, I decided to implement a lot of the scripting as a custom API loaded from an external `.js` file library. The library, whose code is dissected and explained in Chapter 50, "Cross-Browser Dynamic HTML Issues," contains functions for most of the scriptable items you can access in DHTML. Having these functions available simplified what would have been more complex functions in the main part of the application.

Although I frown on using global variables except where absolutely necessary, I needed to assign a few globals for this application. All of them store information about the state map currently picked up by the user and the associated label. This information needs to survive the invocations of many functions between the time the state is picked up and the time it is dropped and checked against the database of state data.

That database is another global object — a global variable that I don't mind using at all. Constructed as a multidimensional array, each record in the database stores several fields about the state, including the coordinates for its destination inside the outline map, and a Boolean field to store whether the state has been correctly placed in position.

The custom API

To begin the analysis of the code, you should be familiar with the API that is linked in from an external `.js` library file. Listing 50-3 contains that code and its description.

The main program

Code for the main program is shown in Listing 59-1. The listing is a long document, so I interlace commentary throughout the listing. Before diving into the code, however, allow me to present a preview of the structure of the document. With two exceptions (the map silhouette and the help panel), all positionable elements have their styles set through style sheets in the `head` of the document. Note the way class and id selectors are used to minimize the repetitive nature of the styles across so many similar items. After the style sheets come two `script` elements pointing to the external general scripts for the page, followed by the `script` element for the embedded scripts particular for the page. All this material is inside the `head`

element. I leave the `body` element to contain the visible content of the page. This approach is an organization style that works well for me, but you can adopt any style you like, provided various elements that support others on the page are loaded before the dependent items (for example, define a style before assigning its name to the corresponding content tag's id attributes).

LISTING 59-1

The Main Program (mapgame.htm)

```
<html>
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8">
      <title>Map Game</title>
```

Most of the positionable elements have their CSS properties established in the `style` element at the top of the document. Positionable elements whose styles are defined here include a text label for each state, a map for each state, a congratulatory message, and some of the pieces for the help panel and the background map. Note that the names of the label and state map objects begin with a two-letter abbreviation of the state. This labeling comes in handy in the scripts when synchronizing the selected map and its label.

The label objects are nested inside the background map object. Therefore, the coordinates for the labels are relative to the coordinate system of the background map, not the page. That's why the first label has a `top` property of zero.

You'll notice that I said "some of the pieces for the help panel and the background map." Although both of these are also positionable elements, scripts need to read the positions of these elements without first setting the values. Recall that in the W3C DOM, the `style` property of an object does not reveal property values that are set in remote style sheet rules. Although there are ways to read the effective style properties applied to an element from style sheets (the `currentStyle` property in IE5+ and the `window.getComputedStyle()` method in NN6+/Moz/Safari1.2+), I elected to specify the style sheet rules for the background map and help panel as `style` attributes in those two elements' tags later in the listing.

```
<style type="text/css">
   .labels { position:absolute;
      Width:100px; height:28px;
      background-color:red; border:none;
      text-align:center;
   }
   #azlabel { left:310px; top:0px;
   }
   #calabel { left:310px; top:29px;
   }
   #orlabel { left:310px; top:58px;
   }
```

```
#utlabel { left:310px; top:87px;
}
#walabel { left:310px; top:116px;
}
#nvlabel { left:310px; top:145px;
}
#idlabel { left:310px; top:174px;
}

#camap { position:absolute; left:20px; top:100px;
}
#ormap { position:absolute; left:45px; top:100px;
}
#wamap { position:absolute; left:100px; top:100px;
}
#idmap { position:absolute; left:140px; top:100px;
}
#nvmap { position:absolute; left:180px; top:100px;
}
#azmap { position:absolute; left:220px; top:100px;
}
#utmap { position:absolute; left:260px; top:100px;
}

#helpImage { height:22px; width:22px; border:none;
}
#help { background-color:#98FB98; border:none;
    top:80px; width:300px;
}
#helpTitle { text-align:center; font-weight:bold;
   margin-top:5px;
}
#helpCloseBtnForm { text-align:center;
}
#helpList { margin-right:20px;
}
#helpRule { height: 0px; border: 0px;
   border-top: 1px solid seagreen;
}

#bgmap { position:absolute;
   width:406px;
}

#usMapImage { width:306px; height:202px;
   border: 1px solid;
}

#congrats { position:absolute; left:20px; top:100px;
   width:1px; color:red; visibility:hidden;
}
```

The next statements load the external `.js` library files that contain the API described in Chapter 50 and the common external `.js` library file that contains the event-handling code used in much of the book. You'll find it in the root script directory on the CD ROM: `jsb-global.js`. I tend to load external library files before listing any other JavaScript code in the page, just in case the main page code relies on global variables or functions in its initializations.

```
<script type="text/javascript" src="../jsb-global.js"></script>
<script type="text/javascript" src="DHTMLapi.js"></script>
```

Note

One of the property assignment event-handling techniques employed in this chapter uses the more modern approach of binding events with the `addEventListener()` (NN6+/Moz/W3C) or `attachEvent()` (IE5+) methods. This modern cross-browser event-handling technique is explained in detail in Chapter 32, "Event Objects." Another event-handling technique employed in the code in this chapter and much of the book is a deliberate simplification to make the code more readable. It is generally better to use the more modern approach of binding events, as we do in this example. ■

Now comes the main script, which contains all the document-specific functions and global variables. Two event handlers discussed later are bound here. The global variables are ready to hold information about the selected state object (and associated details), as well as the offset between the position of a click inside a map object and the top-left corner of that map object. You will see that this offset is important for allowing the map to track the cursor at the same offset position within the map. And because the tracking is done by repeated calls to a function (triggered by numerous mouse events), these offset values must have global scope.

```
<script type="text/javascript">
// initialize when the page has loaded
addEvent(window, "load", init);
addEvent(window, "resize", setWinWidth);
// global declarations
var offsetX = 0;
var offsetY = 0;
var selectedObj;
var states = new Array();
var statesIndexList = new Array();
var selectedStateLabel;
```

You also have the option of creating just one, all-encompassing, custom JavaScript object in the global space, and defining properties of that object with the same names shown here for individual global variables. It means that subsequent references to these values would be written in *objectName.propertyName* form. For this example, however, I'll keep the global variables short and sweet to help you visualize how they are used in later script statements.

As you will see later in the code, an `onload` event handler for the document invokes an initialization function, whose main job is to build the array of objects containing information about each state. The fields for each `state` object record are for the two-letter state abbreviation, the full name (not used in this application, but included for use in a future version), the x and y coordinates (within the coordinate system of the background map) for the exact position of the state, and a Boolean flag to be set to `true` whenever a user correctly places a state. I will come back to the last two statements of the constructor function in a moment.

Calculating the data for the x and y coordinates required some legwork during development. As soon as I had the pieces of art for each state and the code for dragging them around the screen, I disengaged the part of the script that tested for accuracy. Instead, I added a statement to the code that revealed the x and y position of the dragged item in the status bar (rather than being bothered by alerts). When I carefully positioned a state in its destination, I copied the coordinates from the status bar into the statement that created that state record. Sure, it was tedious, but after I had that info in the database, I could adjust the location of the background map and not have to worry about the destination coordinates, because they were based on the coordinate system inside the background map.

```
// object constructor for each state; preserves destination
// position; invokes assignEvents()
function state(abbrev, fullName, x, y)
{
   this.abbrev = abbrev;
   this.fullName = fullName;
   this.x = x;
   this.y = y;
   this.done = false;
   assignEvents(this);
   statesIndexList[statesIndexList.length] = abbrev;
}
// initialize array of state objects
function initArray()
{
   states["ca"] = new state("ca", "California", 7, 54);
   states["or"] = new state("or", "Oregon", 7, 24);
   states["wa"] = new state("wa", "Washington", 23, 8);
   states["id"] = new state("id", "Idaho", 48, 17);
   states["az"] = new state("az", "Arizona", 45, 105);
   states["nv"] = new state("nv", "Nevada", 27, 61);
   states["ut"] = new state("ut", "Utah", 55, 69);
}
```

The act of creating each `state` object causes all statements in the constructor function to execute. Moreover, they were executing within the context of the object being created. That opened up channels for two important processes in this application. One was to maintain a list of abbreviations as its own array. This becomes necessary later on when the script needs to loop through all objects in the `states` array to check their `done` properties. Because the array is set up like a hash table (with string index values), a `for` loop using numeric index values is out of the question. So, this extra `statesIndexList` array provides a numerically indexed array that can be used in a `for` loop; values of that array can then be used as index values of the `states` array. Yes, it's a bit indirect, but other parts of the application benefit greatly by having the state information stored in a hash table–like array.

We now come to the functions that operate while the user interacts with the map puzzle pieces. The first function, acting as a vital behind-the-scenes utility function, is called `setSelectedMap()`. It receives as its sole parameter an event object that is of the proper type for the browser currently running (that's done in the `engage()` function, described next). `setSelectedMap()` has three jobs to do, two of which set global variables. The first global variable, `selectedObj`, maintains a reference to the layer (the puzzle piece) being dragged

by the user. At the same time, the `selectedStateLabel` variable is assigned a value that is a reference to the layer that holds the label (recall that its color changes during dragging and release). All this requires DOM-specific references that are generated through the aid of object-detecting branches of the function. The last job of this function is to set the stacking order of the selected map to a value higher than the others, so that while the user drags the map, it is in front of everything else on the page.

To assist in establishing references to the map and label layers, naming conventions of the HTML objects (shown later in the code) play an important role. In addition to event handlers being assigned to the images, the mouse events are also targeted at the image objects. To assist in associating labels with images, the image objects are assigned uppercase abbreviations of the state names. As `setSelectedMap()` begins to execute, it uses object detection to extract a reference to the element object regarded as the target of the event. To make sure that the event being processed comes from an image, the next statement checks that the `tagName` property of the event target is `img`, in which case a lowercase version of the name is assigned to the `abbrev` local variable. That `abbrev` variable then becomes the basis for the element name used in the reference to the `selectedStateLabel` object.

The presence of a value assigned to `selectedObj` becomes an important case for all three drag-related functions later. That's why the `setSelectedMap()` function nulls out the value if the event comes from some other source.

```
/**************************************************
BEGIN INTERACTION FUNCTIONS
**************************************************/

// set global reference to map being engaged and dragged
function setSelectedMap(evt)
{
   var target = (evt.target)
              ? evt.target : evt.srcElement;
   var abbrev = (target.tagName == "IMG")
              ? target.name.toLowerCase() : "";
   if (abbrev)
   {
      if (document.getElementById)
      {
         selectedObj = target;
         selectedStateLabel =
               document.getElementById(abbrev + "label");
      }
      setZIndex(selectedObj, 100);
      return;
   }
   selectedObj = null;
   selectedStateLabel = null;
   return;
}
```

Next comes the `engage()` function definition. This function is invoked by `mousedown` events inside any of the state map layers (event handler assignment code comes later). W3C DOM

browsers pass an event object as the sole parameter to the function (picked up by the `evt` parameter variable). If that parameter contains a value, it stands as the event object for the rest of the processing; but for IE, the `window.event` object is assigned to the `evt` variable. After setting the necessary object globals through `setSelectedMap()`, the next major task for `engage()` is to calculate and preserve in global variables the number of pixels within the state map layer at which the `mousedown` event occurred. By preserving these values, the `dragIt()` function makes sure that the motion of the state map layer keeps in sync with the mouse cursor at the very same point within the state map. If it weren't for taking the offset into account, the layer would jump unexpectedly to bring the top-left corner of the layer underneath the cursor. That's not how users expect to drag items on the screen.

The calculations for the offsets require a variety of DOM-specific properties. A nested object detection takes place in each assignment statement. The IE branch has some additional branching within each of the assignment statements. These extra branches cover a disparity in the way IE reports the offset properties of an event as related to window scrolling. Later calculations for positioning must take window scrolling into account, so that scrolling is factored into the preserved offset global values if there are indications that the window has scrolled and the values are being affected by the scroll (in which case the offset values go negative). The logic is confusing, and it won't make much sense until you see later how the positioning is invoked. Conceptually, all these offset value calculations may seem like a can of worms, but they are essential, and are performed compactly.

After the offsets are established, the background color of the state's label layer is set to yellow. The function ends with `return false` to make sure that the `mousedown` event doesn't propagate through the page (causing a contextual menu to appear on the Macintosh, for instance).

```
// set relevant globals onmousedown; set selected map
// object global; preserve offset of click within
// the map coordinates; set label color to yellow
function engage(evt)
{
   evt = (evt) ? evt : window.event;
   setSelectedMap(evt);
   if (selectedObj)
   {
      if (evt.pageX)
      {
         offsetX = evt.pageX - selectedObj.offsetLeft;
         offsetY = evt.pageY - selectedObj.offsetTop;
      }
      else if (evt.offsetX || evt.offsetY)
      {
         offsetX = evt.offsetX - document.body.scrollLeft;
         offsetY = evt.offsetY - document.body.scrollTop;
      }
      setBGColor(selectedStateLabel,"yellow");
      return false;
   }
}
```

The `dragIt()` function, compact as it is, provides the main action in the application by keeping a selected state object under the cursor as the user moves the mouse. This function is called repeatedly by the `mousemove` events, although the actual event-handling methodology varies with platform (precisely the same way as with `engage()`, as shown previously). Regardless of the event property detected, event coordinates (minus the previously preserved offsets) are passed the `shiftTo()` function in the API.

Before the dragging action branch of the function ends, the event object's `cancelBubble` property is set to `true`. It's important that this function operate as quickly as possible, because it must execute with each `mousemove` event. Canceling event bubbling helps in a way, but more importantly, the cancellation allows the `mousemove` event to be used for other purposes, as described shortly.

```
// move div on mousemove
function dragIt(evt)
{
   evt = (evt) ? evt : event;
   if (selectedObj)
   {
      if (evt.pageX)
      {
         shiftTo(selectedObj, (evt.pageX - offsetX),
                 (evt.pageY - offsetY));
      }
      else if (evt.clientX || evt.clientY)
      {
         shiftTo(selectedObj, (evt.clientX - offsetX),
                 (evt.clientY - offsetY));
      }
      evt.cancelBubble = true;
      return false;
   }
}
```

When a user drops the currently selected map object, the `release()` function invokes the `onTarget()` function to find out if the current location of the map is within range of the desired destination. If it is in range, the background color of the state label object is set to green, and the `done` property of the selected state's database entry is set to `true`. One additional test (the `isDone()` function call) looks to see if all the `done` properties are `true` in the database. If so, the `congrats` object is shown. But if the map object is not in the right place, the label reverts to its original red color. In case the user moves a state that was previously okay, its database entry is also adjusted. No matter what the outcome, however, the user has dropped the map, so key global variables are set to `null`, and the layer order for the item is set to zero (bottom of the heap) so that it doesn't interfere with the next selected map.

One more condition is possible in the `release()` function. As shown later in the initialization function, the `document` object's `onmousemove` event handler is assigned to the `release()` function (to compare the `onmousemove` events for the state maps, go to `dragIt()`). The reasoning behind this document-level event assignment is that no matter how streamlined the dragging

function may be, it is possible for the user to move the mouse so fast that the map can't keep up. At that point, mousemove events are firing at the document (or other object, eventually bubbling up to the document), and not the state map. If that happens while a state map is registered as the selected object, but the image is no longer the target of the event, the code performs the same act as if the user had released the map. The label reverts to red, and all relevant globals are set to null, preventing any further interaction with the map until the user mouses down again on the map.

```
// onmouseup, see if dragged map is near its destination
// coordinates; if so, mark it as 'done' and color label green
   function release(evt)
   {
      evt = (evt) ? evt : event;
      var target = (evt.target)
                 ? evt.target : evt.srcElement;
      var abbrev = (target.tagName == "IMG")
                 ? target.name.toLowerCase() : "";
      if (abbrev && selectedObj)
      {
         if (onTarget(evt))
         {
            setBGColor(selectedStateLabel, "green");
            states[abbrev].done = true;
            if (isDone())
            {
               show("congrats");
            }
         }
         else
         {
            setBGColor(selectedStateLabel, "red");
            states[abbrev].done = false;
            hide("congrats");
         }

         setZIndex(selectedObj, 0);
      }
      else if (selectedStateLabel)
      {
         setBGColor(selectedStateLabel, "red");
      }
      selectedObj = null;
      selectedStateLabel = null;
   }
```

To find out whether a dropped map is in (or near) its correct position, the onTarget() function first calculates the target spot on the page by adding the location of the bgmap object to the coordinate positions stored in the states database. Because the bgmap object doesn't come into play in other parts of this script, it is convenient to pass merely the object name to the two API functions that get the object's left and top coordinate points.

Next, the script uses platform-specific properties to get the recently dropped state map object's current location. A large `if` condition checks whether the state map object's coordinate points are within a 4-pixel square region around the target point. If you want to make the game easier, you can increase the cushion values of each coordinate point from 2 to 3 or 4. You might make the cushion values even higher if the map is intended as an educational tool for children.

If the map is within the range, the script calls the `shiftTo()` API function to snap the map into the exact destination position, and reports back to the `release()` function the appropriate Boolean value.

```
// compare position of dragged element against the destination
// coordinates stored in corresponding state object; after shifting
// element to actual destination, return true if item is within
// 2 pixels.
function onTarget(evt)
{
   evt = (evt) ? evt : event;
   var target = (evt.target)
            ? evt.target : evt.srcElement;
   var abbrev = (target.tagName == "IMG")
            ? target.name.toLowerCase() : "";
   if (abbrev && selectedObj)
   {
      var x = states[abbrev].x + getObjectLeft("bgmap");
      var y = states[abbrev].y + getObjectTop("bgmap");
      var objX, objY;
      if (selectedObj.style)
      {
         objX = parseInt(selectedObj.style.left);
         objY = parseInt(selectedObj.style.top);
      }
      if ((objX >= x-2 && objX <= x+2)
          && (objY >= y-2 && objY <= y+2))
      {
         shiftTo(selectedObj, x, y);
         return true;
      }
      return false;
   }
   return false;
}
```

A `for` loop cycles through the `states` database (with the help of the hash table values stored indirectly in the `statesIndexList` array) to see if all the done properties are set to `true`. When they are, the `release()` function (which calls the `isDone()` function) displays the congratulatory object.

```
// test whether all state objects are marked 'done'
function isDone()
{
```

```
      for (var i = 0; i < statesIndexList.length; i++)
      {
         if (!states[statesIndexList[i]].done)
         {
            return false;
         }
      }
      return true;
   }
```

The help panel is created differently than the map and label objects (details coming up in a moment). When the user clicks the Help button at the top of the page, the instructions panel slides in from the right edge of the window (see Figure 59-2). The `showHelp()` function begins the process by setting its location to the current right window edge, bringing its layer to the very front of the heap, and showing the object. To assist `moveHelp()` in calculating the center position on the screen, the `showHelp()` function retrieves (just once per showing) the property for the width of the help panel. That value is passed as a parameter to `moveHelp()`, as it is repeatedly invoked through the `setInterval()` mechanism.

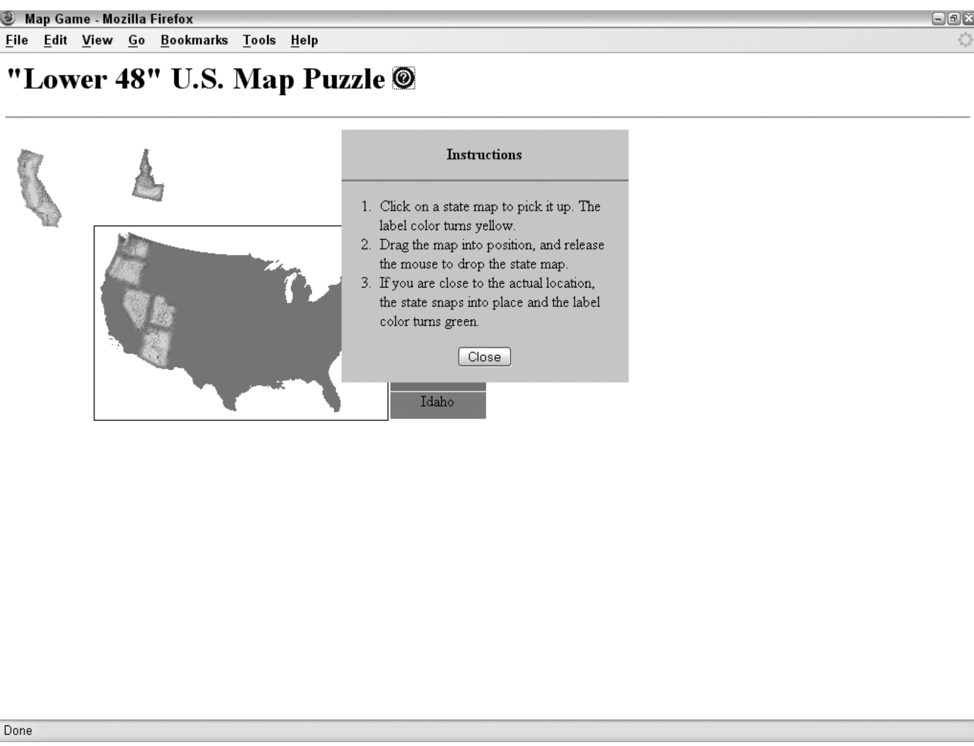
```
/***********************************************
BEGIN HELP ELEMENT FUNCTIONS
************************************************/
// initiate show action
function showHelp()
{
   var objName = "help";
   var helpWidth = 0;
   shiftTo(objName, insideWindowWidth, 80);
   setZIndex(objName,1000);
   show(objName);
   if (document.getElementById)
   {
      if (document.getElementById(objName).offsetWidth >= 0)
      {
         helpWidth = document.getElementById(objName).offsetWidth;
      }
   }
   intervalID = setInterval("moveHelp(" + helpWidth + ")", 1);
}
```

In the `moveHelp()` function, the help object is shifted in 5-pixel increments to the left. The ultimate destination is the spot where the object is in the middle of the browser window. That midpoint must be calculated each time the page loads, because the window may have been resized. The width of the `help` object, received as a parameter to the function, gets a workout in the mid-point calculation.

This function is called repeatedly under the control of a `setInterval()` method in `showHelp()`. But when the object reaches the middle of the browser window, the interval ID is canceled, which stops the animation.

FIGURE 59-2

Instruction panel slides in from left to center screen.



The help object processes a mouse event to hide the object. An extra `clearInterval()` method is called here in case the user clicks the object's Close button before the object has reached mid-window (where `moveHelp()` cancels the interval). The script also shifts the position to the right edge of the window, but this isn't absolutely necessary, because the `showHelp()` method positions the window there.

```
// iterative move helps div to center of window
function moveHelp(w)
{
   shiftBy("help",-5,0);
   var objectLeft = getObjectLeft("help");
   if (objectLeft <= (insideWindowWidth/2) - w/2)
   {
      clearInterval(intervalID);
   }
}
// hide the help div
```

```
function hideMe()
{
   clearInterval(intervalID);
   hide("help");
   shiftTo("help", insideWindowWidth, 80);
}
```

The document's `onload` event handler invokes the `init()` function, which, in turn, calls two functions and assigns the `document` object's mouse event handlers. The first function is `initArray()`, which builds the `states[]` database and assigns event handlers to the state map layers. Because the layers are defined so late in the document, initializing their events after the page has loaded is safest.

For convenience in moving the help window to the center of the browser window, the `setWinWidth()` function sets a global variable (`insideWindowWidth`) to hold the width of the browser window. This function is also invoked by the `onresize` event handler for the window to keep the value up to date.

The most important parts of the `init()` function are the event handler assignments. The trio — engaging the map on `mousedown`, dragging it on `mousemove`, releasing it on `mouseup` — are assigned to the `document` object so that the events track correctly even if the cursor speeds past a puzzle piece's edge on a fast drag action.

```
   // calculate center of window for help div
   function setWinWidth()
   {
      if (window.innerWidth)
      {
         insideWindowWidth = window.innerWidth;
      }
      else if (document.body.scrollWidth)
      {
         insideWindowWidth = document.body.scrollWidth;
      }
   }

   /*************************************************
   INITIALIZE THE APPLICATION
   *************************************************/
   // initialize application
   function init()
   {
      initArray();
      setWinWidth();
      document.onmousedown = engage;
      document.onmousemove = dragIt;
      document.onmouseup = release;
   }
   </script>
</head>
```

Now comes the part of the document that generates the visible content. The question mark icon includes an `onclick` event handler to display the Help panel.

```
<body>
   <h1>"Lower 48" U.S. Map Puzzle
      <a href="javascript:void showHelp()"
         onmouseover="status='Show help panel...';return true"
         onmouseout="status='';return true"><img alt="image"
         src="info.gif" id="helpImage" /></a>
   </h1>
```

Next come tags for all the `div` and `img` elements. The `style` attribute for the `bgmap div`, in combination with the styles set in the embedded style sheet in the `head` section, lets scripts read the positioned values to assist in calculating positions in the `onTarget()` function, as shown previously. The `bgmap` layer also contains all labels, so that if the design calls for moving the map to another part of the page, the labels follow automatically. Notice how the lowercase state abbreviations are part of the names of both the label and map layers. As you saw in a few functions shown previously, a systematic approach to object naming can offer powerful shortcuts in determining references to elements.

```
   <div id="bgmap" style=" left:100px; top:180px;">
      <img alt="image" id="usMapImage" src="us11.gif" /> 
      <div class="labels" id="azlabel">Arizona</div>
      <div class="labels" id="calabel">California</div>
      <div class="labels" id="orlabel">Oregon</div>
      <div class="labels" id="utlabel">Utah</div>
      <div class="labels" id="walabel">Washington</div>
      <div class="labels" id="nvlabel">Nevada</div>
      <div class="labels" id="idlabel">Idaho</div>
   </div>
   <img id="camap" alt="image" name="CA" src="ca.gif"
        width="47px" height="82px" border="0" />
   <img id="ormap" alt="image" name="OR" src="or.gif"
        width="57px" height="45px" border="0" />
   <img id="wamap" alt="image" name="WA" src="wa.gif"
        width="38px" height="29px" border="0" />
   <img id="idmap" alt="image" name="ID" src="id.gif"
        width="34px" height="55px" border="0" />
   <img id="azmap" alt="image" name="AZ" src="az.gif"
        width="38px" height="45px" border="0" />
   <img id="nvmap" alt="image" name="NV" src="nv.gif"
        width="35px" height="56px" border="0" />
   <img id="utmap" alt="image" name="UT" src="ut.gif"
        width="33px" height="41px" border="0" />
   <div id="congrats"><h1>Congratulations!</h1></div>
   <div id="help" onclick="hideMe()"
        style="position:absolute; visibility:hidden;">
      <p id="helpTitle">Instructions</p>
      <hr id="helpRule" />
      <ol id="helpList">
```

```
            <li>Click on a state map to pick it up.
                The label color turns yellow.</li>
            <li>Drag the map into position, and release
                the mouse to drop the state map.</li>
            <li>If you are close to the actual location,
                the state snaps into place and the label
                color turns green.</li>
         </ol>
         <form id="helpCloseBtnForm">
            <input type="button" value="Close" />
         </form>
      </div>
   </body>
</html>
```

This page has a lot of code to digest in one reading. Run the application, study the structure of the source code listing file, and re-read the previous explanations. It may take several readings for a mental picture of the application to form.

Further Thoughts

As soon as the external cross-platform API was in place, it helped frame a lot of the other code in the main program. The APIs provided great comfort in that they encouraged me to reference a complex object fully in the main code as a platform-shared value (for example, the `selectedObj` and `selectedStateLabel` global variables). At the same time, I could reference top-level elements (that is, non-nested objects) simply by their names when passing them to API functions.

It's worth pointing out that the version of the Map Puzzle example game you just saw is considerably cleaner than similar versions found in previous editions of this book. By finally abandoning legacy (version 4 and earlier) browsers and focusing on modern DOMs, it's possible to unify and simplify a lot of the code in the application. Over time, browser vendors will hopefully continue to hammer out their differences so that it is eventually possible to write JavaScript code that has no browser branching hacks at all.

But without a doubt the biggest lesson you learn from working on a project like this is how important it is to test an application on as many browsers and operating systems as possible. Unfortunately, designing a cross-platform application with any degree of complexity on one browser and having it run flawlessly on the other the first time is nearly impossible (at least for now). Be prepared to go back and forth among multiple browsers, breaking and repairing existing working code along the way, until you eventually reach a version that works on every browser that you can test.

CHAPTER 60

Application: Transforming XML Data

IN THIS CHAPTER

Mapping XML data to JavaScript objects with Ajax

Complex JavaScript data structures

Advanced array sorting

Dynamic tables

Chapter 55, "Application: Outline-Style Table of Contents," ends with an example of an interactive outliner whose data arrives from an external XML file, a feature supported by modern browsers. The application described in this chapter picks up from there.

As you recall from the Chapter 55 outline, the node structure of the XML data was used as a guide to the structure for a one-time rendering of HTML elements. There was a one-to-one correlation between XML element nesting and the HTML element nesting. Adjusting style sheet properties for displaying or hiding elements controlled all interactivity. What you're about to see here is a case for converting XML into JavaScript objects that can be used multiple times, as a convenient data source for HTML that is displayed in any number of formats. In particular, you see how JavaScript's array-sorting prowess supplies XML data with extraordinary flexibility in presentation.

You will see a lot of code in this chapter. The code is presented here as a way to demonstrate the potential for rich data handling. At the same time, the code may provide ideas for server-side processing of XML data being output to the client.

Application Overview

Understanding the data is a good place to start in describing this application. The scenario is a small American company (despite its grandiose name: GiantCo) that has divided the country into three sales regions. Two of the regions have two sales representatives, whereas the third region has three reps. The time-frame is the end of the fiscal year, when management wants to review and present the performance of each salesperson. An XML

report (`salesrpt.xml`) delivers the sales forecast and the actual sales per quarter for each sales rep. An HTML and JavaScript page is charged not only with loading the XML data, which displays the raw tabular data, but also with allowing for a variety of views and sorting possibilities, so that management can analyze performance by sales rep and region, as well as by quarter.

A server-based searching and reporting program collects the requested data and outputs each sales rep's record in an XML structure, similar to the following excerpt from the `salesrpt.xml` file:

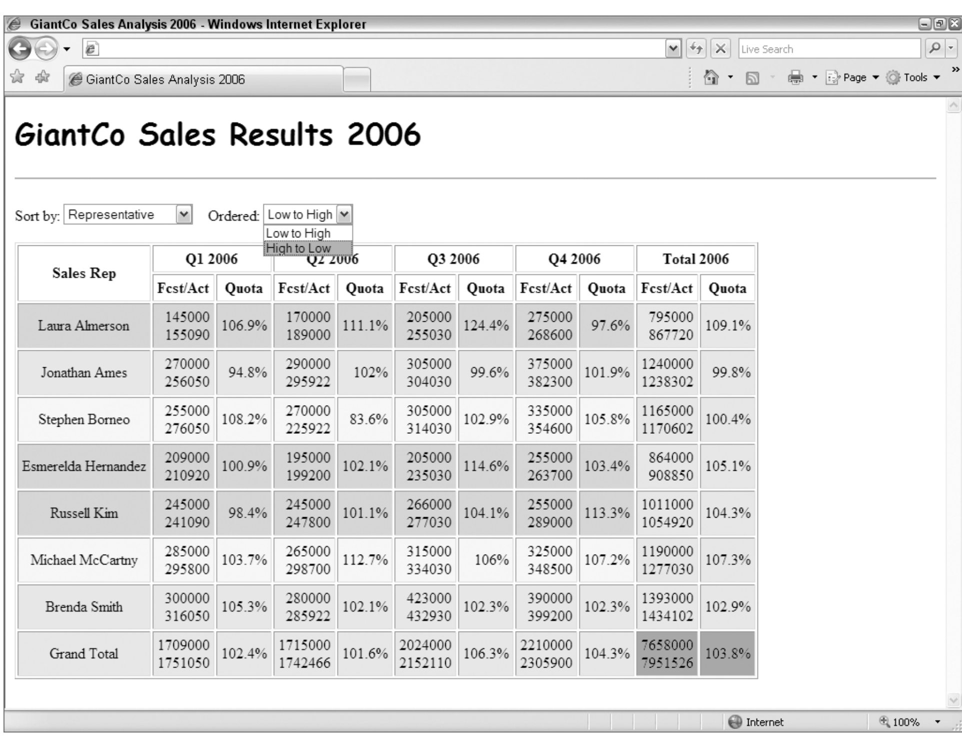
```
<salesrep>
   <employeeid>12345</employeeid>
   <contactinfo>
      <firstname>Brenda</firstname>
      <lastname>Smith</lastname>
      <email>brendas@giantco.com</email>
      <phone>312-555-9923</phone>
      <fax>312-555-9901</fax>
   </contactinfo>
   <manager>
      <employeeid>02934</employeeid>
      <firstname>Alistair</firstname>
      <lastname>Renfield</lastname>
   </manager>
   <region>Central</region>
   <salesrecord>
      <period>
         <id>Q1_2006</id>
         <forecast>300000</forecast>
         <actual>316050</actual>
      </period>
      <period>
         <id>Q2_2006</id>
         <forecast>280000</forecast>
         <actual>285922</actual>
      </period>
      <period>
         <id>Q3_2006</id>
         <forecast>423000</forecast>
         <actual>432930</actual>
      </period>
      <period>
         <id>Q4_2006</id>
         <forecast>390000</forecast>
         <actual>399200</actual>
      </period>
   </salesrecord>
</salesrep>
```

As you can see, the data consists of several larger blocks, such as contact information and a pointer to the rep's manager, and then the details of each quarterly period's forecast and actual

sales. The goal is to present the data in table form with a structure similar to the table shown in Figure 60-1. Not only is the raw data presented, but numerous calculations are also made on the results, such as the percentage of quota attained for each reporting period, plus totals along each axis of the spreadsheet-like table.

FIGURE 60-1

One view of the XML data output.



Sales Rep	Q1 2006		Q2 2006		Q3 2006		Q4 2006		Total 2006	
	Fcst/Act	Quota	Fcst/Act	Quota	Fcst/Act	Quota	Fcst/Act	Quota	Fcst/Act	Quota
Laura Almerson	145000 155090	106.9%	170000 189000	111.1%	205000 255030	124.4%	275000 268600	97.6%	795000 867720	109.1%
Jonathan Ames	270000 256050	94.8%	290000 295922	102%	305000 304030	99.6%	375000 382300	101.9%	1240000 1238302	99.8%
Stephen Borneo	255000 276050	108.2%	270000 225922	83.6%	305000 314030	102.9%	335000 354600	105.8%	1165000 1170602	100.4%
Esmerelda Hernandez	209000 210920	100.9%	195000 199200	102.1%	205000 235030	114.6%	255000 263700	103.4%	864000 908850	105.1%
Russell Kim	245000 241090	98.4%	245000 247800	101.1%	266000 277030	104.1%	255000 289000	113.3%	1011000 1054920	104.3%
Michael McCartny	285000 295800	103.7%	265000 298700	112.7%	315000 334030	106%	325000 348500	107.2%	1190000 1277030	107.3%
Brenda Smith	300000 316050	105.3%	280000 285922	102.1%	423000 432930	102.3%	390000 399200	102.3%	1393000 1434102	102.9%
Grand Total	1709000 1751050	102.4%	1715000 1742466	101.6%	2024000 2152110	106.3%	2210000 2305900	104.3%	7658000 7951526	103.8%

Just above the table are two `select` elements. These controls' labels indicate that the table's data can be sorted by a number of criteria, and that the results of each sort can be ordered in different ways. The example table offers the following sorting possibilities:

```
Representative's Name
Sales Region
Q1 Forecast
Q1 Actual
Q1 Performance
[the last three also for Q2, Q3, Q4]
Total Forecast
Total Actual
Total Performance
```

Ordering of the sorted results is a choice between "Low to High" or "High to Low." Although ordering of most sort categories is obviously based on numeric value, the sorting of the representatives' names is based on the alphabetical order of the last names. One other point about the user interface is that the design needs to signify, via table cell background color, the sales region of each representative. The colors aren't easily distinguishable in Figure 60-1, but if you open the actual example in a browser you will see the coloration. You'll have to make sure that the code you're opening in your browser is coming from an HTTP server.

Implementation Plan

Clearly, all the data needed for numerous sorted and ordered views arrives in one batch from the XML file. Despite the element- and node-referencing properties and methods of the W3C DOM, trying to use the XML elements as the sole place for scripts to sort the data each time would be impractical. For one thing, none of the elements have ID attributes — there's no need for it in the XML stored on the server database. And even if they did have IDs, how would scripts that you desire to write for generalizability make use of them unless the IDs were generated in a well-known sequence? Moreover, after a sales rep's record is rendered in the table, how easy would it be to dive back into that record and drill down for further information, such as the name of a representative's manager?

A solution that can empower the page author in this case is to use the node-walking properties and methods of the W3C DOM to assemble a JavaScript-structured database while the page loads. In other words, the conversion is performed just once during page loading, and the JavaScript version is preserved in an array (of XML "records," in this case) as a global variable. Any transformations on the data can be done from the JavaScript database with the help of additional powers of the language.

Given that route, the basic operation of the scripting of the page is schematically simple:

1. Use Ajax to load and convert the XML into an array of objects.
2. Predefine all necessary sorting functions based on properties of those objects.
3. Provide a function that rebuilds the HTML table each time data is sorted.

With this sequence in mind, look into the code that does the job.

The Code

Rather than work through the long document in source-code order, the following descriptions adhere to a more functional order. You can open the actual source code files (`salesrpt.html` and `salesrpt.js`) to see where the various functions are positioned. To best understand this application, seeing the "how" rather than the "where" is more important.

Style sheets

For the example provided on the CD-ROM, one set of style sheet rules is embedded in the HTML document. As you can see from the rule selectors, many are tied to very specific classes of table-related elements used to render the content. In a production version of this application, I would expect that there would be more and quite different views of the data available to the users, such as bar charts for each salesperson or region. Each view would likely require its own unique set of style sheet rules. In such a scenario, the proper implementation would be to use the LINK element to bring in a different external style sheet file for each view type. All style sheets could be linked in at the outset, but only the current `styleSheet` object would be enabled.

```
<style type="text/css">
   td
   {
      text-align:right
   }
   td.rep, td.grandTotalLabel
   {
      text-align:center
   }
   tr.East
   {
      background-color:#FFFFCC
   }
   tr.Central
   {
      background-color:#CCFFFF
   }
   tr.West
   {
      background-color:#FFCCCC
   }
   tr.QTotal
   {
      background-color:#FFFF00
   }
   td.repTotal
   {
      background-color:#FFFF00
   }
   td.grandTotal
   {
      background-color:#00FF00
   }
   h1
   {
      font-family:"Comic Sans MS",Helvetica,sans-serif
   }
</style>
```

One style sheet rule is essential: the one that suppresses the rendering of any XML element. That data is hidden from the user's view.

Initialization sequence

An `onload` event handler invokes the `init()` function, which triggers a series of events to get the document ready for user interaction. The URL of the XML data file is passed to the primary Ajax function (`loadXMLDoc()`), which not only loads the data, but upon successful loading assembles a JavaScript database from the XML elements.

```
// Create JavaScript object that simulates XML structure,
// plus some data transforms that will facilitate sorting
var db = new Array();

// Initialization called by onload
function init()
{
   loadXMLDoc("salesrpt.xml");
}
window.onload = init;
```

The `loadXMLDoc()` function gives priority to the native `XMLHttpRequest` object, but loads the ActiveX version for Internet Explorer prior to version 7. When the XML data has loaded successfully (the request object's `readyState` is `4` and the `status` is `200`), a copy of the XML data is assigned to a local variable, `xDoc`, primarily for the convenience of a shorter variable name. If it is further verified that the document has data, a `for` loop builds the JavaScript database (stored in the `db` global variable for use later). The assembled database is then sorted based on the current choice in the sorting `select` element. As you'll see in a moment, the `selectSort()` function then triggers the initial rendering of the table.

```
// initialize when the page has loaded
addEvent(window, "load", initialize);
```

Converting the data

The controlling factor for creating the JavaScript database is the structure of the XML data. With the complete XML document stored in a single variable, scripts can use DOM methods to look for elements bearing relevant tag names. Data for each sales rep is contained by a `salesrep` element. The number of `salesrep` elements determines how many records (JavaScript objects) are to be added to the `db` array. A call to the `getOneSalesRep()` function creates an object for each sales representative's data.

Despite the length of the `getOneSalesRep()` function, its operation is very straightforward. Most of the statements do nothing more than retrieve the data inside the various XML elements within a `salesrep` container and assign that data to a like-named property of the custom object. Following the structure of the XML example shown earlier in this chapter, you can see where some properties of a JavaScript object representing the data are, themselves, objects or arrays. For example, one of the properties is called `manager`, corresponding to the `manager` element.

That element has nested items inside; making those nested elements properties of a `manager` object is only natural. Similarly, the repetitive nature of the data within each of the four quarterly periods calls for even greater nesting: The object property named `sales` is an array, with each item of the array corresponding to one of the periods. Each period also has three properties (a period ID, forecast sales, and actual sales). Thus, the `sales` property is an array of objects.

```
function getOneSalesRep(xDoc, i)
{
   var oneRecord =
      new Object();
   var oneElem =
      xDoc.getElementsByTagName("salesrep")[i];
   oneRecord.id =
      oneElem.getElementsByTagName("employeeid")[0].firstChild.data;
   var contactInfoElem =
      oneElem.getElementsByTagName("contactinfo")[0];
   oneRecord.firstName =
      contactInfoElem.getElementsByTagName("firstname")[0].firstChild.data;
   oneRecord.lastName =
      contactInfoElem.getElementsByTagName("lastname")[0].firstChild.data;
   oneRecord.eMail =
      contactInfoElem.getElementsByTagName("email")[0].firstChild.data;
   oneRecord.phone =
      contactInfoElem.getElementsByTagName("phone")[0].firstChild.data;
   oneRecord.fax =
      contactInfoElem.getElementsByTagName("fax")[0].firstChild.data;
   oneRecord.manager =
      new Object();
   var oneMgrElem =
      oneElem.getElementsByTagName("manager")[0];
   oneRecord.manager.id =
      oneMgrElem.getElementsByTagName("employeeid")[0].firstChild.data;
   oneRecord.manager.firstName =
      oneMgrElem.getElementsByTagName("firstname")[0].firstChild.data;
   oneRecord.manager.lastName =
      oneMgrElem.getElementsByTagName("lastname")[0].firstChild.data;
   oneRecord.region =
      oneElem.getElementsByTagName("region")[0].firstChild.data;
   oneRecord.sales =
      new Array();
   var allPeriods =
      oneElem.getElementsByTagName("salesrecord")[0].childNodes;
   var temp;
   var accumForecast = 0, accumActual = 0;
   for (var i = 0; i < allPeriods.length; i++)
   {
      if (allPeriods[i].nodeType == 1)
      {
         temp =
            new Object();
         temp.period =
            allPeriods[i].getElementsByTagName("id")[0].firstChild.data;
```

```
            temp.forecast =
              parseInt(allPeriods[i].getElementsByTagName("forecast")
                      [0].firstChild.data);
            temp.actual =
               parseInt(allPeriods[i].getElementsByTagName("actual")
                      [0].firstChild.data);
            temp.quotaPct =
               getPercentage(temp.actual, temp.forecast);
            oneRecord.sales[temp.period] =
               temp;
            accumForecast += temp.forecast;
            accumActual += temp.actual;
         }
      }
      oneRecord.totalForecast = accumForecast;
      oneRecord.totalActual = accumActual;
      oneRecord.totalQuotaPct = getPercentage(accumActual, accumForecast);
      return oneRecord;
   }
   function getPercentage(actual, forecast)
   {
      var pct = (actual/forecast * 100) + "";
      pct = pct.match(/\d*\.\d/);
      return parseFloat(pct);
   }
   // End JavaScript object simulator
```

Assuming that the raw XML database stores only the sales forecast and actual dollar figures, it is up to analysis programs to perform their own calculations, such as how the actual sales compare against the forecasts. As you saw in the illustration of the rendered table, this application not only displays the percentage differences between the pairs of values, but it also provides sorting facilities on those percentages. To speed the sorting, the percentages are calculated as the JavaScript database is being accumulated, and stored as properties of each object. Percentage calculation is called upon in two different statements of the `getOneSalesRep()` function, so that the calculation is broken out to its own function, `getPercentage()`. In that function, the two passed values are massaged to calculate the percentage value, and then the string result is formatted to no more than one digit to the right of the decimal (by way of a regular expression). The value returned for the property assignment is converted to a number data type, because sorting on these values needs to be done according to numeric sorting, rather than string sorting.

You can already get a glimpse at the contribution JavaScript is making to the scripted representation of the data transmitted in XML form. By virtue of planning for subsequent calculations, the JavaScript object contains considerably more information than was originally delivered, yet all the properties are derived from “hard” data supplied by the server database.

Sorting the JavaScript database

With so many sorting keys for the user to choose from, it’s no surprise that sorting code occupies a good number of script lines in this application. All sorting code consists of two major blocks: *dispatching* and *sorting*.

The dispatching portion is nothing more than one gigantic `switch` construction that sends execution to one of the 17 (!) sorting functions that match whichever sort key is chosen in the `select` element on the page. This dispatcher function, `selectSort()`, is also invoked from the `init()` function at load time. Thus, if the user makes a choice in the page, navigates to another page, and then returns with the page still showing the previous selection, the `onload` event handler will reconstruct the table precisely as it was. When sorting is completed, the table is drawn, as you see shortly.

```
function selectSort(chooser)
{
   switch (chooser.value)
   {
   case "byRep" :
      db.sort(sortDBByRep);
      break;
   case "byRegion" :
      db.sort(sortDBByRegion);
      break;
   case "byQ1Fcst" :
      db.sort(sortDBByQ1Fcst);
      break;
   case "byQ1Actual" :
      db.sort(sortDBByQ1Actual);
      break;
   case "byQ1Quota" :
      db.sort(sortDBByQ1Quota);
      break;
   case "byQ2Fcst" :
      db.sort(sortDBByQ2Fcst);
      break;
   case "byQ2Actual" :
      db.sort(sortDBByQ2Actual);
      break;
   case "byQ2Quota" :
      db.sort(sortDBByQ2Quota);
      break;
   case "byQ3Fcst" :
      db.sort(sortDBByQ3Fcst);
      break;
   case "byQ3Actual" :
      db.sort(sortDBByQ3Actual);
      break;
   case "byQ3Quota" :
      db.sort(sortDBByQ3Quota);
      break;
   case "byQ4Fcst" :
      db.sort(sortDBByQ4Fcst);
      break;
   case "byQ4Actual" :
      db.sort(sortDBByQ4Actual);
```

```
      break;
   case "byQ4Quota" :
      db.sort(sortDBByQ4Quota);
      break;
   case "byTotalFcst" :
      db.sort(sortDBByTotalFcst);
      break;
   case "byTotalActual" :
      db.sort(sortDBByTotalActual);
      break;
   case "byTotalQuota" :
      db.sort(sortDBByTotalQuota);
      break;
   }
   drawTextTable();
}
```

Each specific sorting routine is a function that automatically works repeatedly on pairs of entries in an array (see Chapter 18, "The Array Object"). Array entries here (from the `db` array) are objects — and rather complex objects at that. The benefit of using JavaScript array sorting is that the sorting can be performed on any property of objects stored in the array. For example, sorting on the `lastName` property of each `db` array object is based on a comparison of the `lastName` property for each of the pairs of array entries passed to the `sortDBByRep()` function. But looking down a little further, you can see that the mechanism allows sorting on even more deeply nested properties, such as the `sales.Q1_2006.forecast` property of each array entry. If a property in an object can be referenced, it can be used as a sorting property inside one of these functions.

```
function sortDBByRep(a, b)
{
   if (document.getElementById("orderChooser").value == "inc")
   {
      return (a.lastName < b.lastName) ? -1 : 1;
   }
   else
   {
      return (a.lastName > b.lastName) ? -1 : 1;
   }
}
function sortDBByRegion(a, b)
{
   if (document.getElementById("orderChooser").value == "inc")
   {
      return (a.region < b.region) ? -1 : 1;
   }
   else
   {
      return (a.region > b.region) ? -1 : 1;
   }
}
```

```
function sortDBByQ1Fcst(a, b)
{
   if (document.getElementById("orderChooser").value == "inc")
   {
      return (a.sales.Q1_2006.forecast - b.sales.Q1_2006.forecast);
   }
   else
   {
      return (b.sales.Q1_2006.forecast - a.sales.Q1_2006.forecast);
   }
}
function sortDBByQ1Actual(a, b)
{
   if (document.getElementById("orderChooser").value == "inc")
   {
      return (a.sales.Q1_2006.actual - b.sales.Q1_2006.actual);
   }
   else
   {
      return (b.sales.Q1_2006.actual - a.sales.Q1_2006.actual);
   }
}
function sortDBByQ1Quota(a, b)
{
   if (document.getElementById("orderChooser").value == "inc")
   {
      return (a.sales.Q1_2006.quotaPct - b.sales.Q1_2006.quotaPct);
   }
   else
   {
      return (b.sales.Q1_2006.quotaPct - a.sales.Q1_2006.quotaPct);
   }
}
function sortDBByQ2Fcst(a, b)
{
   if (document.getElementById("orderChooser").value == "inc")
   {
      return (a.sales.Q2_2006.forecast - b.sales.Q2_2006.forecast);
   }
   else
   {
      return (b.sales.Q2_2006.forecast - a.sales.Q2_2006.forecast);
   }
}
function sortDBByQ2Actual(a, b)
{
   if (document.getElementById("orderChooser").value == "inc")
   {
      return (a.sales.Q2_2006.actual - b.sales.Q2_2006.actual);
   }
   else
```

```
      {
         return (b.sales.Q2_2006.actual - a.sales.Q2_2006.actual);
      }
   }
   function sortDBByQ2Quota(a, b)
   {
      if (document.getElementById("orderChooser").value == "inc")
      {
         return (a.sales.Q2_2006.quotaPct - b.sales.Q2_2006.quotaPct);
      }
      else
      {
         return (b.sales.Q2_2006.quotaPct - a.sales.Q2_2006.quotaPct);
      }
   }
   function sortDBByQ3Fcst(a, b)
   {
      if (document.getElementById("orderChooser").value == "inc")
      {
         return (a.sales.Q3_2006.forecast - b.sales.Q3_2006.forecast);
      }
      else
      {
         return (b.sales.Q3_2006.forecast - a.sales.Q3_2006.forecast);
      }
   }
   function sortDBByQ3Actual(a, b)
   {
      if (document.getElementById("orderChooser").value == "inc")
      {
         return (a.sales.Q3_2006.actual - b.sales.Q3_2006.actual);
      }
      else
      {
         return (b.sales.Q3_2006.actual - a.sales.Q3_2006.actual);
      }
   }
   function sortDBByQ3Quota(a, b)
   {
      if (document.getElementById("orderChooser").value == "inc")
      {
         return (a.sales.Q3_2006.quotaPct - b.sales.Q3_2006.quotaPct);
      }
      else
      {
         return (b.sales.Q3_2006.quotaPct - a.sales.Q3_2006.quotaPct);
      }
   }
   function sortDBByQ4Fcst(a, b)
   {
      if (document.getElementById("orderChooser").value == "inc")
```

```
      {
         return (a.sales.Q4_2006.forecast - b.sales.Q4_2006.forecast);
      }
      else
      {
         return (b.sales.Q4_2006.forecast - a.sales.Q4_2006.forecast);
      }
   }
   function sortDBByQ4Actual(a, b)
   {
      if (document.getElementById("orderChooser").value == "inc")
      {
         return (a.sales.Q4_2006.actual - b.sales.Q4_2006.actual);
      }
      else
      {
         return (b.sales.Q4_2006.actual - a.sales.Q4_2006.actual);
      }
   }
   function sortDBByQ4Quota(a, b)
   {
      if (document.getElementById("orderChooser").value == "inc")
      {
         return (a.sales.Q4_2006.quotaPct - b.sales.Q4_2006.quotaPct);
      }
      else
      {
         return (b.sales.Q4_2006.quotaPct - a.sales.Q4_2006.quotaPct);
      }
   }
   function sortDBByTotalFcst(a, b)
   {
      if (document.getElementById("orderChooser").value == "inc")
      {
         return (a.totalForecast - b.totalForecast);
      }
      else
      {
         return (b.totalForecast - a.totalForecast);
      }
   }
   function sortDBByTotalActual(a, b)
   {
      if (document.getElementById("orderChooser").value == "inc")
      {
         return (a.totalActual - b.totalActual);
      }
      else
      {
         return (b.totalActual - a.totalActual);
      }
```

```
}
function sortDBByTotalQuota(a, b)
{
   if (document.getElementById("orderChooser").value == "inc")
   {
      return (a.totalQuotaPct - b.totalQuotaPct);
   }
   else
   {
      return (b.totalQuotaPct - a.totalQuotaPct);
   }
}
```

For this application, all sorting functions branch in their execution based on the choice made in the "Ordered" `select` element on the page. The relative position of the two array elements under test in these simple subtraction comparison statements reverses when the sort order is reversed from low-to-high (increasing) to high-to-low (decreasing). This kind of array sorting is extremely powerful in JavaScript and probably escapes the attention of most scripters.

Constructing the table

As recommended in Chapter 41's discussion of `table` and related elements, it is often convenient to manipulate the structure of a `table` element by way of the specialized methods for tables, rather than mess with nodes and elements. The `drawTextTable()` function is devoted to employing those methods to create the rendered contents of the table below the headers (which are hard-wired in the document's HTML). Composing an 11-column table requires a bit of code, and the `drawTextTable()`'s length attests to that fact. You can tell by just glancing at the code, however, that for big chunks of it, there is a comfortable regularity that is aided by the JavaScript object that holds the data.

Additional calculations take place while the table's elements are being added to the `table` element. Column totals are accumulated during the table assembly (row totals are calculated as the object is generated, and preserved as properties of the object). A large `for` loop cycles through each (sorted) row of the `db` array; each row of the `db` array corresponds to a row of the table. Class names are assigned to various rows or cells so that they will pick up the style sheet rules defined earlier in the document. Another subtlety of this version is that the `region` property of a sales rep is assigned to the `title` property of a row. If the user pauses the mouse pointer anywhere in that row, the name of the region pops up briefly.

```
function drawTextTable()
{
   var newRow;
   var accumQ1F = 0, accumQ1A = 0, accumQ2F = 0, accumQ2A = 0;
   var accumQ3F = 0, accumQ3A = 0, accumQ4F = 0, accumQ4A = 0;
   deleteRows(document.getElementById("mainTableBody"));
   for (var i = 0; i < db.length; i++)
   {
      newRow = document.getElementById("mainTableBody").insertRow(i);
```

```
        newRow.className = db[i].region;
        newRow.title = db[i].region
                    + " Region";
        appendCell(newRow, "rep",
                   db[i].firstName +
                   " " +
                   db[i].lastName);
        appendCell(newRow, "Q1",
                   db[i].sales.Q1_2006.forecast +
                   "<br />" +
                   db[i].sales.Q1_2006.actual);
        appendCell(newRow, "Q1",
                   db[i].sales.Q1_2006.quotaPct +
                   "%");
        appendCell(newRow, "Q2",
                   db[i].sales.Q2_2006.forecast +
                   "<br />" +
                   db[i].sales.Q2_2006.actual);
        appendCell(newRow, "Q2",
                   db[i].sales.Q2_2006.quotaPct +
                   "%");
        appendCell(newRow, "Q3",
                   db[i].sales.Q3_2006.forecast +
                   "<br />" +
                   db[i].sales.Q3_2006.actual);
        appendCell(newRow, "Q3",
                   db[i].sales.Q3_2006.quotaPct +
                   "%");
        appendCell(newRow, "Q4",
                   db[i].sales.Q4_2006.forecast +
                   "<br />" +
                   db[i].sales.Q4_2006.actual);
        appendCell(newRow, "Q4",
                   db[i].sales.Q4_2006.quotaPct +
                   "%");
        accumQ1F += db[i].sales.Q1_2006.forecast;
        accumQ1A += db[i].sales.Q1_2006.actual;
        accumQ2F += db[i].sales.Q2_2006.forecast;
        accumQ2A += db[i].sales.Q2_2006.actual;
        accumQ3F += db[i].sales.Q3_2006.forecast;
        accumQ3A += db[i].sales.Q3_2006.actual;
        accumQ4F += db[i].sales.Q4_2006.forecast;
        accumQ4A += db[i].sales.Q4_2006.actual;
        appendCell(newRow, "repTotal",
                   db[i].totalForecast +
                   "<br />" +
                   db[i].totalActual);
        appendCell(newRow, "repTotal",
                   db[i].totalQuotaPct +
                   "%");
    }
```

```
   newRow = document.getElementById("mainTableBody").insertRow(i);
   newRow.className = "QTotal";
   newRow.title = "Totals";
   appendCell(newRow,
              "grandTotalLabel",
              "Grand Total");
   appendCell(newRow,
              "Q1",
              accumQ1F + "<br />" + accumQ1A);
   appendCell(newRow,
              "Q1",
              getPercentage(accumQ1A, accumQ1F) + "%");
   appendCell(newRow,
              "Q2",
              accumQ2F + "<br />" + accumQ2A);
   appendCell(newRow,
              "Q2",
              getPercentage(accumQ2A, accumQ2F) + "%");
   appendCell(newRow,
              "Q3",
              accumQ3F + "<br />" + accumQ3A);
   appendCell(newRow,
              "Q3",
              getPercentage(accumQ3A, accumQ3F) + "%");
   appendCell(newRow,
              "Q4",
              accumQ4F + "<br />" + accumQ4A);
   appendCell(newRow,
              "Q4",
              getPercentage(accumQ4A, accumQ4F) + "%");
   var grandTotalFcst = accumQ1F
                      + accumQ2F
                      + accumQ3F
                      + accumQ4F;
   var grandTotalActual = accumQ1A
                        + accumQ2A
                        + accumQ3A
                        + accumQ4A;
   appendCell(newRow,
              "grandTotal",
              grandTotalFcst + "<br />" + grandTotalActual);
   appendCell(newRow,
              "grandTotal",
              getPercentage(grandTotalActual, grandTotalFcst) + "%");
}

function appendCell(Trow, Cclass, txt)
{
   var newCell = Trow.insertCell(Trow.cells.length);
   newCell.className = Cclass;
   newCell.innerHTML = txt;
```

```
}
function deleteRows(tbl)
{
   while (tbl.rows.length > 0)
   {
      tbl.deleteRow(0);
   }
}
```

Note

As handy as it may be, in a strict W3C approach to JavaScript, you wouldn't use the `innerHTML` property since it isn't officially part of the W3C standard. However, it is often too powerful a convenience property to ignore, as much of the code throughout this book is a testament. The book does show the W3C node manipulation alternative to `innerHTML` in some examples. Refer to Chapter 29, "The Document and Body Objects," for a thorough explanation and examples of the W3C alternative to `innerHTML`. ■

Many standalone statements at the end of the `drawTextTable()` function are devoted exclusively to generating the Grand Total row, in which the accumulated column totals are entered. At the same time, the `getPercentage()` function, described earlier, is invoked several times again to derive the quota percentage for the accumulated grand total values in each quarter, as well as the complete year.

select controls

To round out the code listing for this application, the values assigned to the two `select` elements obviously have a lot to do with the execution of numerous functions in this application. Nothing magical takes place here, but you can see the extent of the detail required in assigning script-meaningful hidden values and human-meaningful text for both `select` elements. For example, dividing lines help organize the long sort key list into three logical blocks.

```
<p>Sort by:
 <select id="sortChooser" onchange="selectSort(this)">
   <option value="byRep">Representative</option>
   <option value="byRegion">Sales Region</option>
   <option value="">---------------------</option>
   <option value="byQ1Fcst">Q1 Forecast</option>
   <option value="byQ1Actual">Q1 Actual</option>
   <option value="byQ1Quota">Q1 Performance</option>
   <option value="byQ2Fcst">Q2 Forecast</option>
   <option value="byQ2Actual">Q2 Actual</option>
   <option value="byQ2Quota">Q2 Performance</option>
   <option value="byQ3Fcst">Q3 Forecast</option>
   <option value="byQ3Actual">Q3 Actual</option>
   <option value="byQ3Quota">Q3 Performance</option>
   <option value="byQ4Fcst">Q4 Forecast</option>
   <option value="byQ4Actual">Q4 Actual</option>
   <option value="byQ4Quota">Q4 Performance</option>
   <option value="">---------------------</option>
```

```
      <option value="byTotalFcst">Total Forecast</option>
      <option value="byTotalActual">Total Actual</option>
      <option value="byTotalQuota">Total Performance</option>
    </select>
       Ordered:
    <select id="orderChooser"
            onchange="selectOrder()">
      <option value="inc">Low to High</option>
      <option value="dec">High to Low</option>
    </select>
  </p>
```

Note

The property assignment event handling technique used in this example is a deliberate simplification to make the code more readable. It is generally better to use the more modern approach of binding events using the `addEventListener()` (NN6+/Moz/W3C) or `attachEvent()` (IE5+) methods. A modern cross-browser event handling technique is explained in detail in Chapter 32, "Event Objects." ■

Further Thoughts

Confining the example to just one type of view — a table of numbers — should help you grasp the important processes taking place. But with the XML data converted to JavaScript objects, you can build many other views of the same data into the same page. For example, a script could completely hide the numeric table and generate a different one that draws bar charts for each sales representative or each region (see Chapter 58, "Application: Decision Helper," for a scripted bar chart example). The horizontal axis would be the four quarters, and the vertical axis would be dollars or quota percentages. Clicking a bar opens a small window or layer to reveal more detail from the sales representative's record, such as the name of the person's manager. More `select` elements can let the user select any combination of subsets of the data in either bar chart or numeric table form to facilitate visual comparisons. You might be even more creative, and devise ways of showing the data by way of overlapping positioned elements.

CHAPTER

61

Application: Creating Custom Google Maps

IN THIS CHAPTER

The basics of Google mapping

Converting a physical address into geocoordinates

XML as a map language

Using Ajax to create Google maps

Seeing as how JavaScript is an incredibly practical technology, it's only fitting that this book should end with what may be the most practical of all JavaScript examples: a custom interactive map based upon Google's popular Google Maps tool. Google Maps is an online mapping tool that utilizes asynchronous JavaScript and XML (Ajax) to provide a traditional vector map view, along with a satellite photography view, and even a hybrid view that overlays vector street information onto the photographic satellite map. One of the features that makes Google Maps so interesting is the fact that it is customizable through a programming interface that Google has made available to web developers. Both JavaScript and XML factor heavily into Google Maps and how you go about customizing it. This chapter introduces you to the inner workings of Google Maps and guides you through a complete map customization example.

A Google Maps Primer

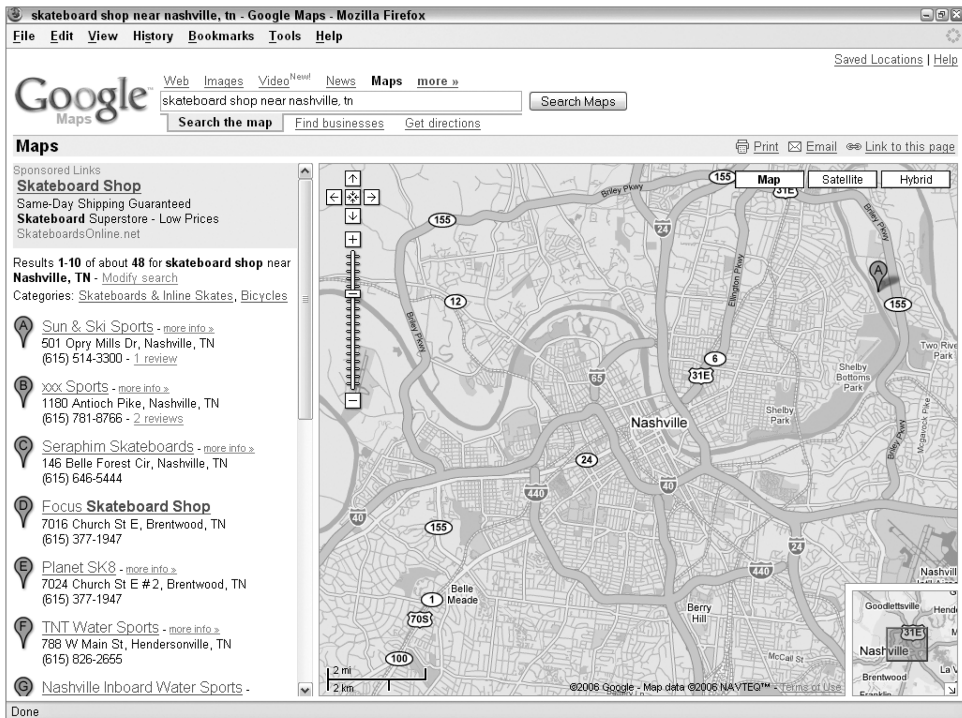
Although Google Maps is plenty of fun to tinker with purely from a user's perspective in terms of exploring the built-in map features, it's the development potential that truly makes it an exciting technology. In 2005 Google made available to the public an application programming interface (API) for Google Maps that allows web developers to build custom mapping applications. This API is the key to creating custom Google maps, such as the one examined in this chapter.

If you've never used Google Maps, take a moment to explore `http://maps.google.com`. It's important to understand the core tool before attempting to customize it to suit your own needs. When you get accustomed to how a standard Google map works, you'll have a greater appreciation of what goes into a custom map.

When Google Maps first starts, there is nothing particularly surprising about it, since all you see is a map of the United States. But keep in mind that this is Google we're talking about, the master of the search engine. Enter a search in Google Maps to watch it come to life. For example, enter "skateboard shop near nashville, tn" in the main search box, and click the Search button (feel free to insert your own hometown). Figure 61-1 shows the resulting map after searching for skateboard shops near Nashville, Tennessee.

FIGURE 61-1

The search feature in Google Maps is very effective at finding places of interest.

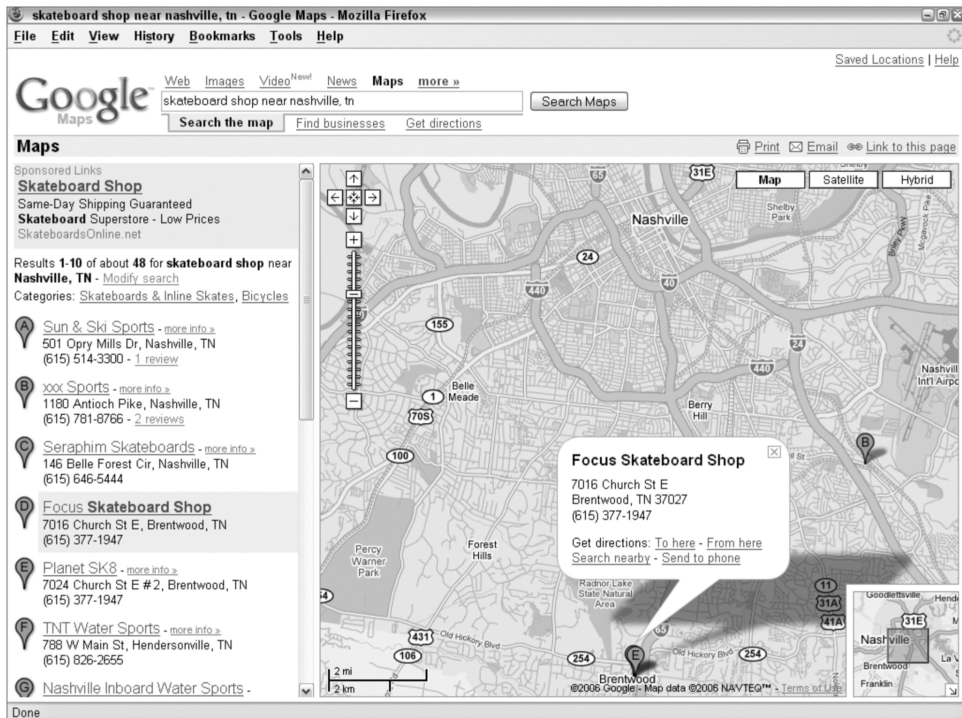


Now you're starting to see some of the power of Google Maps, and of the concept known as mashups. A *mashup* describes a web-based application that combines two existing applications to create an entirely new one. In this case, you can think of Google Maps itself as somewhat of a mashup because it combines an online map with a business directory. In reality, Google Maps has become a popular component of other mashups, some of which you find out about in a moment.

Getting back to the example, each search match in Google Maps results in a marker being placed on the map. To find out more information about one of the skateboard shops found, click it in the results list, or find its marker and click it on the map. Figure 61-2 shows how an information window appears that identifies the name of the location, along with its address. If a web site had been found for the business, it would also appear here.

FIGURE 61-2

Clicking a marker enables you to get more information about a location.



Keep in mind that at any point while using Google Maps you can click and drag the map or click the zoom control in the upper-left corner of the map. Or maybe you want to see an actual satellite photo image of the map instead of vector map graphics. Just click the Satellite button in the upper-right corner of the map. More than likely you'll find the Hybrid view to be more useful than the Satellite view. The Hybrid view displays a satellite image with streets and street names overlaid, which makes most maps much easier to read.

Note

You can use the plus (+) and minus (–) keys on your keyboard as shortcuts for zooming in and out of a map, respectively. ■

Local search is another very handy feature of Google Maps that you won't want to overlook. While viewing any map on the screen, you can click Local Search, and then perform a search on any piece of information. As an example, say you're planning on visiting Franklin, Tennessee, on business, and you're interested in doing a little cycling on the Natchez Trace Parkway while you're in town. You've already been exploring the area in Google Maps, but you don't know anything about area bicycle shops. Just enter "bicycle rental" as a local search and let Google Maps do the rest.

Google always has several interesting applications in the works. To learn more about the current experiments, visit `http://www.googlelabs.com/`.

There are a lot of other interesting features that Google Maps has to offer, but at some point I have to get to the point of this chapter, which is how to use XML to customize Google Maps. However, I'd like to leave you with a few Google Maps–related links to explore as you learn more about the service and what all it can do:

- **Google Moon:** The Moon as viewed through Google Maps; make sure you zoom all the way in (`http://moon.google.com/`)
- **Chicago Crime:** Locations of crimes committed in the Chicago metropolitan area (`http://www.chicagocrime.org/`)
- **Weather Bonk:** Real-time mapped weather information and web cams (`http://www.weatherbonk.com/`)
- **Google Maps Directory:** Directory of Google Maps applications (`http://www.gmdir.com/`)

This list provides plenty of interesting sites to keep you busy for a while. When you're finished traveling around these maps, prepare to take the next logical step: create your own mapping application.

Google Maps Customization Basics

Customizing Google Maps to create a custom mapping application is fairly straightforward, but it does require a few preliminary steps before you can even start thinking about JavaScript code or web pages.

- First off, although Google exposes the Google Maps API for anyone to use, the company requires you to obtain a special API key in order to use the API in your own web pages.
- Secondly, you need to understand some basics about the Google Maps API and how it is used to create and customize maps through JavaScript code.
- Finally, locations in Google Maps are specified through geocoordinates, which you probably don't have readily available for the locations you're interested in mapping. For this reason, you need to learn how to obtain the latitude and longitude of a location based upon its physical (mailing) address.

The next few sections explore each of these topics in more detail.

Getting your own API key

The easiest part of the map creation process is obtaining your own Google Maps API key. This virtual key is literally your key to being able to view, test, and share your map creations with others. A Google Maps API key is completely free of charge; presumably, it is required so that Google can keep close tabs on how their technology is being used.

You can sign up and obtain an API key by visiting `http://www.google.com/apis/maps/signup.html`. This page requires you to enter the URL of the web site to which you plan on publishing your maps. A single API key gives you the ability to publish maps to one folder on one web server. As an example, if I register an API key for `http://www.example.com/maps`, I can publish maps only to the `maps` folder on my web server, and nowhere else. (Well, technically I can publish them wherever I want on my web server, but they'll only work if I place them in the `maps` folder.) Make sure you actually publish your maps to the URL you specify when obtaining your API key. If you want to create different maps and store them in different folders, you'll need to obtain a different API key for each unique base URL.

The API key is provided to you as a long text code that you will cut and paste into your mapping code later. For now, copy and paste the code into a text file and save it for later.

Inside the Google Maps API

The Google Maps API consists of a set of JavaScript functions that you call to create and manipulate a map within an HTML web page. Every map must have a standard `<script>` tag in the head of the HTML document that references the Google Maps API and specifies your API key. Following is an example of how this tag is coded:

```
<script
src="http://maps.google.com/maps?file=api&amp;v=2&amp;key=ABQIAAAAY7yQRiC6EI41ks
qEpofW3BRdYuPG5WBWUHzdnWjZCVWXDwCyyRS26iZXfNWk97uGn615z44
   _i9klyg&amp;sensor=false" type="text/javascript"></script>
```

The API key is too long to fit on one line of printed text in this book, which is why you see it broken across multiple lines. In your HTML code, be sure not to add any spaces or line breaks in the API key. After placing the API key script code in the head of a web page, you can begin calling API functions to create the map itself. Unlike the API key script code, the actual map script code goes in the body of the document.

Note

As you will quickly see, the Google Maps API names its functions and objects with identifiers starting with a capital `G`. If you see a reference in the example code here that begins with `G`, it is not part of the JavaScript language or the browser's DOM, but rather something provided by Google's API. ■

As of this writing, the Google Maps API is at version 2. Version 1 is no longer supported, so this chapter's original application, written for an earlier edition of this book, no longer works. However, a copy of the version 1 code has been included in the CD-ROM, so that if you have to upgrade from version 1 to version 2, you'll have some clues as to the direction you should go. Be aware that the Google Maps API is updated often, sometimes weekly. Revisit your web site frequently and test to see if an API update has affected your map.

The `GMap2()` function is called to create a new map. This function expects, as its only argument, the element on the page that is to contain the map:

```
var map = new GMap2(document.getElementById("myMap"));
```

The map container element is typically a div element, as in the following example code:

```
<div id="myMap" style="width:800px; height:600px"></div>
```

Notice that the ID of the div element is myMap, which corresponds to the ID of the element passed into the GMap2() function. This is how the map gets connected to a container element on the web page.

The Google Maps API offers several different user interface controls for you to use. The standard control that consists of full-sized pan/zoom buttons is represented by the GLargeMapControl object. There is also a smaller control called GSmallMapControl, as well as a control with no pan features at all called GSmallZoomControl. To set the control for a map, you call the addControl() method on the newly created map, like this:

```
map.addControl(new GLargeMapControl());
```

There is also a control that determines whether or not you can switch between the different map views (Map, Satellite, and Hybrid). To enable this control, add the GMapTypeControl object to the map with the following piece of code:

```
map.addControl(new GMapTypeControl());
```

With the map and controls in place, you're ready to set the default area for the map, which is based upon a geocoordinate as well as a zoom level. You learn how to find a geocoordinate for a location in the next section. For now, just understand that it consists of two numbers that represent the latitude and longitude of a location. Furthermore, the zoom level of the map is set to an integer number that you will likely need to experiment with to find a zoom level that suits your specific map. Following is the code that centers and zooms a map at a specific geocoordinate and zoom level:

```
var point = new GLatLng(36.071689, -88.053171);
map.setCenter(point, 8);
```

The last step in creating a custom map is creating the markers on the map, which can also involve using custom icon images, if you so choose. Custom icons in Google Maps actually consist of two images: the icon image and a shadow for the icon. You learn more about these images a bit later in the chapter. For now, take a look at the following code, which shows how to create a marker with a custom icon, as well as how to set a listener function that opens an information window when the marker is clicked.

```
var baseIcon = new GIcon();
baseIcon.iconSize = new GSize(48, 48);
baseIcon.shadowSize = new GSize(70, 48);
baseIcon.iconAnchor = new GLatLng(38, 24);
baseIcon.infoWindowAnchor = new GLatLng(20, 24);
var icon = new GIcon(baseIcon);
icon.image = "ghosticon.png";
icon.shadow = "ghosticon_sh.png";
var marker = new GMarker(point, icon);
```

```
GEvent.addListener(marker, "click",
   function()
   {
      // code to build the HTML goes here
      marker.openInfoWindowHtml(placeHTML);
   }
);
```

The main thing to note in this code is how the various icon sizes are specified, including the relative offset of the icon anchor and the information window anchor. These offsets determine how the marker icon image is positioned relative to the location on the map, as well as the offset of the information window with respect to the marker. The last seven lines of code look messier than they truly are; all they do is open an information window and display within it an HTML element, which you might have to build. You'll see an example later.

Obtaining the geocoordinates of a location

Although Google Maps is typically queried based upon the mailing address of a location, when you deal directly with the Google Maps API, you are required to use more accurate location data. More specifically, you must use a location's latitude and longitude, also known as its geocoordinates, when specifying its position to the Google Maps API. Since you probably haven't committed to memory the latitude and longitude of your favorite hang outs, you'll likely need to use a tool to find out the coordinates of any address that you want to include on a map. Fortunately, exactly such a tool exists in the form of the geocoder.us web site, located at `http://geocoder.us`.

This web site enables you to enter an address much as you would enter it in Google Maps. Assuming the address is successfully found, geocoder displays the latitude and longitude of the address, which you can then use in Google Maps to specify the exact position of the location.

Another decent option for obtaining the latitude and longitude of a location is to use a GPS receiver. Of course, you'll have to physically travel to the location so that its latitude and longitude are registered on your GPS device.

Note

If you have trouble finding a match for an address on the geocoder.us site, you might consider just getting close by using a known nearby address. ■

Designing a Google Maps Mashup

There are all kinds of interesting prospects out there when it comes to dreaming up your own custom Google Maps application. The example application that you work through in the remainder of this chapter is based upon the notion of organizing a collection of relevant places on a map. More specifically, you find out how to create a haunted map that reveals several haunted places in the state of Tennessee.

In this example, the custom mapping application involves several facets:

- Developing an XML language for coding haunted places
- Interacting with the Google Maps API to open and display a map centered on a certain place
- Using the Google Maps API to create custom markers for the places
- Using CSS with generated HTML to present a unique information window for each place when it is clicked in Google Maps

Although this example map is certainly intended to be for fun, you could just as easily plug in more serious data and create a map that serves a vital purpose. That's what's so great about data-driven applications such as Google Maps.

Developing a Custom Map Document

Developing a custom Google Maps application first involves deciding on a format for the data that you plan on feeding into Google Maps. Since the example application you're working through in this chapter involves mapping haunted places, it's worth considering what pieces of information you might want to code into an XML document to map a place that is supposedly haunted:

- Place name
- Geocoordinates (latitude and longitude)
- Physical address (mailing address)
- Thumbnail image
- Summary of why the place is considered haunted
- Link to a web site containing more information

These pieces of information are sufficient to describe any haunted place for the purposes of displaying information about it in Google Maps. The trick is then determining the best way to effectively represent this information in the context of an XML document. Listing 61-1 contains a portion of a document that solves this problem.

LISTING 61-1

Modeling Haunted Place Data in XML

```
<?xml version="1.0"?>
<places>
   <place>
      <location lat="36.585414" long="-87.061648" />
      <description>
         <name>Bell Witch Cave</name>
```

continued

LISTING 61-1 *(continued)*

```
         <url>http://www.bellwitchcave.com/</url>
         <address>430 Keysburg Rd</address>
         <address2>Adams, TN 37010</address2>
         <img>bellwitch.jpg</img>
         <summary>Without a doubt the most famous Tennessee ghost story, the
         Bell Witch tormented the Bell family in the early 1800s. Legend has
         it that the witch was even responsible for the death of John Bell,
         then owner of the Bell farm. Bell Witch sightings exist to this day, and
         the witch is thought to live in a cave on the property that has become
         known as Bell Witch Cave.</summary>
      </description>
   </place>
   <place>
      <location lat="35.904865" long="-86.862472" />
      <description>
         <name>Carnton Mansion</name>
         <url>http://www.carnton.org/</url>
         <address>1345 Carnton Ln</address>
         <address2>Franklin, TN 37064</address2>
         <img>carnton.jpg</img>
         <summary>The Battle of Franklin represents one of the bloodiest
         Confederate losses in the Civil War, and Carnton Mansion played host
         to many of the dead and wounded soldiers. Five generals died on the
         front porch of the mansion, and one of them has since reportedly been
         seen riding a horse through the fields around the mansion. There is
         also legend of a restless soldier who has been seen and heard walking
         through the house.</summary>
      </description>
   </place>
...
</places>
```

The listing contains a portion of the code for the places.xml example document that houses XML data for the haunted places example mapping application. The document is laid out as a series of <place> elements, each representing an individual haunted place. These elements appear within the root <places> element. Inside of each <place> element is a location element that contains the latitude and longitude and a <description> element that describes more details about the place. The <name>, <url>, <address>, <address2>, <img>, and <summary> elements all combine to provide data that will appear in the information window in Google Maps when the user clicks on a marker.

Although the description information is important for adding context to the map, all Google Maps really cares about is the latitude and longitude stored within the <location> element. Everything else in the document is just supplementary data to provide additional information when the user clicks one of the markers on the map.

Note

It's important to point out that the haunted place images, CSS file, and XML file in this example document are specified without any path information, which means that they are expected to reside in the same directory as the HTML document. ■

Hacking Together a Custom Google Map

Developing a customized map for Google Maps is really a three-step process:

1. You put together a suitable XML document to house your map data, which you've already done.
2. Then you create an HTML web page with requisite JavaScript code that creates the map and handles the majority of the work in running the application.
3. Finally, you create a function that is responsible for generating the HTML content to be displayed in an information window when a marker is clicked on the map. You might also choose to develop an external CSS style sheet for formatting the content.

Displaying the custom map

Earlier in the chapter, I mentioned that Google Maps relies on Ajax as a critical part of its design. You now get to see how JavaScript code is used to manipulate XML into creating a custom interactive map, all with the power of asynchronous Ajax code.

Cross-Reference

Refer to Chapter 39, "Ajax, E4X, and XML," if you need to brush up on Ajax. ■

Listing 61-2 contains the complete code for the `hauntmap.html` web page.

LISTING 61-2

Taming Haunted Google Maps with JavaScript

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
   <head>
      <meta http-equiv="content-type" content="text/html;charset=utf-8" />
      <title>Haunted Places Map</title>
      <script
src="http://maps.google.com/maps?file=api&amp;v=2&amp;key=ABQIAAAAY7RCdttCOWnTdYI37l
6JMRR6nvXvyb_76ZEGxDpMxLfUsiDxfRTlqj8Soxj8oOKiIIu99sdtoRfObw&amp;sensor=false"
type="text/javascript"></script>
   </head>
   <body onunload="GUnload()">
      <h1>Haunted Places Map</h1>
```

continued

LISTING 61-2 *(continued)*

```
<hr />
<div id="hauntedMap"  style="width:800px; height:600px"></div>
<script type="text/javascript">
<!--
   // Initialize the map and icon variables
   var map = new GMap2(document.getElementById("hauntedMap"));
   map.addControl(new GLargeMapControl());
   map.addControl(new GMapTypeControl());
   var point = new GLatLng(36.071689, -88.053171);
   map.setCenter(point, 8);
   var baseIcon = new GIcon();
   baseIcon.iconSize = new GSize(48, 48);
   baseIcon.shadowSize = new GSize(70, 48);
   baseIcon.iconAnchor = new GLatLng(38, 24);
   baseIcon.infoWindowAnchor = new GLatLng(20, 24);

   function buildHTML(placeName, placeURL, placeAddress1,
                      placeAddress2, placeIMG, placeSummary)
   {
      var theHTML =  '<!DOCTYPE html><html><head>';
      theHTML += '<meta http-equiv="content-type" ' +
                 'content="text/html;charset=utf-8"> ';
      theHTML += '<title>' +
                 placeName +
                 '</title> ';
      theHTML += '<link rel="stylesheet" ' +
                 'href="hauntedPlaces.css" type="text/css"> ';
      theHTML += '</head> <body>';
      theHTML += '<table class="hauntedData"><tr> ';
      theHTML += '<td colspan="2" class="hauntedURL"> ';
      theHTML += '<a href="' +
                 placeURL +
                 '">' +
                 placeName +
                 '</a> ';
      theHTML += '</td> </tr>';
      theHTML += '<tr class="hauntedLocation"> ';
      theHTML += '<td> <div>' +
                 placeAddress1 +
                 '<br />' +
                 placeAddress2 +
                 '</div> <hr /> ';
      theHTML += '<div>' +
                 placeSummary +
                 '</div> </td> ';
      theHTML += '<td class="hauntedImage"> <div> ';
      theHTML += '<img src="' +
                 placeIMG +
                 '" />';
```

```
   theHTML += '</div> </td>';
   theHTML += '</tr></table></body></html>';

   return theHTML;
}


function createMarker(place, point, description)
{
   // Create the custom icon
   var icon = new GIcon(baseIcon);
   icon.image = "ghosticon.png";
   icon.shadow = "ghosticon_sh.png";

   // Create the marker and register the info window listener function
   var marker = new GMarker(point, icon);
   GEvent.addListener(marker, "click",
      function()
      {
         // Get values from XML file to send to buildHTML function
         var placeName = description.getElementsByTagName("name")
                         [0].firstChild.nodeValue;
         var placeURL = description.getElementsByTagName("url")
                        [0].firstChild.nodeValue;
         var placeAddress1 = description.getElementsByTagName("address")
                             [0].firstChild.nodeValue;
         var placeAddress2 = description.getElementsByTagName("address2")
                             [0].firstChild.nodeValue;
         var placeIMG = description.getElementsByTagName("img")
                        [0].firstChild.nodeValue;
         var placeSummary = description.getElementsByTagName("summary")
                            [0].firstChild.nodeValue;
         placeHTML = buildHTML(placeName, placeURL, placeAddress1,
                               placeAddress2, placeIMG, placeSummary);
         marker.openInfoWindowHtml(placeHTML);
      }
   );
   return marker;
}

// Open and process the haunted places XML document
GDownloadUrl("places.xml", function(data,responseCode)
{
   var xmlDoc = GXml.parse(data);
   var hauntedPlaces =
       xmlDoc.documentElement.getElementsByTagName("place");
   var hauntedLocations =
       xmlDoc.documentElement.getElementsByTagName("location");
   var descriptions =
       xmlDoc.documentElement.getElementsByTagName("description");
   for (var i = 0; i < hauntedPlaces.length; i++)
   {
      var hauntedPoint = new GLatLng(
```

continued

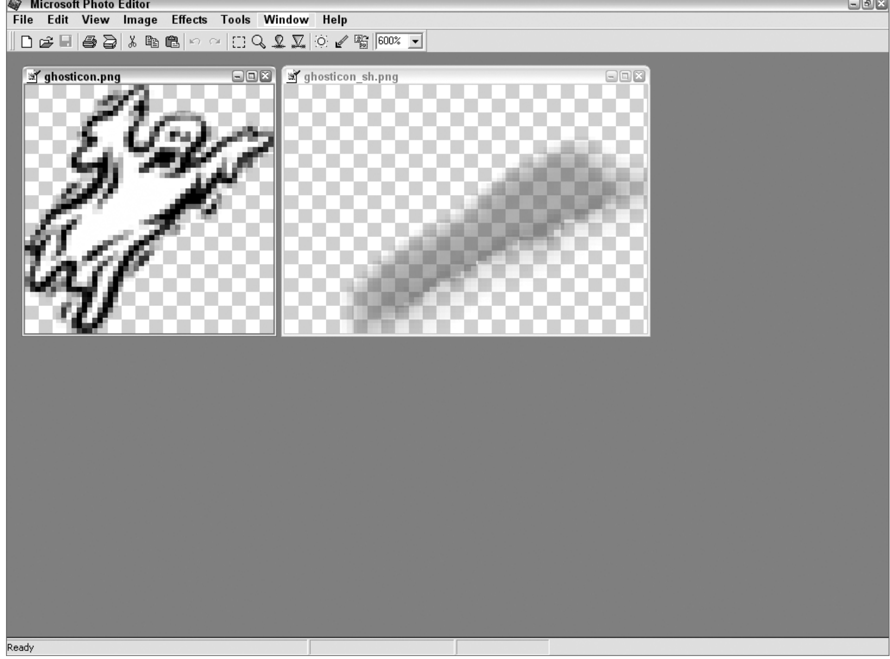
LISTING 61-2 *(continued)*

```
                    parseFloat(hauntedLocations[i].getAttribute("lat")),
                    parseFloat(hauntedLocations[i].getAttribute("long"))
                );
                var marker = createMarker(hauntedPlaces[i], hauntedPoint,
                                          descriptions[i]);
                map.addOverlay(marker);
            }
        }
        );
    //-->
    </script>
  </body>
</html>
```

The first part of this web page should be somewhat familiar to you, thanks to the earlier primer on the Google Maps API. Most of this code follows the general template you saw earlier, regarding how a map is created, controls are added, a default view is established, and so on. You'll notice that two marker images are required in the script: one for the ghost marker icon and one for its shadow. Figure 61-3 shows the two marker images used in this example application.

FIGURE 61-3

A marker actually consists of two images: the marker icon and its shadow.



The figure shows how a marker that appears on a Google Maps map actually consists of two icons: the marker image and a shadow image with transparency. When combined, these two icons provide a clever visual trick that makes the markers appear to rise off of the map in 3-D.

Once the map and its icon variables are initialized, you'll see the function `buildHTML` that generates the HTML string that is used later by the `createMarker()` function. You'll notice that an external CSS file is specified so that each information window will have the same look and feel.

Each custom icon is used to create a custom marker in the `createMarker()` function that also establishes the information window. Additionally, this function extracts node data from the XML description node that was grabbed in the `GDownloadUrl` function (described next) and calls the `buildHTML` function to generate the custom HTML string from the data in the XML document. This code is important because it binds a special listener function to the `onclick` event for the newly created marker. This results in the display of an information window in response to the user clicking a marker.

The last big chunk of code in the HTML document is the `GDownloadUrl` where the XML document is processed. The XML file is opened and its relevant nodes (`place`, `location`, and `description`) are grabbed. A loop is then entered that cycles through the haunted places, creating a marker on the map for each one via calls to the previously mentioned `createMarker()` function.

The HTML document has now created the custom map in Google Maps, complete with unique markers and information windows ready to spring into action when the markers are clicked.

Styling a custom information window

The CSS style sheet for the haunted places map application has only one responsibility: format the text data for the haunted place and display it next to the thumbnail image of the place. Listing 61-3 contains the code for the `hauntedPlaces.css` style sheet.

LISTING 61-3

Formatting the XML Data for the HTML String

```
body              {
   font-family:Georgia; serif;
}

img               {
   width:175px; height:175px;
}
```

continued

LISTING 61-3 *(continued)*

```
:link             {
   color:black;
}

.hauntedData      {
   width:450px; height:200px;
   text-align:left;
   font-size:6pt;
}

.hauntedImage     {
   text-align:right;
}

.hauntedLocation  {
   vertical-align:top;
}

.hauntedURL       {
   font-weight:bold; font-size:8pt;
}
```

Testing the finished map

Figure 61-4 shows the haunted places map example upon first being loaded and viewed in Firefox.

Not surprisingly, the application starts out with all the ghost place markers in view on the map. This is no accident, by the way — I carefully selected the initial viewing area and zoom level of the map so that you could see all of the places. Don't forget that what makes your custom Google Maps application so interesting is that you can still use all of the familiar navigational features built into Google Maps. You can drag the map around to view other areas, as well as

zoom in and out on the places and their surroundings. You can also switch back and forth between Map, Satellite, and Hybrid views. Perhaps most importantly, you can click any of the place markers to get information about each specific place. Figure 61-5 shows the haunted places map, zoomed in, with one of the place information windows open.

This figure reveals how compelling the merger of a geographical XML database and Google Maps can be. You've effectively taken an extremely powerful online tool and customized it with your own data. The net effect is an application that would be unbelievably difficult to create from scratch.

FIGURE 61-4

The haunted places map begins with all the places in view.

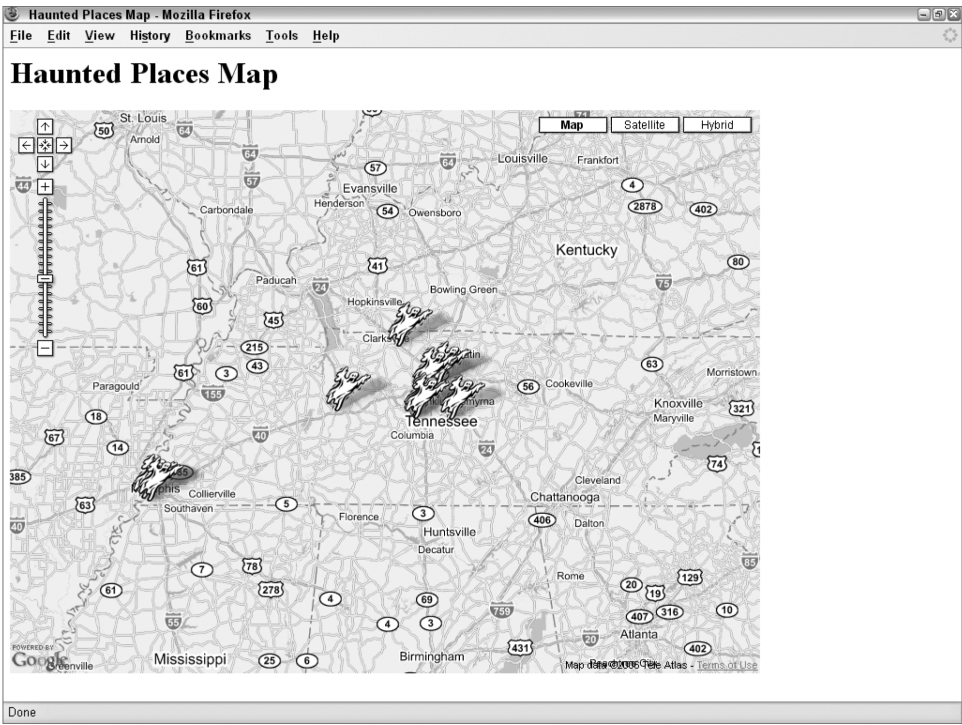
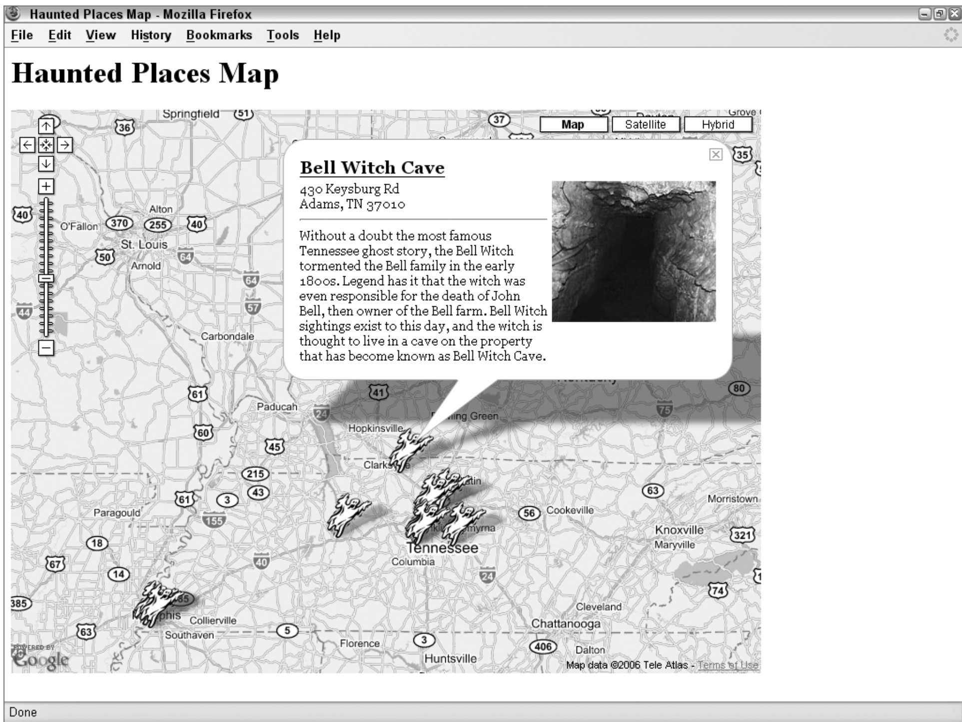


FIGURE 61-5

Clicking a ghost marker reveals additional information about a haunted place.



Further Thoughts

Who would've thought a few years ago that the Web would make it possible to view up-to-date crime statistics on a map of where they occurred? This is only the tip of the iceberg in terms of how an application such as Google Maps can improve our interactions with the world. And thanks to Google opening up their API to developers, you can be a part of it all. Perhaps you have some unique local knowledge that you could add to a map others would find useful. Hopefully the haunted places example in this chapter will spark you to try creating something unique using JavaScript and Google Maps.

Think about any data you may have access to that could somehow be presented geographically. Massage the data into an XML format, and you'll have your very own Google Maps mashup.

Part V

Appendixes

IN THIS PART

Appendix A
JavaScript and Browser Objects Quick Reference

Appendix B
What's on the CD-ROM

APPENDIX

JavaScript and Browser Objects Quick Reference

JavaScript and Browser Objects Quick Reference

String 15

constructor
length
prototype

anchor("*anchorName*")
big()
blink()
bold()
charAt(*index*)
charCodeAt([i])
concat(*string2*)
fixed()
fontcolor(*#rrggbb*)
fontsize(*1to7*)
fromCharCode(*n1...*)*
indexOf("*str*" [,i])
italics()
lastIndexOf("*str*" [,i])
link(*url*)
localeCompare()
match(*regexp*)
replace(*regexp,str*)
search(*regexp*)
slice(i,j)
small()
split(*char*)
strike()
sub()
substr(*start,length*)
substring(*intA, intB*)
sup()
toLocaleLowerCase()
toLocaleUpperCase()
toLowerCase()
toString()
toUpperCase()
valueOf()

*Method of the static String object.

Regular Expressions 45

global
ignoreCase
input[ESC]
lastIndex
multiline[EMSC]
lastMatch[EMSC]
lastParen[EMSC]
leftContext
prototype
rightContext
source
$1...$9

compile(*regexp*)
exec("*string*")*
test("*string*")

*Returns array with properties: index, input, [0],...[n].

Array 18

constructor
length
prototype

concat(*array2*)
every(*func*[, *thisObj*])[M1.8S3O9.5C]
filter(*func*[, *thisObj*])[M1.8S3O9.5C]
forEach(*func*[, *thisObj*])[M1.8S3O9.5C]
indexOf(*func*[, *thisObj*])[M1.8S3O9.5C]
join("*char*")
lastIndexOf(*func*[, *thisObj*])[M1.8S3O9.5C]
map(*func*[, *thisObj*])[M1.8S3O9.5C]
pop()
push()
reduce()[MS4]
reduceRight()[MS4]
reverse()
shift()
slice(i,[j])
some(*func*[, *thisObj*])[M1.8S3O9.5C]
sort(*compareFunc*)
splice(i,j[,*items*])
toLocaleString()
toString()
unshift()

Function 23

arguments
caller
constructor
length
prototype

apply(*this, argsArray*)
call(*this*[,*arg1*[,...*argN*]])
toString()
valueOf()

Date 17

constructor
prototype

getFullYear()
getYear()
getMonth()
getDate()
getDay()
getHours()
getMinutes()
getSeconds()
getTime()
getMilliseconds()
getUTCFullYear()
getUTCMonth()
getUTCDate()
getUTCDay()
getUTCHours()
getUTCMinutes()
getUTCSeconds()
getUTCMilliseconds()
setYear(*val*)
setFullYear(*val*)
setMonth(*val*)
setDate(*val*)
setDay(*val*)
setHours(*val*)
setMinutes(*val*)
setSeconds(*val*)
setMilliseconds(*val*)
setTime(*val*)
setUTCFullYear(*val*)
setUTCMonth(*val*)
setUTCDate(*val*)
setUTCDay(*val*)
setUTCHours(*val*)
setUTCMinutes(*val*)
setUTCSeconds(*val*)
setUTCMilliseconds(*val*)
getTimezoneOffset()
toDateString()
toGMTString()
toLocaleDateString()
toLocaleString()
toLocaleTimeString()
toString()
toTimeString()
toUTCString()
parse("*dateString*")*
UTC(*dateValues*)*

*Method of the static Date object.

Math* 16

E
LN2
LN10
LOG2E
LOG10E
PI
SQRT1_2
SQRT2

abs(*val*)
acos(*val*)
asin(*val*)
atan(*val*)
atan2(*val1, val2*)
ceil(*val*)
cos(*val*)
exp(*val*)
floor(*val*)
log(*val*)
max(*val1, val2*)
min(*val1, val2*)
pow(*val1, power*)
random()
round(*val*)
sin(*val*)
sqrt(*val*)
tan(*val*)

*All properties and methods are of the static Math object.

Error 21

prototype
constructor
description[E]
fileName[M]
lineNumber[M]
message
name
number[E]

toString()

Control Statements 21

```
if (condition) {
  statementsIfTrue
}

if (condition) {
  statementsIfTrue
} else {
  statementsIfFalse
}

result = condition ? expr1 : expr2

for ([init expr]; [condition]; [update expr]) {
  statements
}

for (var in object) {
  statements
}

for each ([var] varName in objectRef) {
  statements
}M1.8.1

with (objRef) {
  statements
}

do {
  statements
} while (condition)

while (condition) {
  statements
}

return [value]

switch (expression) {
  case labelN :
    statements
    [break]
  ...
  [default :
    statements]
}

label :
continue [label]
break [label]

try {
  statements to test
}
catch (errorInfo) {
  statements if exception occurs in try block
}
[finally {
  statements to run, exception or not
}]

throw value
```

Number 16

constructor
MAX_VALUE
MIN_VALUE
NaN
NEGATIVE_INFINITY
POSITIVE_INFINITY
prototype

toExponential(n)
toFixed(n)
toLocaleString()
toString([*radix*])
toPrecision(n)
valueOf()

Boolean 16

constructor
prototype

toString()
valueOf()

JavaScript and Browser Objects Quick Reference

Globals 24

Functions
atob()[M]
btoa()[M]
decodeURI("*encodedURI*")
decodeURIComponent("*encComp*")
encodeURI("*URIString*")
encodeURIComponent("*compString*")
escape("*string*" [,1])
eval("*string*")
isFinite(*number*)
isNaN(*expression*)
Number("*string*")
parseFloat("*string*")
parseInt("string" [,*radix*])
toString([*radix*])
unescape("*string*")
unwatch(*prop*)
watch(*prop*, *handler*)

Statements
// /*...*/
const
var

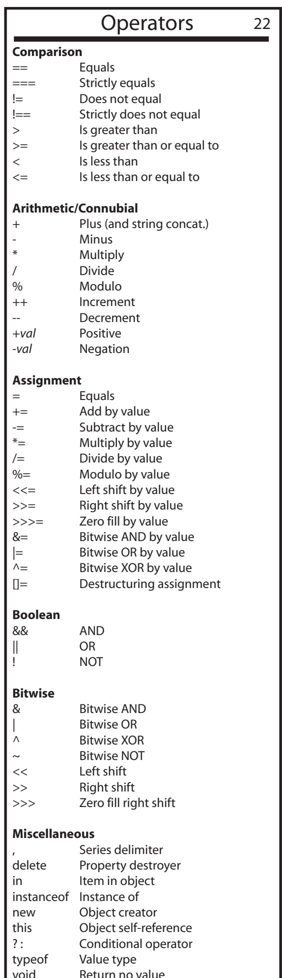
Appendix A
JavaScript Bible, 7th Edition
by Danny Goodman

How to Use This Quick Reference

This guide contains quick reference info for the core JavaScript language and browser object models starting with IE 5.5, Mozilla, and Safari.

Numbers in the upper right corners of object squares are chapter numbers in which the object is covered in detail.

Each term is supported by all baseline browsers unless noted with a superscript symbol indicating browser brand and version:
E—Internet Explorer M—Mozilla S—Safari
O—Opera C—Google Chrome
For example, M1.4 means the term is supported only by Mozilla 1.4 or later; E means the terms is supported only by Internet Explorer.



Operators 22

Comparison

==	Equals
===	Strictly equals
!=	Does not equal
!==	Strictly does not equal
>	Is greater than
>=	Is greater than or equal to
<	Is less than
<=	Is less than or equal to

Arithmetic/Connubial

+	Plus (and string concat.)
-	Minus
*	Multiply
/	Divide
%	Modulo
++	Increment
--	Decrement
+*val*	Positive
-*val*	Negation

Assignment

=	Equals
+=	Add by value
-=	Subtract by value
*=	Multiply by value
/=	Divide by value
%=	Modulo by value
<<=	Left shift by value
>>=	Right shift by value
>>>=	Zero fill by value
&=	Bitwise AND by value
|=	Bitwise OR by value
^=	Bitwise XOR by value
[]=	Destructuring assignment

Boolean

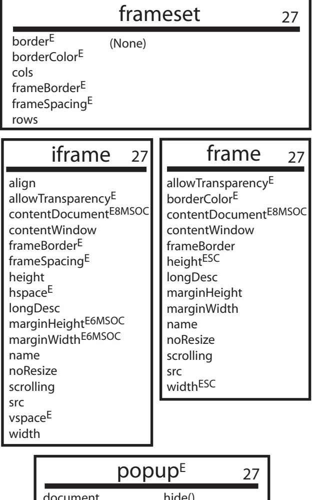
&&	AND
||	OR
!	NOT

Bitwise

&	Bitwise AND
|	Bitwise OR
^	Bitwise XOR
~	Bitwise NOT
<<	Left shift
>>	Right shift
>>>	Zero fill right shift

Miscellaneous

,	Series delimiter
delete	Property destroyer
in	Item in object
instanceof	Instance of
new	Object creator
this	Object self-reference
? :	Conditional operator
typeof	Value type
void	Return no value



frameset 27

borderE
borderColor[E]
cols
frameBorder[E]
frameSpacing[E]
rows

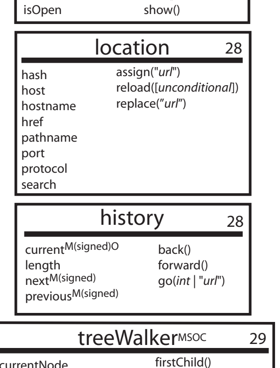
(None)

iframe 27

align
allowTransparency[E]
contentDocument[E8MSOC]
contentWindow
frameBorder[E]
frameSpacing[E]
height
hspace[E]
longDesc
marginHeight[E6MSOC]
marginWidth[E6MSOC]
name
noResize
scrolling
src
vspace[E]
width

frame 27

allowTransparency[E]
borderColor[E]
contentDocument[E8MSOC]
contentWindow
frameBorder
height[ESC]
longDesc
marginHeight
marginWidth
name
noResize
scrolling
src
width[ESC]



popup[E] 27

document	hide()
isOpen	show()

location 28

Properties	Methods
hash	assign("*url*")
host	reload([*unconditional*])
hostname	replace("*url*")
href	
pathname	
port	
protocol	
search	

history 28

Properties	Methods
current[M(signed)O]	back()
length	forward()
next[M(signed)]	go(*int* | "*url*")
previous[M(signed)]	

treeWalker[MSOC] 29

Properties	Methods
currentNode	firstChild()
expandEntityReference	lastChild()
filter	nextNode()
root	nextSibling()
whatToShow	parentNode()
	previousNode()
	previousSibling()

window 27

Properties
appCore[M]
clientInformation[ES1.2C]
clipboardData[E]
closed
Components[][M]
content[M]
controllers[][M]
crypto[M]
defaultStatus[EMSO]
dialogArguments[EMSC]
dialogHeight[E]
dialogLeft[E]
dialogTop[E]
dialogWidth[E]
directories[M]
document
event/Event
external[EM]
frameElementEMS1.2OC
frames[]
fullScreen[M1.4]
history
innerHeight[MSOC]
innerWidth[MSOC]
length
location
locationbar[M]
menubar[M]
name
navigator
netscape[M]
offscreenBuffering[ES1.2C]
opener
outerHeight[MSOC]
outerWidth[MSOC]
pageXOffset[MSOC]
pageYOffset[MSOC]
parent
personalbar[M]
pkcs11[M]
prompter[M]
returnValue[EMS]
screen
screenLeft[ES1.2OC]
screenTop[ES1.2OC]
screenX[MS1.2OC]
screenY[MS1.2OC]
scrollbars[M]
scrollMaxX[M1.4]
scrollMaxY[M1.4]
scrollX[MSOC]
scrollY[MSOC]
self
sidebar[M]
status[EMSO]
statusbar[M]
toolbar[M]
top
window

Methods
addEventListener("*evt*", *func*, *capt*)[MS]
alert("*msg*")
attachEvent("*evt*", *func*)[E]
back()[M]
blur()
clearInterval(ID)
clearTimeout(ID)
close()
confirm("*msg*")
createPopup()[E]
detachEvent("*evt*", *func*)[E]
dispatchEvent()[MS]
dump("*msg*")[M1.4]
execScript("*exprList*"[, *lang*])[E]
find(["*str*"[, *case*[, *up*]]])[MSC]
fireEvent("*evt*"[, *evtObj*])[E]
focus()
forward()[M]
geckoActiveXObject(ID)[M1.4]
getComputedStyle(*node*, "")[MSOC]
getSelection()[MSOC]
home()[MO]
moveBy(Δx, Δy)[EMSO]
moveTo(x, y)[EMSO]
navigate("*url*")[EO]
open("*url*", "*name*"[, *specs*])
openDialog("*url*", "*name*"[, *specs*])[M]
print()
prompt("*msg*", "*reply*")
removeEventListener("*evt*", *func*, *capt*)[MS]
resizeBy(Δx, Δy)[EMS1]
resizeTo(*width*, *height*)[EMS1]
scroll()
scrollBy(Δx, Δy)
scrollByLines(n)[M]
scrollByPages(n)[M]
scrollTo(x, y)
setInterval(*func*, *msecs*[, *args*])
setTimeout(*func*, *msecs*[, *args*])
showHelp("*url*")[E]
showModalDialog("*url*"[, *args*][, *features*])[EMS2.01C]
showModelessDialog("*url*"[, *args*][, *features*])[E]
sizeToContent()[M]
stop()[MSOC]

Event handlers
onabort[M]
onafterprint[E]
onbeforeprint[E]
onbeforeunload[EMSC]
onblur
onclick
onclose
onerror[EM]
onfocus
onhelp[E]
onkeydown
onkeypress
onkeyup
onload
onmousedown
onmousemove
onmouseout
onmouseover
onmouseup
onmove
onreset
onresize
onscroll[EMS1.3OC]
onselect
onsubmit
onunload

JavaScript and Browser Objects Quick Reference

document 29

activeElement
alinkColor
anchors[]
applets[]
baseURI[M]
bgColor
body
charset[ESOC]
characterSet[MSOC]
compatMode
contentType[M]
cookie
defaultCharset[ESC]
defaultView[MSOC]
designMode
doctype
documentElement
documentURI[M1.7SOC]
domain
embeds[]
expando[E]
fgColor
fileCreatedDate[E]
fileModifiedDate[E]
fileSize[E]
forms[]
frames[]
height[MSC]
images[]
implementation[E6MSOC]
inputEncoding[M1.8SC]
lastModified
linkColor
links[]
location
media[E]
mimeType[E]
nameProp[E6]
namespaces[][E]
parentWindow[EO]
plugins[]
protocol[E]
referrer
scripts[][ESOC]
security[E]
selection[EO]
strictErrorChecking[M1.8]
styleSheets[]
title
URL
URLUnencoded
vlinkColor
width[MSC]
xmlEncoding[M1.8SC]
xmlStandalone[M1.8SC]
xmlVersion[M1.8SC]

clear()
close()
createAttribute("*name*")[E6MSOC]
createCDATASection("*data*")[MSOC]
createComment("*text*")[E6MSOC]
createDocumentFragment()[E6MSOC]
createElement("*tagname*")
createElementNS("*uri*","*tagname*)
createEvent("*evtType*")[MSOC]
createEventObject([*evtObj*])[E]
createNSResolver(*nodeResolver*)[MSOC]
createRange()[MSOC]
createStyleSheet(["*url*"[, *index*]])[E]
createTextNode("*text*")
createTreeWalker(*root, what, filterfunc, exp*)[M1.4SOC]
elementFromPoint(x, y)
evaluate("*expr*", *node, resolver, type, result*)[MSOC]
execCommand("*cmd*"[, *UI*][, *param*])[EM1.3S1.3OC]
getElementById("*ID*")
getElementsByName("*name*")
hasFocus()[EMSC]
importNode(*node, deep*)[MSOC]
open(["*mimetype*"][, "*replace*"])
queryCommandEnabled("*commandName*")[EM1.3SOC]
queryCommandIndterm("*commandName*")[EM1.3SC]
queryCommandState("*commandName*")[EM1.3SOC]
queryCommandSupported("*commandName*")[ESOC]
queryCommandText("*commandName*")[E]
queryCommandValue("*commandName*")[EM1.3SOC]
recalc([*all*])[E]
write("*string*")
writeln("*string*")

onselectionchange[E]
onstop[E]

All HTML Element Objects 26

accessKey
all[][EO]
attributes[]
baseURI[MSOC]
behaviorUrns[][E]
canHaveChildren[E]
canHaveHTML[E]
childNodes[]
children[EMS1.2OC]
cite[E6MSOC]
className
clientHeight
clientLeft
clientTop
clientWidth
contentEditable[EMS1.2OC]
currentStyle[E]
dateTime[E6MSOC]
dataFld[E]
dataFormatAs[E]
dataSrc[E]
dir
disabled
document[ES1.2O]
filters[][E]
firstChild
height
hideFocus[E]
id
innerHTML
innerText[ESOC]
isContentEditable[ES1.2OC]
isDisabled[E]
isMultiLine[E]
isTextEdit[E]
lang
language[E]
lastChild
length
localName[E8MSOC]
namespaceURI[E8MSOC]
nextSibling
nodeName
nodeType
nodeValue
offsetHeight
offsetLeft
offsetParent
offsetTop
offsetWidth
outerHTML[ES1.3OC]
outerText[ES1.3OC]
ownerDocument
parentElement[ES1.2OC]
parentNode
parentTextEdit[E]
prefix[E8MSOC]
previousSibling
readyState[E]
recordNumber[E]
runtimeStyle[E]
scopeName[E]
scrollHeight
scrollLeft
scrollTop
scrollWidth
sourceIndex[ESOC]
style
tabIndex
tagName
tagUrn[E]
textContent[M1.7SOC]
title
uniqueID[E]
unselectable[EO]
width

addBehavior("*url*")[E]
addEventListener("*evt*", *func,capt*)[MSOC]
appendChild(*node*)
applyElement(*elem*[, *type*])[E]
attachEvent("*evt*", *func*)[EO]
blur()
clearAttributes()[E]
click()
cloneNode(*deep*)
compareDocumentPosition(*node*)[M1.4SOC]
componentFromPoint(x, y)[E]
contains(*elem*)[ESOC]
createControlRange()[E]
detachEvent("*evt*", *func*)[EO]
dispatchEvent(*evtObj*)[MSOC]
doScroll("*action*")[E]
dragDrop()[E]
fireEvent("*evtType*"[, *evtObj*])[E]
focus()
getAdjacentText("*where*")[E]
getAttribute("*name*"[, *case*])
getAttributeNode("*name*")[E6MSOC]
getAttributeNodeNS("*uri*","*name*")[M]
getAttributeNS("*uri*","*name*")[MSOC]
getBoundingClientRect()
getClientRects()
getElementsByTagName("*tagname*")
getElementsByTagNameNS("*uri*","*name*")[MSOC]
getExpression("*attrName*")[E]
getFeature("*feature*","*version*")[M1.7.2O]
getUserData("*key*")[M1.7.2SC]
hasAttribute("*attrName*")
hasAttributeNS("*uri*","*name*")[MSOC]
hasAttributes()
hasChildNodes()
insertAdjacentElement("*where*", *obj*)[ESOC]
insertAdjacentHTML("*where*","*HTML*")[ESOC]
insertAdjacentText("*where*", "*text*")[ESOC]
insertBefore(*newNode, refNode*)
isDefaultNamespace("*uri*")[M1.7.2SOC]
isEqualNode(*node*)[M1.7.2SOC]
isSameNode(*node*)[M1.7.2SOC]
isSupported("*feature*","*version*")[MSOC]
item(*index*)
lookupNamespaceURI("*prefix*")[M1.7.2SOC]
lookupPrefix("*uri*")[M1.7.2SOC]
mergeAttributes(*srcObj*)[E]
normalize()
releaseCapture()[E]
removeAttribute("*attrName*"[, *case*])
removeAttributeNode(*attrNode*)[E6MSOC]
removeAttributeNS("*uri*","*name*")[MSOC]
removeBehavior(*ID*)[E]
removeChild(node)
removeEventListener("*evt*", *func,capt*)[MSOC]
removeExpression("*propName*")[E]
removeNode(*childrenFlag*)[E]
replaceAdjacentText("*where*","*text*")[E]
replaceChild(*newNode, oldNode*)
replaceNode(*newNode*)[E]
scrollIntoView(*topFlag*)[EMS2.02OC]
setActive()[E]
setAttribute("*name*","*value*"[, *case*])
setAttributeNode(*attrNode*)[E6MSOC]
setAttributeNodeNS("*uri*","*name*")[MSOC]
setAttributeNS("*uri*","*name*","*value*")[MSOC]
setCapture(*containerFlag*)[E]
setExpression("*propName*","*expr*")[E]
setUserData("*key*", *data, handler*)[M1.7.2SC]
swapNode(*nodeRef*)[E]
tags("*tagName*")[ESOC]
toString()
urns("*behaviorURN*")[E]

onactivate[E]
onafterupdate[E]
onbeforecopy[ES1.3]
onbeforecut[ES1.3]
onbeforedeactivate[E]
onbeforeeditfocus[E]
onbeforepaste[ES1.3C]
onbeforeupdate[E]
onblur
oncellchange[E]
onclick
oncontextmenu[EMSC]
oncontrolselect[E]
oncopy[EMS1.3C]
oncut[EMS1.3C]
ondataavailable[E]
ondatasetchanged[E]
ondatasetcomplete[E]
ondblclick
ondeactivate[E]
ondrag[EMS1.3C]
ondragend[EMS1.3C]
ondragenter[EMS1.3C]
ondragleave[EMS1.3C]
ondragover[EMS1.3C]
ondragstart[EMS1.3C]
ondrop[EMS1.3C]
onerrorupdate[E]
onfilterchange[E]
onfocus
onfocusin[E]
onfocusout[E]
onhelp[E]
onkeydown
onkeypress
onkeyup
onlayoutcomplete[E]
onlosecapture[E]
onmousedown
onmouseenter[E]
onmouseleave[E]
onmousemove
onmouseout
onmouseover
onmouseup
onmousewheel
onmove[E]
onmoveend[E]
onmovestart[E]
onpaste[EMS1.3C]
onpropertychange[E]
onreadystatechange[EMS1.2OC]
onresize
onresizeend[E]
onresizestart[E]
onrowenter[E]
onrowexit[E]
onrowsdelete[E]
onrowsinserted[E]
onscroll
onselectstart[ES1.3C]

link 40

charset
disabled
href
hreflang[E6MSOC]
media
rel
rev
sheet[M]
styleSheet[E]
target
type

(None)

onload[E]

html 40

version[E6MSOC]

head 40

profile

meta 40

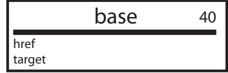
charset[E]
content
httpEquiv
name
url[E]

script 40

defer[E]
event[E]
htmlFor[E]
src
text
type

title 40

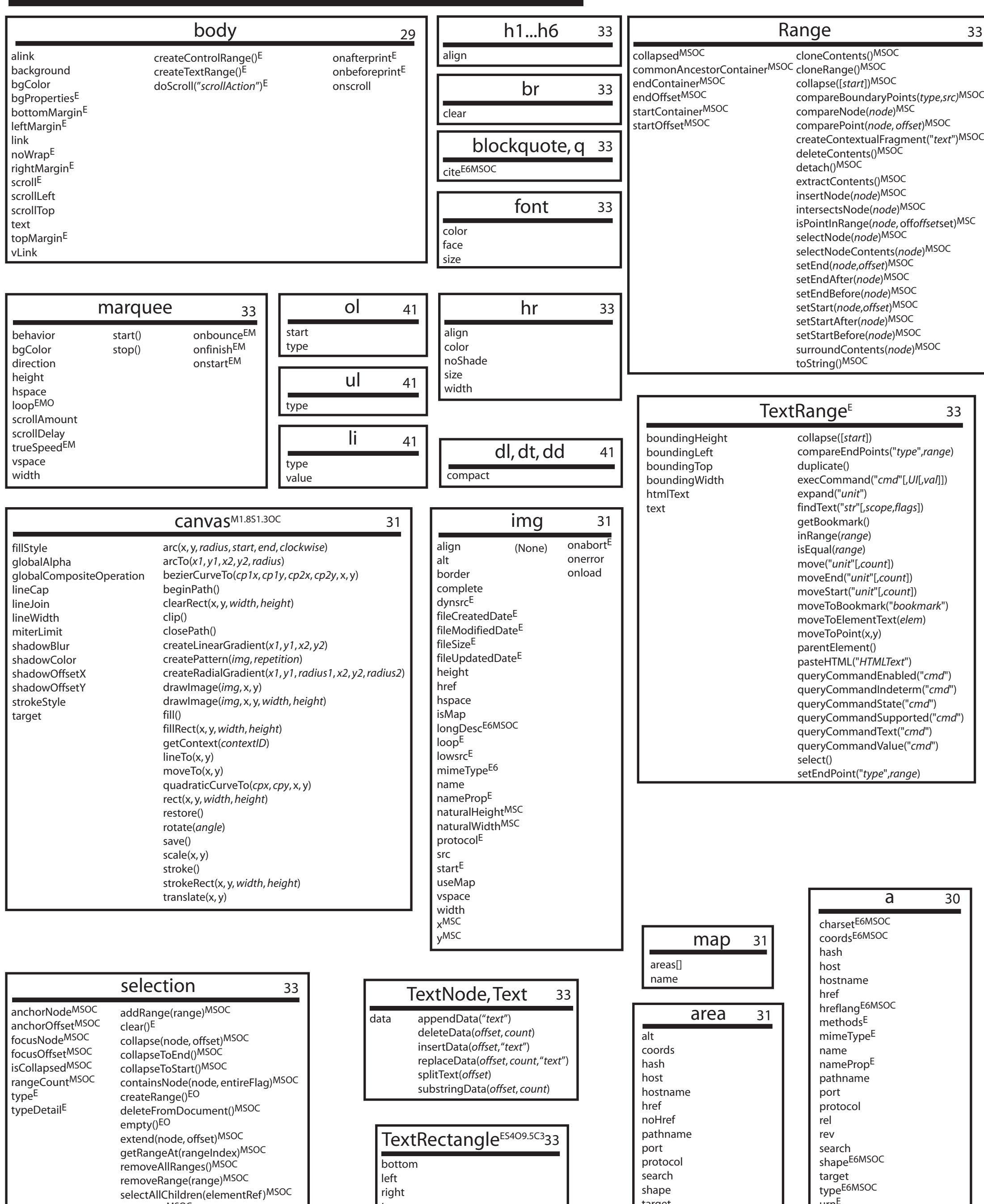
text



base 40

href
target

JavaScript and Browser Objects Quick Reference



body 29
- Properties: alink, background, bgColor, bgProperties[E], bottomMargin[E], leftMargin[E], link, noWrap[E], rightMargin[E], scroll[E], scrollLeft, scrollTop, text, topMargin[E], vLink
- Methods: createControlRange()[E], createTextRange()[E], doScroll("*scrollAction*")[E]
- Events: onafterprint[E], onbeforeprint[E], onscroll

h1...h6 33
- align

br 33
- clear

blockquote, q 33
- cite[E6MSOC]

font 33
- color, face, size

Range 33
- Properties: collapsed[MSOC], commonAncestorContainer[MSOC], endContainer[MSOC], endOffset[MSOC], startContainer[MSOC], startOffset[MSOC]
- Methods: cloneContents()[MSOC], cloneRange()[MSOC], collapse([*start*])[MSOC], compareBoundaryPoints(*type,src*)[MSOC], compareNode(*node*)[MSOC], comparePoint(*node, offset*)[MSOC], createContextualFragment("*text*")[MSOC], deleteContents()[MSOC], detach()[MSOC], extractContents()[MSOC], insertNode(*node*)[MSOC], intersectsNode(*node*)[MSOC], isPointInRange(*node, offoffsetset*)[MSC], selectNode(*node*)[MSOC], selectNodeContents(*node*)[MSOC], setEnd(*node,offset*)[MSOC], setEndAfter(*node*)[MSOC], setEndBefore(*node*)[MSOC], setStart(*node,offset*)[MSOC], setStartAfter(*node*)[MSOC], setStartBefore(*node*)[MSOC], surroundContents(*node*)[MSOC], toString()[MSOC]

marquee 33
- Properties: behavior, bgColor, direction, height, hspace, loop[EMO], scrollAmount, scrollDelay, trueSpeed[EM], vspace, width
- Methods: start(), stop()
- Events: onbounce[EM], onfinish[EM], onstart[EM]

ol 41
- start, type

ul 41
- type

li 41
- type, value

hr 33
- align, color, noShade, size, width

dl, dt, dd 41
- compact

TextRange[E] 33
- Properties: boundingHeight, boundingLeft, boundingTop, boundingWidth, htmlText, text
- Methods: collapse([*start*]), compareEndPoints("*type*",*range*), duplicate(), execCommand("*cmd*"[,*UI*[,*val*]]), expand("*unit*"), findText("*str*"[,*scope,flags*]), getBookmark(), inRange(*range*), isEqual(*range*), move("*unit*"[,*count*]), moveEnd("*unit*"[,*count*]), moveStart("*unit*"[,*count*]), moveToBookmark("*bookmark*"), moveToElementText(*elem*), moveToPoint(x,y), parentElement(), pasteHTML("*HTMLText*"), queryCommandEnabled("*cmd*"), queryCommandIndeterm("*cmd*"), queryCommandState("*cmd*"), queryCommandSupported("*cmd*"), queryCommandText("*cmd*"), queryCommandValue("*cmd*"), select(), setEndPoint("*type*",*range*)

canvas[M1.8S1.3OC] 31
- Properties: fillStyle, globalAlpha, globalCompositeOperation, lineCap, lineJoin, lineWidth, miterLimit, shadowBlur, shadowColor, shadowOffsetX, shadowOffsetY, strokeStyle, target
- Methods: arc(x, y, *radius, start, end, clockwise*), arcTo(*x1, y1, x2, y2, radius*), bezierCurveTo(*cp1x, cp1y, cp2x, cp2y*, x, y), beginPath(), clearRect(x, y, *width, height*), clip(), closePath(), createLinearGradient(*x1, y1, x2, y2*), createPattern(*img, repetition*), createRadialGradient(*x1, y1, radius1, x2, y2, radius2*), drawImage(*img*, x, y), drawImage(*img*, x, y, *width, height*), fill(), fillRect(x, y, *width, height*), getContext(*contextID*), lineTo(x, y), moveTo(x, y), quadraticCurveTo(*cpx, cpy*, x, y), rect(x, y, *width, height*), restore(), rotate(*angle*), save(), scale(x, y), stroke(), strokeRect(x, y, *width, height*), translate(x, y)

img 31
- Properties: align, alt, border, complete, dynsrc[E], fileCreatedDate[E], fileModifiedDate[E], fileSize[E], fileUpdatedDate[E], height, href, hspace, isMap, longDesc[E6MSOC], loop[E], lowsrc[E], mimeType[E6], name, nameProp[E], naturalHeight[MSC], naturalWidth[MSC], protocol[E], src, start[E], useMap, vspace, width, x[MSC], y[MSC]
- Methods: (None)
- Events: onabort[E], onerror, onload

map 31
- areas[], name

a 30
- charset[E6MSOC], coords[E6MSOC], hash, host, hostname, href, hreflang[E6MSOC], methods[E], mimeType[E], name, nameProp[E], pathname, port, protocol, rel, rev, search, shape[E6MSOC], target, type[E6MSOC], urn[E]

selection 33
- Properties: anchorNode[MSOC], anchorOffset[MSOC], focusNode[MSOC], focusOffset[MSOC], isCollapsed[MSOC], rangeCount[MSOC], type[E], typeDetail[E]
- Methods: addRange(range)[MSOC], clear()[E], collapse(node, offset)[MSOC], collapseToEnd()[MSOC], collapseToStart()[MSOC], containsNode(node, entireFlag)[MSOC], createRange()[EO], deleteFromDocument()[MSOC], empty()[EO], extend(node, offset)[MSOC], getRangeAt(rangeIndex)[MSOC], removeAllRanges()[MSOC], removeRange(range)[MSOC], selectAllChildren(elementRef)[MSOC], toString()[MSOC]

TextNode, Text 33
- Properties: data
- Methods: appendData("*text*"), deleteData(*offset, count*), insertData(*offset,"text"*), replaceData(*offset, count,"text"*), splitText(*offset*), substringData(*offset, count*)

area 31
- alt, coords, hash, host, hostname, href, noHref, pathname, port, protocol, search, shape, target

TextRectangle[ES4O9.5C3] 33
- bottom, left, right, top

JavaScript and Browser Objects Quick Reference

form 34

acceptCharset
action
autocomplete[E]
elements[]
encoding[E6MSOC]
enctype[E6MSOC]
length
method
name
target

reset()
submit()

onreset
onsubmit

input 35/36/37

checked[(†)]
complete[E(image)]
defaultChecked[(†)]
defaultValue[(††††)]
form
maxLength[(†††)]
name
readOnly[(††††)]
size[(††††)]
src[(image)]
type
value

blur()[(†††)]
click()[(††)]
focus()[(†††)]
select()[(††††)]

onafterupdate[E(†††)]
onbeforeupdate[E(†††)]
onblur[(†††)]
onchange[(††††)]
onclick[(††)]
onerrorupdate[E(†††)]
onfocus[(†††)]
onmousedown[(button)]
onmouseup[(button)]
onselect[(†††)]

† checkbox, radio
†† button, checkbox, radio
††† text, password, hidden
†††† text, password, hidden, file

textarea 36

cols
form
name
readOnly
rows
type
value
wrap[E]

createTextRange()[E]
select()

onafterupdate[E]
onbeforeupdate[E]
onchange
onerrorupdate[E]

select 37

form
length
multiple
name
options[]
options[i].defaultSelected
options[i].index
options[i].selected
options[i].text
options[i].value
selectedIndex
size
type
value

add(*newOption*[, *index*])[E]
add(*newOption*, *optionRef*)[MS]
item(*index*)[EM5O9]
namedItem("*optionID*")[EM5O9]
options[i].add(*elementRef*[, *index*])[ESOC]
options[i].remove()
remove(*index*)

onchange

option 37

defaultSelected
form
label[E6MSOC]
selected
text
value

caption 41

align
vAlign

fieldset, legend 33/34

align
form

label 33/34

accessKey
form
htmlFor

optgroup[E6MSOC] 37

form
label

screen 42

availHeight
availLeft[MSC]
availTop[MSC]
availWidth
bufferDepth[E]
colorDepth
fontSmoothingEnabled[E]
height
pixelDepth
updateInterval[E]
width

table 41

align
background[E]
bgColor
border
borderColor[E]
borderColorDark[E]
borderColorLight[E]
caption
cellPadding
cells[E]
cellSpacing
cols[E]
datePageSize[E]
frame
height
rows
rules
summary[E6MSOC]
tBodies
tFoot
tHead
width

createCaption()
createTFoot()
createTHead()
deleteCaption()
deleteRow(i)
deleteTFoot()
deleteTHead()
firstPage()[E]
insertRow(i)
lastPage()[E]
moveRow(*srcIndex*, *destIndex*)[E]
nextPage()[E]
previousPage()[E]
refresh()[E]

tbody, tfoot, thead 41

align
bgColor
ch[E6MSOC]
chOff[E6MSOC]
rows
vAlign

deleteRow(i)
insertRow(i)
moveRow(*srcIndex*, *destIndex*)[E]

tr 41

align
bgColor
borderColor[E]
borderColorDark[E]
borderColorLight[E]
cells
ch[E6MSOC]
chOff[E6MSOC]
height[EO]
rowIndex
sectionRowIndex
vAlign

deleteCell(i)
insertCell(i)

td, th 41

abbr[E6MSOC]
align
axis[E6MSOC]
background[E]
bgColor
borderColor[E]
borderColorDark[E]
borderColorLight[E]
cellIndex
ch[E6MSOC]
chOff[E6MSOC]
colSpan
headers[E6MSOC]
height
noWrap
rowSpan
vAlign
width

col, colgroup 41

align
ch[E6MSOC]
chOff[E6MSOC]
span
vAlign
width

navigator 42

appCodeName
appMinorVersion[EO]
appName
appVersion
browserLanguage[EO]
cookieEnabled
cpuClass[E]
language[MSOC]
mimeTypes[MSOC]
onLine
oscpu[M]
platform
plugins[MSOC]
product[MSC]
productSub[MSC]
securityPolicy[M]
systemLanguage[EO]
userAgent
userLanguage[EO]
userProfile[E]
vendor[MSC]
vendorSub[MSC]

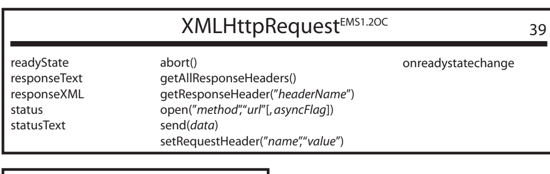
javaEnabled()[MSOC]
preference(*name*[, *val*])[M(signed)]

JavaScript and Browser Objects Quick Reference

event 32

altKey
altLeft[E]
behaviorCookie[E6]
behaviorPart[E6]
bookmarks[E6]
boundElements[E6]
bubbles[MS]
button
cancelable[MSOC]
cancelBubble
charCode[MSC]
clientX
clientY
contentOverflow[E5.5]
ctrlKey
ctrlLeft[E]
currentTarget[MSOC]
dataFld[E6]
dataTransfer[E]
detail[MS2O8C]
eventPhase[MSOC]
fromElement[ESOC]
isChar[M]
isTrusted[M1.7.5]
keyCode[MSC]
layerX[MSC]
layerY[MSC]
metaKey
nextPage[E]
offsetX[ESOC]
offsetY[ESOC]
originalTarget[M]
pageX[MSOC]
pageY[MSOC]
propertyName[E]
qualifier[E6]
reason[E6]
recordset[E6]
relatedTarget[MSOC]
repeat[E]
returnValue[ES1.2OC]
saveType[E]
screenX
screenY
shiftKey
shiftLeft[E]
srcElement[ES1.2OC]
srcFilter[E]
srcUrn[E]
target[MSOC]
timeStamp[MSC]
toElement[ESOC]
type
view[MSOC]
wheelData[E]
x[ESOC]
y[ESOC]

initEvent()[MS]
initKeyEvent()[MS]
initMouseEvent()[MS]
initMutationEvent()[MS]
initUIEvent()[MS]
preventDefault()[MS]
stopPropagation()[MS]



XMLHttpRequest[EMS1.2OC] 39

readyState
responseText
responseXML
status
statusText

abort()
getAllResponseHeaders()
getResponseHeader("*headerName*")
open("*method*","*url*"[, *asyncFlag*])
send(*data*)
setRequestHeader("*name*","*value*")

onreadystatechange

xml[E] 39

src
XMLDocument

applet 44

align
alt[E6MSOC]
altHTML[E]
archive[E6MSOC]
code
codeBase
height
hspace
name
object[E]
vspace
width
(Applet variables)

(Applet methods)

embed 44

align[MSOC]
height
hidden[E]
name
pluginspage[E]
src
units[E]
width
(Object variables)

(Object methods)

onload()
onscroll()

mimeType[MSOC] 42

description
enabledPlugin
type
suffixes

plugin[MSOC] 42

name
filename
description
length

refresh()

object (object) 23

constructor
prototype

hasOwnProperty("*propName*")
isPrototypeOf(objRef)
propertyIsEnumerable("*propName*")
toSource()[M]
toString()
unwatch("*propName*")[M]
valueOf()
watch("*propName*")[M]

object (element) 44

align[ESOC]
alt[E6]
altHTML[E]
archive[E6MSOC]
baseHref[E]
baseURI[MSOC]
border[E6MSOC]
classid[E]
code
codeBase
codeType
contentDocument[MSOC]
data
declare[E6MSOC]
form
height
hspace
name
object[E]
standby[E6MSOC]
type
useMap[E6MSOC]
vspace
width
(Object variables)

(Object methods)

oncellchange
ondataavailable
ondatasetchanged
ondatasetcomplete
onload
onrowenter
onrowexit
onrowsdelete
onrowsinserted
onscroll

Notes

JavaScript and Browser Objects Quick Reference

style, currentStyle, runtimeStyle 38

Text & Fonts
color
font
fontFamily
fontSize
fontSizeAdjust[M]
fontStretch[M]
fontStyle
fontVariant[EMS1.3OC]
fontWeight
letterSpacing
lineBreak[E]
lineHeight
quotes[MO]
rubyAlign[E]
rubyOverhang[E]
rubyPosition[E]
textAlign
textAlignLast[E]
textAutospace[E]
textDecoration
textDecorationBlink[E]
textDecorationLineThrough[E]
textDecorationNone[E]
textDecorationOverline[E]
textDecorationUnderline[E]
textIndent
textJustify[E]
textJustifyTrim[E]
textKashidaSpace[E]
textOverflow[E6S1.3C]
textShadow[MS1.2OC]
textTransform
textUnderlinePosition[E]
unicodeBidi
whiteSpace
wordBreak[ESC]
wordSpacing[E6MSOC]
wordWrap[ES1.3C]
writingMode[E]

Positioning
bottom
height
left
pixelBottom[ESOC]
pixelHeight[ESOC]
pixelLeft[ESOC]
pixelRight[ESOC]
pixelTop[ESOC]
pixelWidth[ESOC]
posBottom[ESOC]
posHeight[ESOC]
posLeft[ESOC]
posRight[ESOC]
posTop[ESOC]
posWidth[ESOC]
position
right
top
width
zIndex

Borders & Edges
border
borderBottom
borderLeft
borderRight
borderTop
borderBottomColor
borderLeftColor
borderRightColor
borderTopColor
borderBottomStyle
borderLeftStyle
borderRightStyle
borderTopStyle
borderBottomWidth
borderLeftWidth
borderRightWidth
borderTopWidth
borderColor
borderStyle
borderWidth
margin
marginBottom
marginLeft
marginRight
marginTop
mozBorderRadius[M]
mozBorderRadiusBottomLeft[M]
mozBorderRadiusBottomRight[M]
mozBorderRadiusTopLeft[M]
mozBorderRadiusTopRight[M]
outline[M1.8.1S1.2OC]
outlineColor[M1.8.1S1.2OC]
outlineOffset[M1.8.1S1.2OC]
outlineStyle[M1.8.1S1.2OC]
outlineWidth[M1.8.1S1.2OC]
padding
paddingBottom
paddingLeft
paddingRight
paddingTop

Tables
borderCollapse[EMS1.3OC]
borderSpacing
captionSide[MSOC]
emptyCells[MS1.3OC]
tableLayout

Lists
listStyle
listStyleImage
listStylePosition
listStyleType

Background
background
backgroundAttachment[EMS1.2OC]
backgroundColor
backgroundImage
gackgroundPosition
backgroundPositionX[ES1.3OC]
backgroundPositionY[ES1.3OC]
backgroundRepeat

Inline Display & Layout
clear
clip
clipBottom[E]
clipLeft[E]
clipRight[E]
clipTop[E]
content[MS1.3OC]
counterIncrement[M1.8SOC]
counterReset[M1.8SO9C]
cssFloat[MSOC]
cursor[EMS1.3OC]
direction
display
filter[E]
layoutGrid[E]
layoutGridChar[E]
layoutGridLine[E]
layoutGridMode[E]
layoutGridType[E]
markerOffset[M]
marks[M]
maxHeight[E7MSOC]
maxWidth[MSOC]
minHeight[MSOC]
minWidth[MSOX]
mozOpacity[M]
opacity[M1.7.2S1.2OC]
overflow
overflowX[EM1.8S1.2OC]
overflowY[EM1.8S1.2OC]
styleFloat[EO]
verticalAlign[EMS1.2OC]
visibility
width
zoom[E]

Page and Printing
orphans[MO]
page[M]
pageBreakAfter[EMS1.3OC]
pageBreakBefore[EMS1.3OC]
pageBreakInside[E8MSOC]
size[MO]
widows[MO]

Miscellaneous
accelerator[E]
behavior[E]
cssText[EMS1.3OC]
imeMode[EM]

Scrollbars
scrollbar3dLightColor[EO]
scrollbarArrowColor[EO]
scrollbarBaseColor[EO]
scrollbarDarkShadowColor[EO]
scrollbarFaceColor[EO]
scrollbarHighlightColor[EO]
scrollbarShadowColor[EO]
scrollbarTrackColor[EO]

styleSheet 38

cssRules[MSO9C]
cssText[E]
disabled
hrefEMSO9C
id[E]
imports[E]
media
ownerNode[MSOC]
ownerRule[MSOC]
owningElement[E]
pages[E]
parentStyleSheet[EMS]
readOnly[E]
rules[ESC]
title
type

addImport("*url*"[, *index*])[E]
addRule("*selector*","*spec*"[, *index*])[ESC]
deleteRule(*index*)[MSOC]
insertRule("*rule*", *index*)[MSOC]
removeRule(*index*)[ESC]

style (element) 38

media
type

cssRule, rule 38

cssText[MSOC]
parentStyleSheet[MSOC]
readOnly[E]
selectorText
style
type[MSOC]

JavaScript Bible, 7th Edition. ©2010 Danny Goodman (dannyg.com). All Rights Reserved.

APPENDIX

B

What's on the CD-ROM

The accompanying Windows–Macintosh CD-ROM contains additional chapters that include many more JavaScript examples, an electronic version of the Quick Reference shown in Appendix A for printing, a complete, searchable version of the entire book, and the Adobe Reader.

System Requirements

To derive the most benefit from the example listings, you should have a Mozilla-based browser (for example, Firefox 3.6+, -, or Camino 2+), Internet Explorer 8+, a WebKit-based browser (for example, Safari 4+, or Google Chrome 5+) or a Presto-based browser (for example, Opera 10+) installed on your computer. Although many scripts run in these and other browsers, several scripts demonstrate features that are available on only a limited range of browsers. To write scripts, you can use a simple text editor, word processor, or dedicated HTML editor.

To use the Adobe Reader (version 7.0), you need the following:

- For Windows XP Pro/Home, or Windows XP Table PC Edition, you should be using a Pentium computer with 128 MB of RAM and 90 MB of hard disk space.
- Macintosh users require a PowerPC G3, G4, or G5 processor, OS X v10.2.8 or later, at least 128 MB of RAM, and 110 MB of disk space.

Disc Contents

When you view the contents of the CD-ROM, you will see files tailored for your operating system. The contents include the following items.

JavaScript listings for text editors

Starting with Part III of the book, almost all example listings are on the CD-ROM in the form of complete HTML files, which you can load into a browser to see the JavaScript item in operation. A directory called Listings contains the example files, with nested folders named for each chapter. The name of each HTML file is keyed to the listing number in the book. For example, the file for Listing 26-1 is named `jsb26-01.html`. Note that no listing files are provided for the tutorial chapters of Part II, because you are encouraged to enter HTML and scripting code manually.

For your convenience, the `_index.html` file in the `Listings` folder provides a front-end table of contents to the HTML files for the book's program listings. Open that file from your browser whenever you want to access the program listing files. If you intend to access that index page frequently, you can bookmark it in your browser(s). Using the index file to access the listing files can be very important in some cases, because several individual files must be opened within their associated framesets to work properly. Accessing the files through the `_index.html` file ensures that you open the frameset. The `_index.html` file also shows browser compatibility ratings for all the listings. This saves you time from opening listings that are not intended to run on your browser. To examine and modify the HTML source files, open them from your favorite text editor program (for Windows editors, be sure to specify the `.html` file extension in the Open File dialog box).

You can open all example files directly from the CD-ROM, but if you copy them to your hard drive, access is faster and you will be able to experiment with modifying the files more readily. Copy the folder named `Listings` from the CD-ROM to any location on your hard drive.

Printable version of the JavaScript and Browser Object Quick Reference from Appendix A

If you like the quick reference in Appendix A, you can print it out with the help of the Adobe Reader, included with the CD-ROM.

Adobe Reader

The Adobe Reader is a helpful program that enables you to view the Quick Reference from Appendix A and the searchable version of this book, both of which are in PDF format on the CD-ROM. To install and run Adobe Reader, follow these steps:

For Windows

1. Navigate to the Adobe_Reader folder on the CD-ROM.
2. In the Adobe_Reader folder, double-click the lone executable file and follow the instructions presented on-screen for installing Adobe Acrobat Reader.

For Macintosh

1. Open the Adobe_Reader folder on the CD-ROM.
2. In the Adobe_Reader folder, double-click the Adobe Reader disk image icon; this will mount the disk image on your computer. Then open the mounted image and copy the Adobe Reader folder to the Applications directory of your computer.

PDF version of book with topical references

In many places throughout the reference chapters of Parts III and IV, you see notations directing you to the CD-ROM for a particular topic being discussed. All these topics are located in the chapters as they appear in complete Adobe Acrobat form on the CD-ROM. A single PDF file is located on the CD-ROM, and it serves as an electronic version of the entire book, complete with full topics that are listed as CD-ROM references in the printed book. For the fastest access to these topics, copy the entire PDF file for the book to your hard disk.

Like any PDF document, the PDF version of the book is searchable. Recent versions of Adobe Reader should automatically load the index file (with the `.pdx` extension) to supply indexed search capabilities (which is much faster than Acrobat's Find command).

To begin an actual search, click the Search icon (binoculars in front of a sheet of paper). Enter the text for which you're searching. To access the index and search facilities in future sessions, the CD-ROM must be in your CD-ROM drive — unless, of course, you've copied both the `.pdx` and `.pdf` files to your hard drive.

Troubleshooting

If you have difficulty installing or using the CD-ROM programs, try the following solutions:

- **Turn off any anti-virus software that you may have running.** Installers sometimes mimic virus activity and can make your computer incorrectly believe that a virus is infecting it. (Be sure to turn the anti-virus software back on later.)
- **Close all running programs.** The more programs you're running, the less memory is available to other programs. Installers also typically update files and programs; if you keep other programs running, installation may not work properly.
- **Reference the ReadMe file.** Refer to the ReadMe file located at the root of the CD-ROM for the latest product information at the time of publication.

Customer Care

If you have trouble with the CD-ROM, please call the Customer Support phone number at (800) 762-2974. Outside the United States, call 1(317) 572-3994. You can also contact Wiley Product Technical Support at `http://support.wiley.com`. John Wiley & Sons will provide technical support only for installation and other general quality control items. For technical support on the applications themselves, consult the program's vendor or author.

To place additional orders or to request information about other Wiley products, please call (877) 762-2974.

Part IX

Appendixes (continued)

IN THIS PART

Appendix C
JavaScript Reserved Words

Appendix D
Answers to Tutorial Exercises

Appendix E
JavaScript and DOM Internet Resources

APPENDIX

C

JavaScript Reserved Words

Every programming language has a built-in vocabulary of keywords that you cannot use for the names of variables and the like. Because a JavaScript function is an object that uses the function name as an identifier for the object, you cannot employ reserved words for function names either. It's worth nothing that many of the keywords in the list are not technically a part of the JavaScript language just yet, but they are reserved for potential future use. Remember that JavaScript keywords are case-sensitive. Although you may get away with using these words in other cases, it may lead to unnecessary confusion for someone reading your scripts.

abstract	boolean	break	byte
case	catch	char	class
const	continue	debugger	default
delete	do	double	else
enum	export	extends	false
final	finally	float	for
function	goto	if	implements
import	in	instanceof	int
interface	long	native	new
null	package	private	protected
public	return	short	static
super	switch	synchronized	this
throw	throws	transient	true
try	typeof	var	void
volatile	while	with	

APPENDIX

Answers to Tutorial Exercises

This appendix provides answers to the tutorial exercises that appear in Part II of this book (Chapters 5 through 14).

Chapter 5 Answers

1. The paragraph in Listing 5-1 was originally:

```
<p>It is currently <span id="output">now</span>.</p>
```

The JavaScript code modifies the contents of the `span` element because, and only because, it has the ID of `output`. Therefore, we can get the script to replace the contents of the entire paragraph simply by shifting the ID to the `p` tag:

```
<p id="output">It is currently <span>now</span>.</p>
```

The `span` element is no longer useful:

```
<p id="output">It is currently now.</p>
```

2. The JavaScript does not need to change in order to reflect this change to the HTML. The steps it performs are as follows:

   a. If there is a first child element,

   b. remove the first child element,

   c. then return to step a.

Notice how versatile and re-purposeable the deletion loop is. It could easily be made into a generic function to delete the contents of any DOM element.

3. The style sheet does not have to be changed to assign the paragraph the same font attributes that the `span` had before, because the styling refers to the element's ID and not its tag name:

```
#output
{
   font-style: italic;
   color: #C33;
}
```

Here again, the CSS rule is not overly specific (it did not point to `span#output` but merely `#output`) and therefore is easily re-purposeable even as the HTML undergoes modest change.

Chapter 6 Answers

1. The catalog page (a) and temperature calculator (d) are good client-side JavaScript applications. Even though the catalog relies on server storage of the image files, you can create a more engaging and responsive user interface of buttons and swappable images. The temperature calculator is a natural for JavaScript, because all processing is done instantaneously on the client, rather than having to access the server for each conversion.

   The Web site visit counter (b) that accumulates the number of different visitors to a Web site is a server-side application, because the count must be updated and maintained on the server. At best, a client-side counter could keep track of the number of visits the user has made to a site and report to the user how many times he or she has been to the site. The storage requires scripting the cookie. A chat room application (c) done properly requires server facilities to open up communication channels among all users connected simultaneously. Client-side scripting by itself cannot create a live chat environment.

2. **a.** Valid, because it is one contiguous word. InterCap spelling is fine.

   b. Valid, because an underscore character is acceptable between words.

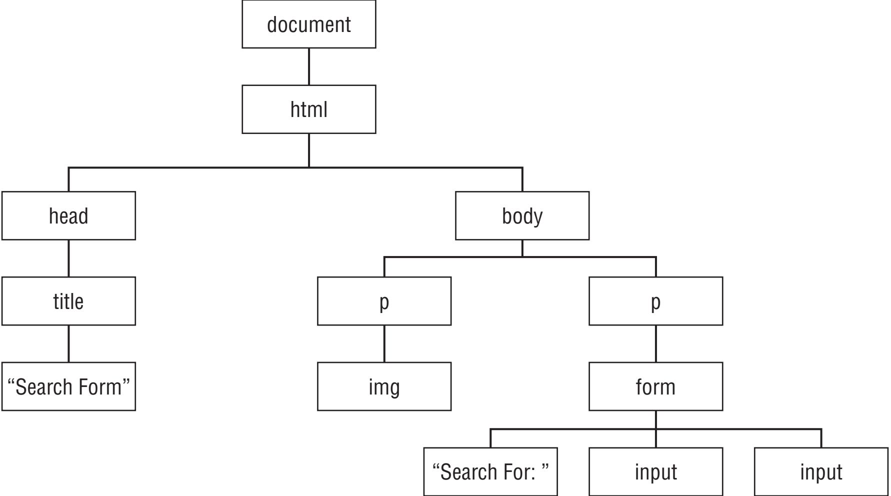
   c. Not valid, because an identifier cannot begin with a numeral.

   d. Not valid, because no spaces are allowed.

   e. Not valid, because apostrophes are not allowed.

   f. Valid, because the colon, hyphen, and period are acceptable characters in HTML IDs. Take caution, however, as JavaScript will choke on such IDs if used outside of quotation marks.

3. The diagram is as follows.



The paragraph element reference is:

```
document.getElementById("logoPar")
```

4. In common:
   - Both are types of nodes, derived from the basic DOM node.
   - Both may be children of parent nodes that act as containers.

   Different:
   - An element node is created by a tag, while a text node has no tag associated with it.
   - A text node cannot be a parent to any other node, but an element node can be either a parent (branch node) or end node (leaf node).

5. HTML:

```
<input type="button" name="Hi" id="Hi" value="Howdy" >
```

JavaScript:

```
var oButton = document.getElementById("Hi");
   if (oButton)
   {
      oButton.onclick=helloBack;
   }
```

```
function helloBack(evt)
{
   alert('Hello to you, too!')
}
```

Chapter 7 Answers

1.
```
<script type="text/javascript">

<!--
document.write("Hello, world.");
// -->
</script>
```

2.
```
<html>

<body>
<script type="text/javascript">
<!--
   document.write("Hello, world.");
// -->
</script>
</body>
</html>
```

3.
```
<html>

<body>
<script type="text/javascript">
<!--
   // write a welcome message to the world
   document.write("Hello, world.");
// -->
</script>
</body>
</html>
```

4. My answer is written so that both event handlers call separate functions. You can also have each event handler invoke the `alert()` method inline. If you prefer to follow the XHTML format, include a space and forward slash character before the right angle bracket of the `input` element's tag.

```
<html>
<head>
<title>An onload script</title>
<script type="text/javascript">
<!--
function done() {
   alert("The page has finished loading.");
```

```
}
function alertUser() {
   alert("Ouch!");
}
// -->
</script>
</head>
<body onload="done()">
Here is some body text.
<form>
    <input type="button" name="oneButton" value="Press Me!"
                     onclick="alertUser()">
</form>
</body>
</html>
```

5. **a.** The page displays two text fields.
   b. The user enters text into the first field and either clicks or tabs out of the field to trigger the `onchange` event handler.
   c. The function writes an all-uppercase version of one field in the other.

Chapter 8 Answers

1. **a.** Valid.
   b. Not valid. The variable needs to be a single word, such as `howMany` or `how_many`.
   c. Valid. The trailing semicolon is missing, but because it is optional for a one-line statement, browsers accept the statement as written.
   d. Not valid. The variable name cannot begin with a numeral. If the variable needs a number to help distinguish it from other similar variables, then put the numeral at the end: `address1`.
2. **a.** 4
   b. 40
   c. “4020”
   d. “Robert”
3. The functions are `parseInt()` and `parseFloat()`. Strings to be converted are passed as parameters to the functions: `parseInt(document.getElementById("entry").value)`.
4. Both text field values are strings that must be converted to numbers before they can be arithmetically added together. You can use the `parseFloat()` functions either on the variable assignment expressions. For example, `var value1 = parseFloat(document.getElementById("inputA").value)` or in the addition expression `(document.getElementById("output").value = parseFloat(value1) + parseFloat(value2))`.
5. Concatenate means to join two strings into one string.

Chapter 9 Answers

1. The following answer shows the HTML markup portion in XHTML, where elements not acting as containers (notably the `input` elements) include a space and forward slash to simulate XHTML's required close tag.

```
<html>
<head>
<script type="text/javascript">
var USStates = new Array(51);
USStates[0] = "Alabama";
USStates[1] = "Alaska";
USStates[2] = "Arizona";
USStates[3] = "Arkansas";
USStates[4] = "California";
USStates[5] = "Colorado";
USStates[6] = "Connecticut";
USStates[7] = "Delaware";
USStates[8] = "District of Columbia";
USStates[9] = "Florida";
USStates[10] = "Georgia";
USStates[11] = "Hawaii";
USStates[12] = "Idaho";
USStates[13] = "Illinois";
USStates[14] = "Indiana";
USStates[15] = "Iowa";
USStates[16] = "Kansas";
USStates[17] = "Kentucky";
USStates[18] = "Louisiana";
USStates[19] = "Maine";
USStates[20] = "Maryland";
USStates[21] = "Massachusetts";
USStates[22] = "Michigan";
USStates[23] = "Minnesota";
USStates[24] = "Mississippi";
USStates[25] = "Missouri";
USStates[26] = "Montana";
USStates[27] = "Nebraska";
USStates[28] = "Nevada";
USStates[29] = "New Hampshire";
USStates[30] = "New Jersey";
USStates[31] = "New Mexico";
USStates[32] = "New York";
USStates[33] = "North Carolina";
USStates[34] = "North Dakota";
USStates[35] = "Ohio";
USStates[36] = "Oklahoma";
USStates[37] = "Oregon";
USStates[38] = "Pennsylvania";
USStates[39] = "Rhode Island";
USStates[40] = "South Carolina";
USStates[41] = "South Dakota";
```

```
USStates[42] = "Tennessee";
USStates[43] = "Texas";
USStates[44] = "Utah";
USStates[45] = "Vermont";
USStates[46] = "Virginia";
USStates[47] = "Washington";
USStates[48] = "West Virginia";
USStates[49] = "Wisconsin";
USStates[50] = "Wyoming";

var stateEntered = new Array(51);
stateEntered[0] = 1819;
stateEntered[1] = 1959;
stateEntered[2] = 1912;
stateEntered[3] = 1836;
stateEntered[4] = 1850;
stateEntered[5] = 1876;
stateEntered[6] = 1788;
stateEntered[7] = 1787;
stateEntered[8] = 0000;
stateEntered[9] = 1845;
stateEntered[10] = 1788;
stateEntered[11] = 1959;
stateEntered[12] = 1890;
stateEntered[13] = 1818;
stateEntered[14] = 1816;
stateEntered[15] = 1846;
stateEntered[16] = 1861;
stateEntered[17] = 1792;
stateEntered[18] = 1812;
stateEntered[19] = 1820;
stateEntered[20] = 1788;
stateEntered[21] = 1788;
stateEntered[22] = 1837;
stateEntered[23] = 1858;
stateEntered[24] = 1817;
stateEntered[25] = 1821;
stateEntered[26] = 1889;
stateEntered[27] = 1867;
stateEntered[28] = 1864;
stateEntered[29] = 1788;
stateEntered[30] = 1787;
stateEntered[31] = 1912;
stateEntered[32] = 1788;
stateEntered[33] = 1789;
stateEntered[34] = 1889;
stateEntered[35] = 1803;
stateEntered[36] = 1907;
stateEntered[37] = 1859;
stateEntered[38] = 1787;
stateEntered[39] = 1790;
stateEntered[40] = 1788;
stateEntered[41] = 1889;
```

```
stateEntered[42] = 1796;
stateEntered[43] = 1845;
stateEntered[44] = 1896;
stateEntered[45] = 1791;
stateEntered[46] = 1788;
stateEntered[47] = 1889;
stateEntered[48] = 1863;
stateEntered[49] = 1848;
stateEntered[50] = 1890;

function getStateDate() {
   var selectedState = document.getElementById("entry").value;
   for ( var i = 0; i < USStates.length; i++) {
      if (USStates[i] == selectedState) {
         break;
      }
   }
   alert("That state entered the Union in " + stateEntered[i] + ".");
}
</script>
</head>
<body>
<form name="entryForm">
   Enter the name of a state:
   <input type="text" name="entry" />
   <input type="button" value="Look Up Entry Date" onclick="getStateDate()" />
</form>
</body>
</html>
```

2. Several problems plague this function definition. Parentheses are missing from the first `if` construction's condition statement. Curly braces are missing from the second nested `if...else` construction. A mismatch of curly braces also exists for the entire function. The following is the correct form (changes and additions in boldface):

```
function format(ohmage) {
   var result;
   if (ohmage >= 10e6) {
      ohmage = ohmage / 10e6;
      result = ohmage + " Mohms";
   } else {
      if (ohmage >= 10e3) {
         ohmage = ohmage / 10e3;
         result = ohmage + " Kohms";
      } else {
         result = ohmage + " ohms";
      }
   }
   alert(result);
}
```

3. Here is one possibility:

```
for (var i = 1; i < tomatoes.length; i++) {
   if (tomatoes[i].looks == "mighty tasty") {
      break;
   }
}
var myTomato = tomatoes[i]
```

4. The new version defines a different local variable name for the dog.

```
<html>
<head>
<script type="text/javascript">
var aBoy = "Charlie Brown";    // global
var hisDog = "Snoopy";        // global
function demo() {
  var WallacesDog = "Gromit"; // local version of hisDog
  var output = WallacesDog + " does not belong to " + aBoy
             + ".<br>";
  document.write(output);
}
</script>
<body>
<script type="text/javascript">
  demo();       // runs as document loads
  document.write(hisDog + " belongs to " + aBoy + ".");
</script>
</body>
</html>
```

5. The application uses three parallel arrays and is structured very much like the solution to question 1. Learn to reuse code whenever you can.

```
<html>
<head>
<script type="text/javascript">
var planets = new Array(4);
planets[0] = "Mercury";
planets[1] = "Venus";
planets[2] = "Earth";
planets[3] = "Mars";

var distance = new Array(4);
distance[0] = "36 million miles";
distance[1] = "67 million miles";
distance[2] = "93 million miles";
distance[3] = "141 million miles";

var diameter = new Array(4);
diameter[0] = "3100 miles";
```

```
diameter[1] = "7700 miles";
diameter[2] = "7920 miles";
diameter[3] = "4200 miles";

function getPlanetData() {
   var selectedPlanet = document.getElementById("entry").value;
   for ( var i = 0; i < planets.length; i++) {
      if (planets[i] == selectedPlanet) {
         break;
      }
   }
   var msg = planets[i] + " is " + distance[i];
   msg += " from the Sun and ";
   msg += diameter[i] + " in diameter.";
   document.getElementById("output").value = msg;
}
</script>
</head>
<body>
<form name="entryForm">
   Enter the name of a planet:
   <input type="text" name="entry" id="entry" />
   <input type="button" value="Look Up a Planet" onclick="getPlanetData()" />
   <br />
   <input type="text" size="70" name="output" id="output" />
</form>
</body>
</html>
```

Chapter 10 Answers

1. **a.** Close, but no cigar. Array references are always plural: `window.document.forms[0]`

   b. Not valid: `self` refers to a window and `entryForm` must refer to a form. Where's the `document`? It should be `self.document.entryForm.entryField.value`.

   c. Valid. This reference points to the `name` property of the third form in the document.

   d. Not valid. The uppercase "D" in the method name is incorrect.

   e. Valid, assuming that `newWindow` is a variable holding a reference to a subwindow.

2. `window.alert("Welcome to my Web page.");`

3. `document.write("<h1>Welcome to my Web page.</h1>");`

4. A script in the `body` portion invokes a function that returns the text entered in a `prompt()` dialog box.

```
<html>
<head>
<script type="text/javascript">
function askName() {
   var name = prompt("What is your name, please?","");
```

```
   return name;
}
</script>
</head>
<body>
<script type="text/javascript">
  document.write("Welcome to my web page, " + askName() + ".");
</script>
</body>
</html>
```

5. The URL can be derived from the `href` property of the `location` object.

```
<html>
<head>
<script type="text/javascript">
function showLocation() {
   alert("This page is at: " + location.href);
}
</script>
</head>
<body onload="showLocation()">
Blah, blah, blah.
</body>
</html>
```

Chapter 11 Answers

1. For Listing 11-2, pass the text input element object because that's the only object involved in the entire transaction.

```
<html>
<head>
<title>Text Object value Property</title>
<script type="text/javascript">
function upperMe(field) {
   field.value = field.value.toUpperCase();
}
</script>
</head>
<body>
<form onsubmit="return false">
   <input type="text" name="convertor" value="sample" onchange="upperMe(this)">
</form>
</body>
</html>
```

For Listing 11-3, the button invokes a function that communicates with a different element in the form. Pass the form object.

```
<html>
<head>
<title>Checkbox Inspector</title>
<script type="text/javascript">
function inspectBox(form) {
   if (form.checkThis.checked) {
      alert("The box is checked.");
   } else {
      alert("The box is not checked at the moment.");
   }
}
</script>
</head>
<body>
<form>
   <input type="checkbox" name="checkThis">Check here<br>
   <input type="button" value="Inspect Box" onclick="inspectBox(this.form)">
</form>
</body>
</html>
```

For Listing 11-4, again the button invokes a function that looks at other elements in the form. Pass the form object.

```
<html>
<head>
<title>Extracting Highlighted Radio Button</title>
<script type="text/javascript">
function fullName(form) {
   for (var i = 0; i < form.stooges.length; i++) {
      if (form.stooges[i].checked) {
         break;
      }
   }
   alert("You chose " + form.stooges[i].value + ".");
}
</script>
</head>

<body>
<form>
   <p>Select your favorite Stooge:
   <input type="radio" name="stooges" value="Moe Howard" checked>Moe
   <input type="radio" name="stooges" value="Larry Fine">Larry
   <input type="radio" name="stooges" value="Curly Howard">Curly<br>
   <input type="button" name="Viewer" value="View Full Name..."
             onclick="fullName(this.form)"></p>
</form>
</body>
</html>
```

For Listing 11-5, all action is triggered by and confined to the `select` object. Pass only that object to the function.

```
<html>
<head>
<title>Select Navigation</title>
<script type="text/javascript">
  function goThere(list) {
   location = list.options[list.selectedIndex].value;
}
</script>
</head>

<body>
<form>
Choose a place to go:
<select name="urlList" onchange="goThere(this)">
   <option selected value="index.html">Home Page
   <option value="store.html">Shop Our Store
   <option value="policies">Shipping Policies
   <option value="http://www.google.com">Search the Web
</select>
</form>
</body>
</html>
```

2. Here are the most likely ways to reference the text box object:

```
document.getElementById("email")
document.forms[0].elements[0]
document.forms["subscription"].elements[0]
document.subscription.elements[0]
document.forms[0].elements["email"]
document.forms["subscription"].elements["email"]
document.subscription.elements["email"]
document.forms[0].email
document.forms["subscription"].email
document.subscription.email
```

The reference `document.all.email` (or any reference starting with `document.all`) works only in Internet Explorer and other browsers that emulate IE, but not in Mozilla or Safari. Other valid references may include the W3C DOM `getElementsByTagName()` method. Since the question indicates that there is only one form on the page, the text box is the first `input` element in the page, indicating that `document.body.getElementsByTagName("input")[0]` would be valid for this page.

3. The `this` keyword refers to the text input object, so that `this.value` refers to the `value` property of that object.

```
<html>
<head>
   <script>
      function showText(txt) {
```

```
         alert(txt);
      }
   </script>
</head>
<body>
   <p>Hello World!</p>
   <input type="button" value="Click Me"
          onclick="showText(this.value)" />
</body>
</html>
```

4.
```
document.accessories.acc1.value = "Leather Carrying Case";

document.forms[1].acc1.value = "Leather Carrying Case";
```

5. The `select` object invokes a function that does the job.

```
<html>
<head>
<title>Color Changer</title>
<script type="text/javascript">
function setColor(list) {
   var newColor = list.options[list.selectedIndex].value;
   document.bgColor = newColor;
}
</script>
</head>

<body>
<form>
   Select a background color:
   <select onchange="setColor(this)">
   <option value="red">Stop
   <option value="yellow">Caution
   <option value="green">Go
   </select>
</form>
</body>
</html>
```

Chapter 12 Answers

1. Use `string.indexOf()` to see if the field contains the "@" symbol.

```
<html>
<head>
<title>E-mail @ Validator</title>
<script type="text/javascript">
function checkAddress(form) {
   if (form.email.value.indexOf("@") == -1) {
      alert("Check the e-mail address for accuracy.");
```

```
      return false;
   }
   return true;
}
</script>
</head>

<body>
<form onsubmit="return checkAddress(this)">
   Enter your e-mail address:
   <input type="text" name="email" size="30"><br>
   <input type="submit">
</form>
</body>
</html>
```

2. Remember that the substring goes up to, but does not include, the index of the second parameter. Spaces count as characters.

```
myString.substring(0,3)    // result = "Int"
myString.substring(11,17)  // result = "plorer"
myString.substring(5,12)   // result = "net Exp"
```

3. The `indexOf()` function returns a `-1` if the character is not found. The `substring()` function will extract a substring starting at the indexed position +1. If there are no spaces in `stringA`, `firstSpace` will be = `-1`. Adding a `1` to that yields `0`. Strings start at position `0`, so `excerpt` will return the entire `stringA`.

4. The missing `for` loop is in boldface. You could also use the increment operator on the `count` variable (`++count`) to add 1 to it for each letter "e."

```
function countE(form) {
   var count = 0;
   var inputString = form.mainstring.value.toLowerCase();
   for (var i = 0; i < inputString.length; i++) {
      if (inputString.charAt(i) == "e") {
         count += 1;
      }
   }
   var msg = "The string has " + count;
   msg += " instances of the letter e.";
   alert(msg);
}
```

5. The formula for the random throw of one die is in the chapter.

```
<html>
<head>
<title>Roll the Dice</title>
<script type="text/javascript">
function roll(form) {
   form.die1.value = Math.floor(Math.random() * 6) + 1
```

```
   form.die2.value = Math.floor(Math.random() * 6) + 1
}
</script>
</head>

<body>
<form>
<input type="text" name="die1" size="2">
<input type="text" name="die2" size="2"><br>
<input type="button" value="Roll the Dice"
       onclick="roll(this.form)">
</form>
</body>
</html>
```

6. If you used the `Math.round()` method in your calculations, that is fine given your current exposure to the `Math` object. Another method, `Math.ceil()`, may be more valuable because it rounds up any fractional value.

```
<html>
<head>
<title>Waiting for Santa</title>
<script type="text/javascript">
function daysToXMAS() {
   var oneDay = 1000 * 60 * 60 * 24;
   var today = new Date();
   var XMAS = new Date("December 25, 2001");
   var diff = XMAS.getTime() - today.getTime();
   return Math.ceil(diff/oneDay);
}
</script>
</head>

<body>
<script type="text/javascript">
   document.write(daysToXMAS() + " days until Christmas.");
</script>
</body>
</html>
```

Chapter 13 Answers

1. `onload="parent.currCourse = 'history101'"`

2.

<Framesets>
Top Parent
<Frame> mechanics
<Frame> description
<Frame> navigation

3. All three frames are siblings, so references include the parent.

```
parent.mechanics.location.href = "french201M.html";
parent.description.location.href = "french201D.html";
```

4. A script in one of the documents is attempting to reference the `selector` object in one of the frames but the document has not fully loaded, causing the object to not yet be in the browser's object model. Rearrange the script so that it fires in response to the `onload` event handler of the framesetting document.

5. From the subwindow, the `opener` property refers back to the frame containing the `window.open()` method. To extend the reference to the frame's parent, the reference includes both pieces: `opener.parent.location`.

Chapter 14 Answers

1. As the document loads, the `<img>` tag creates a document image object. A memory image object is created with the `new Image()` constructor. Both objects have the same properties, and assigning a URL to the `src` property of a memory object loads the image into the browser's image cache.

2. `var janeImg = new Image(100,120);`

   `janeImg.src = "jane.jpg";`

3. `document.images["people"].src = janeImg.src;`

4. Surround `<img>` tags with link (`<a>`) tags, and use the link's `onclick`, `onmouseover`, and `onmouseout` event handlers. Set the image's `border` attribute to zero if you don't want the link highlight to appear around the image.

5. The following works in all W3C DOM-compatible browsers. The order of the first two statements may be swapped without affecting the script.

```
var newElem = document.createElement("a");
var newText = document.createTextNode("Next Page");
newElem.href = "page4.html";
newElem.appendChild(newText);
document.getElementById("forwardLink").appendChild(newElem);
```

APPENDIX

JavaScript and DOM Internet Resources

As an online technology, JavaScript has plenty of support online for scripters. Items recommended here were taken as a snapshot of Internet offerings in early 2010. But beware! Sites tend to change. URLs can change too. Be prepared to hunt around for these items if the information provided here is updated or moved around by the time you read this.

Support and Updates for This Book

The most up-to-date list of errata and other notes of interest pertaining to this edition of *JavaScript Bible* can be found at the book's web site, located at:

```
http://www.wiley.com/WileyCDA/WileyTitle/productCd-
  0470526912.html
```

FAQs

One situation that arises with a popular and accessible technology, such as JavaScript and DHTML authoring, is that the same questions get asked over and over, as newcomers arrive on the scene daily. Rather than invoke the ire of newsgroup users, look through existing FAQ files to see if your concern has already been raised and answered. Here are some of the best JavaScript FAQ sites:

```
http://javascript.faqts.com
http://javascripter.net/faq/index.htm
```

For less-frequently asked questions — but previously asked and answered in a public form — use the Google Groups search, described later in this appendix under Newsgroups.

Online Documentation

Locations of web sites that dispense official documentation for one browser or another are extremely fluid. Therefore, the following information contains links only to top-level areas of appropriate web sites, along with tips on what to look for after you are at the site.

Microsoft has condensed its developer documentation into a massive site called MSDN (Microsoft Developer Network). The place to begin is:

```
http://msdn.microsoft.com/library/
```

This page is the portal to many technologies, but the one most applicable to JavaScript and client-side scripting is one labeled "Library." Within the MSDN Library, you can then click "Web Development" to access information related to JavaScript and other Web development technologies. Inside the Web Development area of the library you'll find a section named "Scripting." Here you'll find plenty of documentation and technical articles for Microsoft scripting technologies, including JScript (Microsoft's flavor of JavaScript).

For Mozilla-based browser technologies, start at:

```
http://www.mozilla.org/docs/web-developer/
```

Finally, you can read the industry standards for HTML, CSS, and ECMAScript technologies online. Be aware that these documents are primarily intended for developers of tools that we use — browsers, WYSIWYG editors, and so forth — to direct them on how their products should respond to tags, style sheets, scripts, and so on. Reading these documents has frequently been cited as a cure for insomnia. That said, however, you can still glean a lot of useful information about scripting, HTML code, and CSS code from them.

```
http://www.ecma-international.org/publications/standards/Ecma-262.htm
http://www.w3.org/TR/html401/
http://www.w3.org/TR/html5/
http://www.w3.org/TR/xhtml1/
http://www.w3.org/TR/xhtml2/
http://www.w3.org/standards/techs/css
http://www.w3.org/DOM/
http://www.w3.org/TR/xml/
http://www.w3.org/TR/
```

Please note that just because a particular item is described in an industry standard doesn't mean that it is implemented in any or all browsers. In the real world, we must develop for the way the technologies are actually implemented in browsers.

World Wide Web

The number of web sites devoted to JavaScript tips and tricks is mind-boggling. Many sites come and go in the middle of the night, leaving no trace of their former existence. If you are looking for more example code for applications not covered in this book, perhaps the best place to begin your journey is through the traditional search engines. Narrowing your search through careful keyword choice is vital. In addition to the Mozilla and (heavily Windows-oriented) Microsoft developer web sites (plus numerous online articles of mine listed at `http://www.dannyg.com/pubs/index.html`), a few other venerable sites are:

```
http://www.javascript.com/
http://www.w3schools.com/js/
http://www.webreference.com/js/
http://en.wikipedia.org/wiki/Javascript
https://wiki.mozilla.org/JavaScript
http://javascript.internet.com/
http://www.d.umn.edu/itss/support/Training/Online/webdesign/javascript.html
http://www2.webkit.org/perf/sunspider-0.9/sunspider.html
```

Feeds and Blogs

The current way to read news and articles about JavaScript (or any subject for that matter) is via RSS feeds, Atom feeds, and blogs. Typically, these feeds are read in a dedicated RSS feed reader program, email client, or browser. Google provides a free reader, called Google Reader. The following links can be added as subscriptions to any of the feed readers:

```
http://www.alistapart.com/site/rss
http://z.about.com/6/g/javascript/b/rss2.xml?r=9F
http://z.about.com/6/o/m/javascript_t2.xml?r=9F
http://accessify.com/rss/accessify_rss.xml
http://feeds.feedburner.com/deanedwards/weblog
http://groups.google.com/group/firefoxpow/feed/rss_v2_0_msgs.xml
http://feeds2.feedburner.com/blogspot/Egta
http://blogs.msdn.com/ie/rss.xml
http://www.javascript.com/jss.rdf
http://www.webreference.com/js/tips/channels/last1512.rdf
http://labs.opera.com/rss.dml
http://www.feedzilla.com/rss/programming/javascript
http://feeds.feedburner.com/scriptaculous
http://feeds.feedburner.com/vitaminmasterfeed
http://webkit.org/blog/feed/
http://meyerweb.com/eric/thoughts/category/tech/rss2/full
http://www.d.umn.edu/itss/support/Training/Online/webdesign/webdesign.xml
http://www.webreference.com/webreference.rdf
```

Some blog sites allow you to post questions and get answers, although usually you can only comment on an article or respond to other comments. Hosts for RSS feeds and blogs are still evolving and competing for mind-share. Bloggers will frequently change hosts, so if you can't find a blogger at a link above, just do a search for the title or author to find the new host. Sometimes a blogger will leave a terminal message.

Newsgroups and Forums

The time-honored places to get quick answers to your pressing questions are online newsgroups and forums. To use newsgroups, you will need a newsgroup reader. Most browsers have dropped support for newsfeeds. Some email clients (such as Thunderbird) know how to do this. Newsgroups don't charge fees, but some of the NNTP servers do. You can use your search engine to look for a free NNTP reader.

Here are the top JavaScript-related newsgroups:

On most news servers:

```
comp.lang.javascript
```

On `news://msnews.microsoft.com`:

```
microsoft.public.scripting.jscript
microsoft.public.inetsdk.programming.scripting.jscript
microsoft.public.inetsdk.programming.dhtml_editing
microsoft.public.inetexplorer.scripting
microsoft.public.dotnet.languages.jscript
```

On `news://news.mozilla.org`:

```
mozilla.dev.tech.javascript
mozilla.dev.tech.js-engine
mozilla.dev.tech.js-engine.rhino
mozilla.dev.apps.js-debugger
```

Forums today, on the other hand, are primarily browser-based. Here are some of the top JavaScript-related forums:

```
http://en.wikipedia.org/wiki/Internet_forum
http://www.ajaxtalk.com/
http://groups.google.com/group/firefoxpow/
http://tech.groups.yahoo.com/group/ydn-javascript/
http://old.nabble.com/forum/Search.jtp?query=javascript&page=0&fsearch=y
```

Before you post a question to a newsgroup or forum, read about FAQs discussed earlier in this chapter and also use the extremely valuable newsgroup archive search facility of Google Groups. Visit the Google Groups search page at:

```
http://groups.google.com/
```

Enter the keyword or phrase into the top text box, but then also try to narrow your search by limiting the newsgroup(s) to search. For example, if you have a question about weird behavior you are experiencing with the `borderCollapse` style property in IE, enter `borderCollapse` into the search field, and then try narrowing the search to a specific newsgroup (forum) such as `comp.lang.javascript`.

If you post a question to a newsgroup or forum, you will most likely get a quick and intelligent response if you also provide either some sample code that's giving you a problem, or a link to a temporary file on your server that others can check out. Visualizing a problem you've spent days on is very hard for others. Be as specific as possible, including the browser(s) versions(s) and operating system(s) on which the code must run and the nature of the problem.

Final Thoughts

These sites are by no means the only worthwhile JavaScript and DHTML destinations on the Web. Sometimes having too many sources is as terrifying as having not enough. The links and newsgroups described in this appendix should take you a long way.

Index

SYMBOLS

: (colon)
  date/time, 288
  document.all, 91
  identifiers, 85
  name, 147
, (comma)
  for, 431
  cookies, 923
  operator, 431
  parameters, 91
  regular expressions, BC482
. (dot, period)
  document.all, 91
  dot syntax form, 86, 504
  identifiers, 85
  name, 147
  object operator, 425
  regular expressions
    metacharacter, BC468
?: (question mark/colon)
  if...else, 431
  operator, 431
; (semicolon)
  cookies, 923
  onevent, BC626
  statements, 99
:// (colon/slash-double),
  encodeURI(), 482
& (ampersand)
  bitwise AND operator, 425
  operator precedence, 434
  XHTML, 43
&& (ampersand - double)
  Boolean AND operator, 175,
    421
  operator precedence, 434
&= (ampersand/equals sign)
  assignment operator, 419
  operator precedence, 435
* (asterisk)
  multiplication connubial
    operator, 416
  operator precedence, 434
  regular expressions
    metacharacter, BC469
  XML, 368
*= (asterisk/equals sign)
  assignment operator, 419
  operator precedence, 435
*/ (asterisk/slash), 66
  statements, 492–493
@ (at symbol)
  e-mail, 77, 189
  XML elements, 366
\ (backslash), regular expressions,
  245
\\ (backslash - double), inline
  character, 227
\' (backslash/apostrophe),
  apostrophe inline character,
  227
\" (backslash/quotation mark),
  double quote inline
  character, 227
^ (caret sign)
  bitwise XOR operator, 425
  operator precedence, 434
  regular expressions
    metacharacter, BC470
^= (caret sign/equals sign)
  assignment operator, 419
  operator precedence, 435
{} (curly brackets), 128–129
  if...else, BC571
  JSON, 357
$ (dollar sign), regular expressions
  metacharacter, BC470
= (equals sign)
  assignment operator, 111, 116,
    419
  encodeURI(), 482
  operator precedence, 434–435
== (equals sign - double)
  comparison operator, 117,
    132, 374, 413
  condition, 374
  operator precedence, 434
=== (equals sign - triple), strictly
  equals comparison operator,
  413
! (exclamation mark), 68
  Boolean operator, 421
  operator precedence, 434
!= (exclamation mark/equals sign)
  comparison operator, 374
  operator precedence, 434
!== (exclamation mark/equals
  sign -double), strictly does
  not equal comparison
  operator, 413
> (greater than)
  comparison operator, 117, 413
  operator precedence, 434
>> (greater than - double)
  bitwise operator, 425
  operator precedence, 434
>>= (greater than - double/equals
  sign)
  assignment operator, 419
  operator precedence, 435
>>> (greater than - triple), bitwise
  operator, 425
>>>= (greater than - triple/equals
  sign), assignment operator,
  419
>= (greater than or equal to)
  assignment operator, 419
  comparison operator, 117, 413
  operator precedence, 434–435
(hash symbol), usemap, 1025
- (hyphen)
  document.all, 91

- (hyphen) *(continued)*
  Google Maps zoom, BC784
  identifiers, 85, 514
  minus connubial operator, 416
  `name`, 147
  operator precedence, 434
  subtraction operator, 117
  test for, BC497
-- (hyphen - double)
  decrement connubial operator, 412, 416–417
  operator precedence, 434
--> (hyphen-double/greater than)
  browsers, 98
  XHTML, 44
-= (hyphen/equals sign)
  assignment operator, 419
  operator precedence, 435
< (less than)
  comparison operator, 117, 413
  `condition`, 386
  `length`, 580
  operator precedence, 434
  XHTML, 44
<< (less than - double)
  bitwise operator, 425
  operator precedence, 434
<<= (less than -double/equals sign)
  assignment operator, 419
  operator precedence, 435
<= (less than or equal to)
  comparison operator, 117, 413
  `length`, 580
  operator precedence, 434
<!-- (less than/exclamation mark/hyphen-double)
  browsers, 98
  XHTML, 44
() (parentheses)
  Boolean operators, 433
  `functionName`, 441
  methods, 91, 507
  operator precedence, 434
% (percent sign)
  modulo connubial operator, 415
  operator precedence, 434
  URLs, 483
  string coding, 267
%= (percent sign/equals sign)
  assignment operator, 419
  operator precedence, 435
| (pipe sign)
  bitwise `OR` operator, 425
  operator precedence, 434
  regular expressions, BC470
|| (pipe sign - double)
  Boolean operator, 421
  Boolean `OR` operator, 422
  operator precedence, 434
|= (pipe sign/equals sign)
  assignment operator, 419
  operator precedence, 435
+ (plus sign)
  addition connubial operator, 416
  addition operator, 117
  Google Maps zoom, BC784
  operator precedence, 434
  regular expressions metacharacter, BC469
  string concatenation, 114, 116, 180
++ (plus sign - double)
  increment connubial operator, 416, 416
  operator precedence, 434
+= (plus sign/equals sign)
  add-by-value operator, 180, 239
  assignment operator, 419
  operator precedence, 435
  string variables, 226
? (question mark)
  `encodeURI()`, 482
  operator precedence, 435
  regular expressions metacharacter, BC469
'' (quotes - single), strings, 180
"" (quotes-double)
  identifiers, 85
  strings, 180, 226
/ (slash)
  division connubial operator, 416
  division operator, 117
  `encodeURI()`, 482
  operator precedence, 434
// (slash - double), 66
  comments, 240
  HTML, 43
// (slash-double)
  comment sequence, 44
  script comments, 100
/* (slash/asterisk), 66
  script comments, 100
  statements, 492–493
/= (slash/equals sign)
  assignment operator, 419
  operator precedence, 435
/!= (slash/exclamation mark/equals sign)
  comparison operator, 413
  inequality operator, 117, 182
[] (square brackets)
  `for`, 124
  `Array`, 314
  arrays, 155, 311
    indexes, 130, 133
  destructuring assignment, 419–420
  JSON, 357
  object operator, 426
  operator precedence, 434
[^...] (square brackets with caret sign and dots-triple), regular expressions metacharacter, BC468
[...] (square brackets with dots-triple), regular expressions metacharacter, BC468
[]= (square brackets/equals sign), assignment operator, 419
~ (tilde)
  bitwise `NOT` operator, 425
  operator precedence, 434
_ (underscore)
  global variables, 452
  identifiers, 85
  variable names, 112, 493
`- val`, negative connubial operator, 412, 416-417
`+ val`, positive connubial operator, 412, 416-417

A

a, charset, 998
<a>
  javascript: pseudo-URL, 219
  target, 1002
abbr, td, BC349
abort, addEventListener(), 616
abort(), XMLHttpRequest, BC284
above, frame, BC327
abs(), 278
absbottom, iframe.align, 871
absmiddle, iframe.align, 871
absolute units, BC229
AbsolutePosition, 599
abstract objects, 88
accelerator, BC256
acceptCharset, form, BC114
"Access disallowed from scripts at <URL> to documents at <URL>", error message, BC573
"Access is denied," error message, BC573–BC574, BC591
accessKey, 541–544
  label, BC15
aCollection(), 145
acos(), 278
action
  e-mail, BC111
  form, 157, 175
  <form>, BC114
  validate.php, 379
ActionScript, 9, 12
activeElement, document, 912
ActiveX, 9
  Java plug-ins, BC550
  object, BC550
  trusted, Microsoft, 54
ActiveX Data Objects (ADO), events, 1076–1077
ActiveXObject, 496–497
  HTTPRequest, BC277
add(), select, BC196–BC197
Add(), Dictionary, 498
addBehavior(), 516, 612–615, BC625
AddDesktopComponent(), window.external, 762
AddEvent(), 42
addEvent(), 271, 508
  addEventListener(), 1064
  event handlers, 138, 162, 174
  jsb-global.js, 138, 162
  onload, 1064
  script, 162
addEvent, onload, BC697
addEventListener, attachEvent(), 624
addEventListener(), 531, 615–620, 886, 998, BC494
  addEvent(), 1064
  DHTML cross-browser map puzzle, BC752
  event binding, 1062–1063
  forms, 174
  load, 142
  W3C DOM, 1052
AddFavorite(), window.external, 762
addImport(), styleSheet, BC220
addRange(), selection, BC53
addReadRequest(), userProfile, BC405, BC407–BC408
addRule, styleSheet, BC220–BC221
AddSearchProvider(), window.external, 762
addTotals(), BC668
  select, BC670, BC673
adjustClip(), BC420
ADO. *See* ActiveX Data Objects
Adobe Acrobat Reader, 9, 1134–1135
afterBegin
  getAdjacentText(), 644
  insertAdjacentElement(), 658
  replaceAdjacentText(), 677
afterEnd
  getAdjacentText(), 644
  insertAdjacentElement(), 658
  replaceAdjacentText(), 677
Ajax. *See* Asynchronous JavaScript and XML
alert(), BC578
  modal dialog and, 34
  onblur, 696
  window, 140, 789–790
alert dialog boxes, error messages, BC577–BC578
alertUser(), 104
align
  applet, BC451
  embed, BC462
  fieldset, BC123–BC124
  h1, BC10
  iframe, 870–871
  Image, 1007–1008
  legend, BC123–BC124
  object, BC455
  table, BC321–BC322
aLink, body, 983
alinkColor, document, 912–915
all
  IE, BC613
  rules, BC331
all[], 544–545
.all, Opera, BC615
allowTransparency
  frame, 855
  iframe, 871
Alpha(), BC266
alpha(), BC259
alt
  applet, BC451
  area, 1027
  Image, 1008–1009
  object, BC455
altHTML
  applet, BC451
  object, BC455
altKey, event, 1074–1075, 1099–1100
altLeft, event, 1075
alwaysLowered, window.open(), 806

alwaysRaised, window.open(), 806
anchorNode, selection, BC50
anchorOffset, selection, BC50–BC51
anchors, objects, 995–1002
anchors[], document, 915–917
AND
  bitwise operator, 425
  Boolean operator, 175, 421–422
animation, JavaScript, 48
answers to exercises, BC801–BC817
API. *See* application programming interface
appCodeName, BC362–BC363
appCore, window, 750
append(), 620
appendChild(), 524–525, 620–622, BC310
  document.createComment(), 957
  document.createDocumentFragment(), 957
  exception handling, 400
  insertBefore(), 661
  W3C DOM, 45, 221
  XML, 365
appendData(), Text, BC58–BC63
Apple Safari. *See* Safari
applet, BC449–BC453
  properties, BC451–BC453
  syntax, BC450
<applet>, <object>, BC550
applets
  faceless, BC537–BC542
    <object>, BC537
    onload, BC537
  Java, 10, BC524–BC549
    data type conversion, BC542, BC545
    doNewWindow(), BC547
    eval(), BC547
    faceless, BC537–BC542
    fetchText(), BC539
    getFile(), BC539
    getMember(), BC547
    HTML, BC543
    init(), BC547
    limitations, BC536–BC537
    mainwin, BC547
    object checks, BC595
    permissions, BC543
    privacy, BC592
    properties, BC528–BC529
    source code, BC538–BC539
    window, BC543
    XHTML, BC527–BC528
applets[], document, 917–918
application programming interface (API)
  DHTML, BC617–BC622
    cross-browser map puzzle, BC749
  Google Maps, BC785–BC788
  Java plug-ins, BC551–BC558
  .js, BC618
application/x-javascript, 39
apply(), functions, 446–447
Apply button, handleApply(), 843
applyBehavior(), 93
applyElement(), 622–624
applyVals(), setHotColor(), BC633
appMinorVersion, BC371–BC372
appName, BC362–BC363
  Netscape, BC612
appVersion, BC362–BC365
arc(), canvas, 1039
archive
  applet, BC451–BC452
  object, BC456
arcTo(), canvas, 1039
area, 1024–1028
  click, BC727
  coords, 1025
  javascript: pseudo-URL, 219
  properties, 1027–1028
  syntax, 1025
areas[], map, 1029–1032
arguments. *See* parameters
arguments
  functions, 441–442
  JavaScript, 56
  parameters, BC677
arithmetic
  date/time, 296–298
  operators, 117
arity(), functions, 442
Array, 55
  [] (square brackets), 314
  new, 429
  prototype, 324–326
arrays, 129–133
  [] (square brackets), 155, 311
  accessing data, 130
  browser compatibility, 355
  calculations, BC698
  callbacks, 331–333
  comprehensions, 353–354
    filter(), 352–354
    generators, 354
    iterators, 354
    map(), 353
  concatenation, 326–331
  creating, 129–130
  deconstructing assignment, 354–355
  deleting entries, 315
  document objects, 133
  document.forms, BC121
  elements, BC115
  empty, 312–313
  form elements, BC112
  getElementsByTagName(), 154
  hash tables, 321–353
  indexes, 129, 130, 132, 133, 314–318
  .js, BC663
  JSON, 359
  loops, 318–320
  methods, 326–355
  multidimensional, 320–321
  Navigator 4, 49
  objects, 311–355, 465–466
  parallel, 131–133, 315–320
    lookup tables, BC654
  populating, 313–314
  properties, 323–326
  regular expressions, 352

safe, Visual Basic, 498
select, 322
splice(), 315
string.split(), BC664, BC667
array.concat(), 326–331
array.constructor, 323
array.every(), 331–333
array.filter(), 333–334
array.forEach(), 334–335
array.indexOf(), 335–337
array.join(), 326, 337–339, 353
cookies, 923
event.dataTransfer, 705
split(), 253
array.lastIndexOf(), 335–337
array.pop(), 340–341
array.propertyName, 321
array["propertyName"], 321
array.reduce(), 341–344
array.reduceRight(), 341–344
array.reverse(), 344–346
array.shift(), 340–341
array.slice(), 346
array.some(), 331–333
array.sort(), 346–351
ASCII, 347
split(), 253
array.splice(), 352
array.toLocaleString(), 353
array.toString(), 353
array.unshift(), 340–341
ASCII, array.sort(), 347
asin(), 278
ASP, 77
assign(), 897
assignment operators, 412, 418–420
Asynchronous JavaScript and XML (Ajax), 8, 48, BC272–BC288
real-time validation, BC504
table of contents, BC682–BC687
timing problems, BC577
atan(), 278
atan2(), 278
atEnd(), Enumerator, 498
attachEvent(), 624–625, 886, 998, BC494
DHTML cross-browser map puzzle, BC752
IE, BC626
load, 142
attachToEnd(), exception handling, 401
attributes
forms, BC111
input, BC113
<script>, 95
tags, event binding, 1059–1060
XML, 363–364
elements, 366–367
attributes, Evaluator, 547
attributes[], 545–547
attributes.rules, 664
authorDate, getTime(), BC713
authoring environment. *See* web-authoring environment
autocomplete, form, BC114–BC115
automation objects, 496
autoScroll(), 827
autoupdate.enabled, BC382
availHeight, screen, BC401–BC402
availLeft, screen, BC402–BC403
availTop, screen, BC402–BC403
availWidth, screen, BC401–BC402
axis, td, BC349

B

\B, regular expressions metacharacter, BC468
\b
backspace inline character, 227
regular expressions metacharacter, BC468
back()
history, 901
window, 790–791
BackColor, document.execCommand(), 968
BackCompat, compatMode, 920
back-end programs, 8
background, BC247
body, 983
table, BC322
backgroundAttachment, BC247
backgroundColor, BC248
setBGColor(), BC621
backgroundImage, BC248
backgroundPosition, BC248
backgroundPositionX, BC248
backgroundPositionY, BC248
backgroundRepeat, BC248–BC249
backgrounds, elements, changing, BC412–BC416
backwards, window.find(), 798
Barn(), BC266
base, BC291–BC293
base formats, 273–274
baseHref, object, BC456
<base>href, 976
baseline, iframe.align, 870–871
baseURI, 547
document, 918
document.doctype, 932
object, BC456
BasicImage(), BC266–BC267
baskets, 111
batch mode validation, BC494–BC495
BBEdit, 28
beforeBegin
getAdjacentText(), 644
insertAdjacentElement(), 658
replaceAdjacentText(), 677
beforeEnd
getAdjacentText(), 644
insertAdjacentElement(), 658
replaceAdjacentText(), 677

beginPath(), canvas, 1039–1040
begin-tags, XML, 363
behave(), 93
behavior, BC256, BC624
  marquee, BC17
behavioral style, BC624
behaviorCookie, event, 1075–1076
behaviorName, addBehavior(), 612
behaviorPart, event, 1075–1076
behaviors
  CSS, BC625
  HTML, BC624
  IE, BC623–BC635
  style, BC625
behaviorUrns[], 547
below, frame, BC327
beta browsers, 52
bezierCurveTo(), canvas, 1039
bgColor
  body, 983
  document, 912–915
  marquee, BC19
  table, BC322–BC323
bgProperties, body, 984
binding
  data, tables, BC306
  events, 1059–1064
    addEventListener(), 1062–1063
    browsers, 1063–1064
    HTML, 1060
    IE, 1061–1062
    object properties, 1061
    removeEventListener(), 1062–1063
    tag attributes, 1059–1060
    W3C, 1062–1063
bits, 82, 110
bitwise operators, 412, 424–425
_blank, <form>, BC119
blank frames, 745
blendTrans(), BC261
Blinds(), BC267
<blink>, 264
blockNum, currState, BC692
blockquote, BC3
blogs, BC820–BC821
blur, addEventListener(), 616
blur(), 625–630, BC259
  text, BC164
BODY, identify(), 170
body, 65, 81, 981–993
  all[], 544
  DHTML cross-browser map puzzle, BC750
  document, 142, 911, 918–919
  document.write(), 148
  event bubbling, 1049
  event handlers, 989, 1052
  makeHTML(), BC687
  methods, 987–989
  ondrag, 705
  onload, BC551
  onunload, 1018
  OPML, BC687
  properties, 983–987
  <script>, 97
  setCapture(), 668
  syntax, 982
body objects, 907–993
body text objects, BC2–BC102
Bold, TextRange.execCommand(), BC79
bookmarks, event, 1076–1077
Boolean
  AND, 175
  callbacks, 333
  condition, 379
  data type, 110
  JSON, 359
  operators, 412, 420–424
    () (parentheses), 433
  value
    if, 163
    if...else, 163
  visibility, BC621
  window.confirm(), 140
Boolean, 55, 284
  methods, 284
  new, 429
  properties, 284
border, BC249
  frame, BC327
  frameset, 863–864
  Image, 1009
  object, BC456
  table, BC323
borderBottom, BC249
borderBottomColor, BC249
borderBottomStyle, BC249
borderBottomWidth, BC250
borderCollapse, BC254
borderColor, BC250
  frame, 855–856
  frameset, 864
  table, BC323–BC324
border-color, fieldset, BC124
borderColorDark, table, BC323–BC324
borderColorLight, table, BC323–BC324
borderLeft, BC249
borderLeftColor, BC249
borderLeftStyle, BC249
borderLeftWidth, BC250
borderRight, BC249
borderRightColor, BC249
borderRightStyle, BC249
borderRightWidth, BC250
borderSpacing, BC254
borderStyle, BC250
borderTop, BC249
borderTopColor, BC249
borderTopStyle, BC249
borderTopWidth, BC250
borderWidth, BC250
bottom, BC246
  iframe.align, 871
  name, 1000
  TextRectangle, BC100–BC102
bottomMargin, body, 984–985
boundElements, event, 1076–1077
boundingHeight, TextRange, BC68–BC71
boundingLeft, TextRange, BC68–BC71
boundingTop, TextRange, BC68–BC71
boundingWidth, TextRange, BC68–BC71

box, frame, BC327
br, BC4
break
  for, 165, 387–388
  switch, 381
  while, 389
BrowserAccess, BC602
browserLanguage, BC372–BC373
  navigator, BC373
BrowserRead, BC602
browsers. *See also specific browsers*
  -->, 98
  <!--, 98
  beta, 52
  choosing, 15–16, 28
  compatibility, 17–22, BC610–BC611
    arrays, 355
    designing for, 51–54
    event, 1064–1066
    inline branching, BC611–BC613
    object detection, 49–51, BC615–BC617
    platform equivalency, BC613–BC614
    ratings, 53–54
    scripts, 47–51
    TextRange, BC67–BC68
  cookies, 921, BC714, BC716–BC717
  crashes, BC586
  Date, 303
  date formats, 294–295
  deconstructing assignment, 355
  document.all, BC366
  document.close(), 956
  document.write(), 956
  E4X, 40
  event binding, 1063–1064
  every(), 355
  filter(), 355
  forms, 160
  frames, 192–193
  getElementById(), BC366
  NPAPI, BC542
  <object>, BC528
  object model for, 81–82
  script, 95
  select, 169
  sniffing, 51, 68
  standards support, 16–17
  tabs, 136
  tags, 43
  time stamps, BC707
  type, 40, 95, 1002
  userAgent, BC368–BC371
  versions, object detection, 51
  W3C DOM, 174
  Web Forms 2.0, BC126–BC127
  windows, 136
BrowserWrite, BC602
\B't, 245
bubbles, event, 1100
bufferDepth, screen, BC403
buildHTML, Google Maps, BC795
buildMsg(), this, 334
button, BC128–BC152
  event, 1077, 1100
  event bubbling, 1049
  event handlers, BC133–BC134
  input, 698
  methods, BC132–BC133
  properties, BC131–BC132
  syntax, BC129
buttons
  event handler, 92
  forms, BC112
  input, 163
  input element, 89

C

cache
  debugging, BC577
  images, 208–210
  XML, BC577
calcBlockState(), makeHTML(), BC690, BC691
calculate(), onchange, BC671
calculations, BC695–BC704
  arrays, BC698
  format(), BC699–BC700
  interfaces, BC696–BC697
  multiplier, BC698
  onchange, BC696
  tolerance, BC698
calendars, BC637–BC651
  DHTML, BC646–BC651
  DOM, BC651
  form controls, BC641
  javascript: URL, BC651
  location.href, BC641
  onchange, BC641
  onload, BC644
  pop-up windows, BC651
call()
  functions, 446–447
  JSObject, BC544
callbacks
  arrays, 331–333
  Boolean, 333
  every(), 333
  filter(), 333
caller
  functions, 442–443
  trace(), BC584
CamelCase format, 112, 125
Camino, BC580
cancelable, event, 1100–1101
cancelBubble
  DHTML cross-browser map puzzle, BC756
  event, 1049, 1078, 1101
  fireEvent(), 641
canHaveChildren, 547–549
canHaveHTML, 549–550
canvas, 1032–1042, BC703
  getContext(), 1035
  height, 1033
  methods, 1039–1042
  properties, 1035–1039
  syntax, 1033
  width, 1033
caption, BC341
  table, BC324
captionSide, BC254–BC255
captureEvents()
  document, 954–955, 1046
  layer, 1046
  window, 1046

C

Cascading Style Sheets (CSS), 7–8, 80
behaviors, BC625
class, BC681
className, 555
color, 220
DHTML, 220, BC609
drop-down menus, BC693
font-weight, 220
frames, 192
Hello, World script, 35–36
HTML, 6, 267
IE 4, 515
Levels (1– 3), 20–21
overflow: auto, 192
pop-up windows, BC693
positioning, 22, 897, BC411–BC447
properties, 220
rollovers, 216–219
separated development layers and, 24–25
style, 220
table of contents, BC677
tables, BC303
validation, 31
Validation Service, 31
window.getComputedStyle(), BC610
case, switch, 381
case sensitivity
getAttribute(), 683
setAttribute(), 646, 683
XHTML, 64
caseSensitive, window.find(), 798
catch, 400
throw, 402
CDATA, 43–44
CD-ROM contents, 1133–1135
ceil(), 278
cellIndex, td, BC349–BC350
cellPadding, table, BC325
cells, tables
modifying content, BC306–BC309
populating, BC306
cells
table, BC325
tr, BC345
cellSpacing, table, BC325
center, IE, 835
CGIs. *See* Common Gateway Interfaces
ch, BC339
chain.gif, BC688
change, addEventListener(), 616
channelMode, window.open(), 806
characterSet, document, 919–920
charAt(), 182
charCode, event, 1101–1104
CharSet, 562
charset, 998–999
document, 919
link, BC294
meta, BC298
checkbox
checked, 163–164, BC112
form, 163–164
onclick, 164
checkbox, BC135–BC143
event handlers, BC140–BC143
methods, BC140
properties, BC136–BC140
syntax, BC135
checked
checkbox, 163–164, BC112
checkbox, BC136–BC138
click(), 631
if, 165
radio, BC145–BC146
radio button, BC112
Checkerboard(), BC267
checkForEnter(), onkeypress, 720
checkForm(), BC521–BC522
onsubmit, 175
checkTimer(), selectedIndex, 1018
Chicago Crime, BC785
child(), XML, 370–371
child nodes, 87–88
childNodes, 522
for, 550
children, 553
nodeType, 550
W3C DOM, 647
children, frames, 191–194
children, 552–554
childNodes, 553
child-to-child, frames, 195, 742
child-to-parent, frames, 194–195, 742
chOff, BC339
Chroma(), BC267
chroma(), BC259
Chrome
debugging, BC580
document.lastModified, 943
Drosera, BC580
E4X, BC272
error messages, BC566
file:///, BC708
object literal notation, 474
chrome, window.open(), 806
circ, shape, 1027
circle
coords, 999
shape, 1027
cite, 554–555
blockquote, BC3
class, CSS, BC681
classes, Java, BC525
direct scripting, BC562–BC563
NPAPI, BC562
packages, BC562
classid, object, BC396, BC456–BC457
className, 267, 555–556, BC414
CSS, 555
layers, BC439
classpath, Java, BC542–BC543
clear, BC239
br, BC4
clear()
document, 955
selection, BC54
clearAttributes(), 630–631
clearData(), window.clipboardData, 751
clearData[], dataTransfer, 1085

clearInterval()
  DHTML cross-browser map puzzle, BC760
  setInterval(), 825
  window, 791
clearRect(), canvas, 1040–1041
clearRequest, userProfile, BC408
clearTimeout(), window, 791–793
click
  addEventListener(), 616
  area, BC727
  onchange, 173
  this, 170
click(), 631
  button, BC132–BC133
  checkbox, BC140
  dispatchEvent(), 637
  radio, BC148–BC149
clickevent, onclick, 1043
clientHeight, 556–558
clientInformation, BC361–Bc384
  properties, BC362–BC381
  syntax, BC361
  window, 751
clientLeft, 558
client-side scripts. *See* scripts
clientTop, 558
clientWidth, 556–558
clientX, event, 1078–1084, 1104–1108
clientY, event, 1078–1084, 1104–1108
clip, BC239
clip(), canvas, 1040
clipboardData
  getData(), 730
  window, 751–752
cloneContents(), range, BC25, BC30–BC31
cloneNode(), 525, 631–632
cloneRange(), range, BC30–BC31
close(), 138
  document, 955
  window, 793–794
  window.open(), 806
closed
  subWrite(), 150
  window, 752–755
closeNewWindow(), 138
closePath(), canvas, 1039–1040
closures
  functions, 456–458
  memory leaks, 458
code
  applet, BC452
  object, BC457
Code Base Principles, Mozilla, 58
codeBase
  applet, BC452
  object, BC457
codebase principal, digital certificates, BC597–BC598
codeType, object, BC458
col, BC341–BC343
  tables, BC305
    cells, BC316
ColdFusion, 77
colgroup, BC341–BC343
  tables, BC305
    cells, BC316
collapse()
  range, BC31
  selection, BC54
  TextRange, BC72–BC73
collapsed, range, BC26
collapseToEnd(), selection, BC54
collapseToStart(), selection, BC54
color, BC232
  CSS, 220
  font, BC5–BC7
  hr, BC13
colorArray, BC700
colorButtons, getColor(), 777
colorDepth, screen, BC403–BC404
cols
  frameset, 864–867
  framesets, 744
  rules, BC331
  table, BC326
  textarea, BC175
colSpan, td, BC350
columnHeads, <th>, BC670
columns, tables, modifying, BC316–BC319
columnWidths, BC671
  size, BC673
CommandState, document.queryCommand, 972–973
comments
  // (slash - double), 240
  conditional, BC528
  error messages, BC578
  scripts
    // (slash-double), 100
    /* (slash/asterisk), 100
    HTML, 43
    XHTML, 43–44
Common Gateway Interfaces (CGIs), 77, BC105
  ISPs, BC110
  serverless, 78, 312
  Submit button, BC121
commonAncestorContainer, range, BC26–BC27
compact, dl, BC358
compare(), sort(), 347
compareBoundaryPoints(), range, BC31–BC36
compareDocumentPosition(), 631–633
compareEndPoints(), TextRange, BC73–BC77
compareNode(), range, BC36
comparePoint(), range, BC36
comparison operators, 117, 412–413
  if, 374
compatibility, browsers, 17–22, BC610–BC611
  arrays, 355
  designing for, 51–54
  event, 1064–1066
  inline branching, BC611–BC613
  object detection, 49–51, BC615–BC617

compatibility, browsers, *(continued)*
- platform equivalency, BC613–BC614
- ratings, 53–54
- scripts, 47–51
- TextRange, BC67–BC68

compatMode, document, 920
compile(), RegularExpressionObject, BC485
complete
- Image, 1009–1011
- image, BC152
- readyState, 598

componentFromPoint(), 633–636
components[], window, 750
computed style, BC210
concat(), arrays, 323, 326–331
concatenation
- arrays, 326–331
- numbers, 414
- strings, 114, 116, 117, 180–181, 226, 414

condition
- <, 386
- for, 384, 386
- == (equals sign - double), 374
- Boolean, 379
- if, 374
- val1, 379

conditional comments, BC528
conditional expressions, 374–375
- null, 374
- object detection, 51

conditions, in control structures, 122
confirm()
- onreset, BC122
- window, 794–796

connubial operators, 412, 415–418
const, keyword, 493–495
constructor
- arrays, 323
- error.Object, 408
- functions, 445
- Object, 476–477
- String(), 231

container, form, 155–157
containment, inheritance, 744
contains(), 636–637
- for, 636

containsNode(), selection, BC54
content, BC239–BC240
- meta, BC298
- window, 750

Content/, jsb-global.js, 138
contentDocument, BC432
- execCommand(), 967
- frame, 856–857
- iframe, 871–872
- object, BC458

contentEditable, 558–560
contentType, document, 920
contentWindow
- frame, 857
- iframe, 872

contextMenu, div, 669
continue
- for, 388
- while, 389

control elements, form, BC105–BC106
control structures, 122–123, 373–410
- if, 122–123
- if...else, 123
- JavaScript, 56

controller, img, 211
controllers[], window, 750
conversion
- data types
  - Java applets, BC542, BC545
  - MSObject, BC545
- Date, 295–296
- numbers to strings, 116, 275–276
- strings to numbers, 115–116, 274–275

converter, oInput, 162
cookie, document, 920–930
cookieEnabled, BC373–BC374
cookies, BC705–BC714
- browsers, 921, BC714, BC716–BC717
- debugging, BC729
- Decision Helper, BC718–BC745
- document.cookie, BC719
- domain, 924
- expiration date, 923–924, BC710
- extra batches, 930
- files, 921
- framesets, BC714
- global variables, BC710
- IE, BC714
- JavaScript, 922
- loadDecisionName, BC728
- name, 923
- null, BC723
- onchange, BC729
- onload, BC706
- path, 924
- records, 922
- retrieving data, 924–930
- saving, 922–933
- SECURE, 924
- soft, BC706
  - <img>, BC713
  - lastVisit, BC710
- textarea, BC729
- values, 923

cookies.txt, Mozilla, 921
coordinate point, 91
Coordinated Universal Time (UTC), 285
coords, 999
- area, 1025–1028

Copy, TextRange.execCommand(), BC79
copy(), XML, 371
cos(), 278
count, 457
Count, Enumerator, 498
countamatic(), 457
counterIncrement, BC240
counterReset, BC240
countMe(), 457
cpuClass, BC374
crashes, browsers, BC586
createAttribute(), document, 956–957

CreateBookmark, document.execCommand(), 968
createCaption(), table, BC335
createCDATASection(), document, 957
createComment(), document, 957
createContextualFragment(), range, BC25, BC36–BC38
createControlRange(), 637
  body, 987–988
createDocumentFragment(), document, 957–958
createElement()
  document, 958–959
  exception handling, 400–401
createElementNS(), document, 958–959
createEvent(), document, 959
createEventObject(), document, 959–960
createLinearGradient(), canvas, 1040
CreateLink, document.execCommand(), 968
createMarker(), Google Maps, BC795
createNodeIterator(), 964
createNSResolver(), document, 960
CreateObject(), VBScript, 496
createPattern(), canvas, 1040
createPopup(), window, 796
createRadialGradient(), canvas, 1040
createRange()
  document, 960
  selection, BC55
createStyleSheet(), document, 960–962
createTextNode(), 68
  document, 962–963
  W3C DOM, 45
createTextRange()
  body, 988
  textarea, BC176
createTFoot(), table, BC335
createTHead(), table, BC335
createTreeWalker(), document, 963–964
Crockford, Douglas, 361
cross-confirmation validation, BC519–BC521
crypto, window, 755
CSS. *See* Cascading Style Sheets
.css, .htc, BC625
CSS1Compat, compatMode, 920
cssFloat, BC240
cssRule, BC208, BC223–BC226
  type, BC209
cssRules, BC213–BC214
cssText, BC214, BC256–BC257
  cssRule, BC223–BC224
  properties, BC223–BC226
ctrlKey, event, 1074–1075, 1099–1100
ctrlLeft, event, 1075
currentNode, TreeWalker, 991
currentStyle, 560–561, BC226–BC257, BC632
  DHTML cross-browser map puzzle, BC750
  IE5, BC210, BC416, BC618
currentTarget, event, 1108–1111
currState
  blockNum, BC692
  initExpand(), BC691
  swapState(), BC692
  XOR, BC692
currTitle, global variables, BC719, BC725
cursor, BC240
custom objects
  functions, 437–479
  methods, 460–462
  properties, 460
custom validation functions, BC498–BC499
customer care, 1135
Cut, TextRange.execCommand(), BC79

D

\D, regular expressions metacharacter, BC468
\d, regular expressions metacharacter, BC468
data
  object, BC458
  Text, BC58
data binding, tables, BC306
data islands, XML, BC272
data types
  Boolean, 110
  conversion
    Java applets, BC542, BC545
    MSObject, BC545
  description, 110
  equality of, 413–415
  Function, 110
  JSON, 358–359
  objects, 110
  strings, 110
database
  JavaScript, BC652–BC653
    sort, BC771–BC777
  lookup tables, BC652–BC653
data-entry validation, 77, BC492–BC523
  batch mode, BC494–BC495
  cross-confirmation, BC519–BC521
  date/time, BC501–BC505
  functions, BC495–BC501
    combining, BC499–BC501
    custom, BC498–BC499
    master, BC500–BC501
  industrial-strength, BC506–BC523
  keyboard trapping, BC493–BC494
  real-time, BC492–BC494, BC504–BC505
dataFld, 561–567
  event, 1076–1077
dataFormatAs, 561–567
dataPageSize, 561
  table, BC326
dataSrc, 561–567

D

dataTransfer, event, 1084–1086
DataURL, 562
date, 184–189
  arithmetic, 296–298
  calculations, 186–189
  GMT and, 184, 185
  validation, BC501–BC505
    date range-checking functions, BC513–BC519
    date validation, BC502–BC505, BC512–BC519
Date, 55, 184–189, 285–309
  browsers, 303
  conversion, 295–296
  forms, 304–309
  methods, 186, 289–292
  new, 429
  result, 287
  strings, 293
  toLocaleString(), 69
date, dateObj.setTime(), 295
dateObj, methods, 290–291
Date.parse(), 292, 295
date-time.html, 64, 67, 73
date-time.js, 66
Date.UTC(), 292, 295, 300
daughters, mothers, BC677
db, for, BC777
dd, BC357–BC359
debugging, 72, BC564–BC589
  browser crashes, BC586
  cache, BC577
  Chrome, BC580
  cookies, BC729
  Evaluator, BC581–BC582
  global variables, 452
  HTML tags, BC575
  if, BC587–BC588
  if...else, BC588
  loops, BC587–BC588
  Opera, BC580–BC581
  prevention, BC587–BC588
  Safari, BC580
  <script>, BC587
  source code, BC575–BC576
  status bar, BC578
  testing, BC588–BC589
  timing problems, BC576–BC577
  while, BC588
decimal point, test for, BC498
decimal (base-10) values, 273–274
decimal-to-hexadecimal converter function, 273–274
Decision Helper, BC715–BC746
  cookies, BC718–BC745
  dh1.html, BC728–BC730
  dh2.html, BC730–BC732
  dh3.html, BC732–BC737
  dh4.html, BC737–BC740
  dh5.html, BC740–BC744
  dhHelp.html, BC744–BC745
  dhNav.html, BC724–BC727
  HTML, BC716
  index.html, BC718–BC724
  navigation bar, BC716
  parent, BC716
declare, object, BC458
decodeURI(), 482–483, 891
decodeURIComponent(), 267, 482–483
  cookies, 923
decommafy(), BC480
deconstructing assignment, arrays, 354–355
decreaseFontSize, document.execCommand(), 968
defaultCharset, document, 931
defaultChecked
  checkbox, BC138
  radio, BC147
defaultStatus, window, 755–756
defaultValue, text, BC157–BC158
defaultView, document, 931
defer
  script, BC300
  <script>, 575
deferred scripts, 102–104, BC641
defineGetter(), 529
defineSetter(), 529
Delete
  document.execCommand(), 968
  TextRange.execCommand(), BC79
delete
  object operator, 426–427
  operator precedence, 434
deleteCaption(), table, BC335
deleteCell(), tr, BC347
deleteContents(), range, BC25, BC38–BC39
deleteData(), Text, BC58–BC63
deleteFromDocument(), selection, BC55
deleteRow()
  table, BC336
  tr, BC344
deleteRule(), styleSheet, BC222
deleteTFoot(), table, BC335
deleteTHead(), table, BC335
dependent, window.open(), 806
description
  error.Object, 408
  mimeType, BC386
  plugin, BC390–BC391
designMode, document, 931
detachElement, 624–625
detachEvent(), IE, 1062
detail, event, 1111
Deval(String expression), BC544
development layers, separation of, 24–25
DgetSlot(Int index), BC544
DgetWindow(Applet applet), BC544
dh1.html, BC723
  Decision Helper, BC728–BC730
dh2.html, Decision Helper, BC730–BC732
dh3.html, Decision Helper, BC732–BC737
dh4.html, Decision Helper, BC737–BC740

dh5.html, Decision Helper, BC740–BC744
dhHelp.html, Decision Helper, BC744–BC745
dhNav.html, Decision Helper, BC724–BC727
DHTML. *See* Dynamic HTML
dialogArguments, window, 756
dialogHeight
  IE, 835
  window, 757
dialogLeft
  IE, 835
  window, 757–758
dialogTop
  IE, 835
  window, 757–758
dialogWidth
  IE, 835
  window, 757
Dictionary, 498
digital certificates, Netscape, BC597–BC598
dimensions(), VBArray, 499
dir, 568, BC359
direction, BC241
  marquee, BC19
directive elements, HTML, BC289–BC302
directories
  window, 758–760
  window.open(), 806
disabled, 568–569
  link, BC294–BC295
  styleSheet, BC214–BC215
disassemblers, 47
dispatch Event(), W3C DOM, 1057
dispatch lookup table, BC506–BC508
dispatchEvent(), 637–640
  document.createEvent(), 959
display, BC241, BC678
  span, BC691
  toggle(), BC679
div
  contentEditable, 559
  contextMenu, 669
  img, BC679
  makeHTML(), BC691
  offsetTop, 558
  setCapture(), 668
  span, BC692
  tipStyle, BC724
dl, BC357–BC359
DOCTYPE, 64, 81, 521
  compatMode, 920
  switching, 8, 21–22, 80
doctype, document, 932
document, 82, 143–152, 569, 907–981
  body, 911
  <body>, 142
  captureEvents(), 1046
  element, BC629
  event bubbling, 1049
  event handlers, 980–981, 1052
  getElementById(), 48, 69, 519, 969–970
  getElementByTagName(), 154
  methods, 954–980
  object checks, BC594
  popup, 876
  properties, 912–954
  releaseEvents(), 1046
  setCapture(), 668
  syntax, 910
  UniversalBrowserRead, BC603
  W3C DOM, 907
  window, 760–761
  XMLDocument, BC276
document objects, 907–993. *See also* event handlers; methods; properties
  arrays, 133
  core JavaScript, separation of, 17–18
  definition, 18
  hierarchy, 503–505
document object model (DOM), 67, 70. *See also* W3C DOM
  calendars, BC651
  definition, 17
  DHTML, BC609, BC611
  E4X, 364
  evolution of, 79
  families, 509
    IE 4, 19
    IE 5, 19
    NN4, 19, 510
  forms, 153, 155–157
  HTML, BC609
  IE, BC609, BC611
  nodes, 263
  positioning, BC411–BC447
  runtime error, 174
  Web Forms 2.0, BC127
Document Type Definition (DTD), 516
document.all, 90–91
  browsers, BC366
  Enumerator, 498
  IE4, 49
  object detection, BC616
document.body, engage(), BC629
document.close()
  browsers, 956
  cursor, 956
  document.write(), 148
  windows.open(), 812
document.cookie, BC716
  cookies, BC719
document.createElement, appendChild(), 620
document.createElement(), 146–147, BC307
  applyElement(), 622
  replaceNode(), 678
  swapNode(), 689
document.createTextNode(), 145–146, BC307
  W3C DOM, 147
document.defaultView, BC210
document.domain, BC593–BC594
documentElement, document, 933
document.formReference.elementName, 518
document.forms, 154–155, BC582
  arrays, BC121

document.forms[], 145–146
document.getElementById, 42
  iframe, 196
document.getElementById(), 86, 88, 91, 519, BC115
  contains(), 636
  W3C DOM, 465, BC610
  walkChildren(), 553
document.getElementsByTagName(), 144–145
document.images
  if, 49
  imageOn(), 212
document.images[], 146
document.layers, Netscape, 49
document.links, name, 1000
document.recalc(), setExpression(), 685
document.selection, type, BC50
document.title, <title>, BC302
documentURI, document, 933
document.write(), 100–101, 147–152, BC576, BC670, BC735
  base, BC292
  browsers, 956
  cursor, 956
  document.close(), 147
  DOM, 101
  expression evaluation, 113, 114
  HTML, 147
  <img>, BC713
  insertAdjacentHTML(), 659
  insertAdjacentText(), 659
  newAsOf(), BC709
  object checks, BC594
  output streams, 148
  tables, BC306
  web pages, 147
  window, 749
  windows.open(), 811
  XHTML, 147
documentwriteln(), BC576
doDisplay(), eval(), 465
DOM. *See* document object model
DOMActivate, addEventListener(), 616
domain, cookies, 924
domain, document, 933–934
DOMAttrModified, addEventListener(), 616
DOMCharacterDataModified, addEventListener(), 616
DOMFocusOut, addEventListener(), 616
DOMImplementation, hasFeature(), 942
DOMNodeInserted, addEventListener(), 616
DOMNodeInsertedIntoDocument, addEventListener(), 616
DOMNodeRemoved, addEventListener(), 616
DOMNodeRemovedFromDocument, addEventListener(), 616
DOMSubtreeModified, addEventListener(), 616
doNewWindow(), Java applets, BC547
doReadRequest(), userProfile, BC408–BC410
Dortch, Bill, 925–930, BC719
doScroll(), 640
  body, 988–989
doValidate(), BC507, BC508
do-while, 390
download, BC624
dragDrop(), 640–641
draggable, BC629
draggedElem, layers, BC439
drag.htc, BC629
dragIt()
  DHTML cross-browser map puzzle, BC756
  layers, BC440
Dragonfly, Opera, BC580–BC581
drawImage(), canvas, 1040
drawTextTable(), BC777
Dreamweaver, 27
DremoveMember(String elementName), BC544
drop-down menus, CSS, BC693
dropEffect, dataTransfer, 1085
DropShadow(), BC267–BC268
dropShadow(), BC259–BC260
Drosera, WebKit, BC580
DsetMember (String elementName), BC544
DsetSlot(int index. Object value), BC544
dt, BC357–BC359
DTD. *See* Document Type Definition
dump(), window, 796
duplicate(), TextRange, BC77–BC78
Dynamic HTML (DHTML), 512, BC608–BC622
  API, BC617–BC622
  calendars, BC646–BC651
  cross-browser map puzzle, BC747–BC763
  CSS, 220, BC609
  DOM, BC609, BC611
  innerHTML, 221–222
  layers, BC412
  non-DHTML browsers, BC619
  positioning and, 22
  scripts, 78
  shiftTo(), BC620
  style sheets, 220
  tables, BC646–BC651
  techniques, 220–222
  versions, BC612
  W3C DOM, 221
dynamic tables, BC641–BC645
dynsrc, Image, 1011

E

E (Math.E), 277
E4X. *See* ECMAScript for XML
ECMA standards body, 18, 109
ECMA-262 standard, 18, 109, 179
ECMAScript, 11–12, 18, 48
  JSON, 359
ECMAScript for XML (E4X), 12, 363–372, BC272–BC288
  browsers, 40

DOM, 364
embedding in HTML, embedding, 370
methods, 370–372, BC288
MIME, BC287
edge, IE, 835
effectAllowed, dataTransfer, 1085
element
currentStyle, 561
document, BC629
this, BC625
elements
backgrounds, changing, BC412–BC416
control, form, BC105–BC106
directive, HTML, BC289–BC302
form
arrays, BC112
functions, BC106–BC111
frames, 746
functions, 170–173
href, 997
HTML, 997
IE behaviors, BC627–BC631
<input>, BC113–BC114
naming, 85–86
style sheets, 263
target, 997
XML, BC273–BC274
@ (at symbol), 366
element IDs. *See* identifiers
element referencing. *See* references
elementFromPoint()
document, 964–966
IE, 964
elementRef.style, BC226–BC257
elements, arrays, BC115
elements(), form, 157–158
elements[], form, BC114–BC117
else, 123
if, 376
e-mail
@ (at symbol), 77, 189
action, BC111
enctype, BC111
forms, BC110–BC111
ISPs, BC110
embed, BC460–BC463
properties, BC462–BC463
syntax, BC461
<embed>, <object>, BC396
embedding
E4X in HTML, 370
JavaScript expressions, 366
objects, 467–468, BC448–BC464
embeds[], document, 934
empty(), selection, BC55
empty arrays, 312–313
empty div, innerHTML, BC687
emptyCells, BC255
empty.gif, BC688
empty/null entry, test for, BC496
enabled, document.queryCommand, 972–973
enabledPlugin, mimeType, BC386–BC387
enablePrivilege(), BC601
PrivilegeManager, BC606
encapsulation, objects, 465
encodeURI(), 482–483, 891
encodeURIComponent(), 267, 482–483
cookies, 923
encoding, form, BC118
enctype
e-mail, BC111
form, 157
MIME, BC118
W3C DOM, BC118
endContainer, range, BC27–BC28
endOffset, range, BC28–BC29
end-tags, XML, 363
EndToEnd, setEndPoint(), BC98
EndToStart, setEndPoint(), BC98
engage()
DHTML cross-browser map puzzle, BC754–BC755
document.body, BC629
layers, BC439–BC440
entities, document.doctype, 932
Enumerator, 498–499
Error, 55
new, 429
errors
objects, 407–410
scripts, viewing, 104–105
trapping, JavaScript, 57
error messages, BC565–BC567
alert dialog boxes, BC577–BC578
Chrome, BC566
comments, BC578
details, BC567–BC574
filenames, BC567
Firefox, BC565–BC566
location, BC567–BC569
multiples, BC566–BC567
Opera, BC566
privileges, BC606
Safari, BC566
text, BC569–BC574
errorObject, 407–410
error.Object, 409–410
escape(), 483–484, 891
encodeURIComponent(), 482
escape sequences, 180
EscapeChar, 562
eval(), 484–485, BC614
doDisplay(), 465
execScript(), 797
format(), BC699
Java applets, BC547
JSObject, BC544
JSON, 359, 361
security and, 272
subwin, BC547
window.open(), BC547
EvalError, 55
evaluate(), document, 966
Evaluator
attributes, 547
debugging, BC581–BC582
execScript(), 797
global variables, 797
navigator.appname, BC612
string.split(), 253
Evaluator Jr., 113

Evaluator Sr., 57–58
evaluator.js, BC581
  trace(), BC583
evalulator.html, 58
evalulator.js, 58
Event
  dispatchEvent(), 637
  static objects, 1045, 1098
  W3C DOM, 1045, 1098
  window, 761
event, 1096
  browser compatibility, 1064–1066
  cancelBubble, 1049
  evt, 1065
  IE, 1062
  IE4, 1074
  keyCode, 721
  methods, 1065–1066, 1119–1121
  Mozilla, 1097–1121
    properties, 1099–1119
    syntax, 1098
  NN6, 1097–1121
    properties, 1099–1119
    syntax, 1098
  properties, 1065–1066
  returnValue, 1050
  script, BC300–BC301
  <script>, 41
  syntax, 1074
  window, 761, 1074
events, 92–93, 1043–1121
  ADO, 1076–1077
  binding, object properties, 1061
  bubbling, 515
  IE, properties, 1074–1097
  IE4, 1044, 1047–1052, 1071–1074
  IE5, 1052
  keyCode, 1069
  modifers, 1047
  modifier keys, 1066–1068
  mouse button, 1068–1069
  Mozilla, 959
  NN4, 1045–1047
  NN6, 1071–1074
  onkeydown, 1069
  onkeyup, 1069
  propagation of, 1045–1059
  references, 1059
  scripts, 1044
  types of, 1070–1074
  W3C, 1071–1074
  W3C DOM, 1044, 1052–1058, 1098
event binding, 1059–1064
  addEventListener(), 1062–1063
  browsers, 1063–1064
  HTML, 1060
  IE, 1061–1062
  removeEventListener(), 1062–1063
  tag attributes, 1059–1060
  W3C, 1062–1063
event bubbling
  IE4, 1047–1049
  mouse, BC439
  W3C DOM, 1053–1057
event handlers, 691–738
  addEvent(), 138, 162, 174, 271, 508
  body, 989, 1052
  button, BC133–BC134
  checkbox, BC140–BC143
  document, 980–981, 1052
  form, BC121–BC122
  function call from, 125
  generic HTML element objects, 691–738
  IE, 174
    behaviors, BC626
  Image, 1022–1024
  initialize(), 162
  link, BC297
  marquee, BC21–BC22
  radio, BC149–BC150
  select, BC198–BC199
  target, 997
  text, BC167–BC171
  values, BC111
  variables, BC736
  window, 848–854, 1052
event.clientX, 965
event.clientY, 965
event.dataTransfer, Array.Join(), 705
eventObj.cancelDefault(), onclick, 698
eventPhase, 1053
  event, 1111
event.propertyName, 733
event.returnValue, 707
  onbeforedeactivate, 696
  onbeforepaste, 695, 730
  onfocus, 715
every()
  arrays, 323, 331–333
  browsers, 355
  callbacks, 333
evt
  event, 1065
  Something(), 1061
  window.event, 1065
exception handling, 397–398
  objects, 404–405
  strings, 403–404
  throw, 402–407
  try-catch-finally, 398–402
exec()
  regular expressions, BC473-BC474, BC481
  RegularExpressionObject, BC485–BC486
execCommand()
  document, 966–969
  TextRange, BC78–BC81
execScript()
  eval(), 797
  Evaluator, 797
  window, 796–797
exercises, answers to, BC801–BC817
Exists(), Enumerator, 498
exp(), 278
expand, TextRange, BC81–BC82
expandEntityReference, TreeWalker, 991–992
expando, document, 934–935
expansionstate, head, BC690
"Expected <something>", error message, BC569
expiration date, cookies, 923–924

exponents, 273
export, signed scripts, BC606–BC607
expression, switch, 381
expressions. *See also* regular expressions
conditional, 374–375
null, 374
object detection, 51
evaluation, 112–114
initial, loops, 124
JavaScript, embedding, 366
lambda, 439
update, for, 124, 384
extend(), selection, BC55
extendRow(), BC668
Extensible Markup Language (XML), 363–364, BC272–BC288. *See also* Asynchronous JavaScript and XML; ECMAScript for XML
* (asterisk), 368
appendChild(), 365
attributes, 363–364
cache, BC577
child(), 370–371
copy(), 371–372
data islands, BC272
data transformation, BC764–BC781
elements, 364–366, BC273–BC274
@ (at sign), 366
attributes, 366–367
object.property, 364
object[property], 364
HTML, 364
name(), 371–372
nodes, BC273–BC274
objects, 364–370
serialization, 370
parent-child relationship, BC684
table of contents, BC682–BC687
tags, 363
public:, BC627
toString(), 372
toXMLString(), 372
W3C DOM, BC272
XHTML, 364
external, window, 761–762
extractContents(), range, BC40

F

\f, form feed inline character, 228
face, font, BC7
faceless applets, BC537–BC542
<object>, BC537
onload, BC537
factorial(), 454–456
Fade(), BC268
false, Boolean math, 420–424
FAQs, BC818–BC819
fetchText(), Java applets, BC539
fgColor, document, 912–915
Fibonacci sequence, 332
FieldDelim, 562
fieldset, BC103
border-color, BC124
form controls, BC122–BC124
file, BC204–BC205
XMLHttpRequest, BC280
file:, src, BC595
file:///, Chrome, BC708
File Transfer Protocol (ftp), 893
FileAccess, BC602
fileAccess(), BC606–BC607
fileCreatedDate
document, 934–937
Image, 1011–1012
fileModifiedDate
document, 934–937
Image, 1011–1012
fileName, error.Object, 409
filename, plugin, BC390–BC391
filenames, error messages, BC567
FileRead, BC602
fileSize
document, 934–937
Image, 1011–1012
fileUpdatedDate, Image, 1011–1012
FileWrite, BC602
fill(), canvas, 1040–1041
fillRect(), canvas, 1040–1041
fillStyle, canvas, 1035–1036
Filter, 565
filter, BC258–BC271
IE4, BC259–BC262
IE5, BC265–BC271
TreeWalker, 991–992
filter functions, data-validation functions, BC495–BC501
filter(), BC241
array comprehensions, 353–354
arrays, 323, 333–334
browsers, 355
callbacks, 333
filters[], 569
finally, 402
find(), window, 797–798
findText(), TextRange, BC82–BC88
FireBug, Firefox, BC580
fireEvent(), 641–644
document.createEventObject(), 959
srcElement, 1050
Firefox
compatibility, 17–22
deconstructing assignment, 355
E4X, BC272
error messages, BC565–BC566
FireBug, BC580
IE, 17
map(), 340
object literal notation, 474
select, 170
warnings, BC574–BC575
firstChild, 522, 570–572
W3C DOM, 647
firstChild(), TreeWalker, 992
firstChild.nodeValue, BC273
firstPage(), table, BC336
flag, trace(), BC584
Flash plug-in, 9
flipH(), BC260
flipV(), BC260

floating-point numbers
exponents, 273
integers, 110, 116, 270–274
parseFloat(), BC669
floor(), 184, 278
focus(), 625–631, BC116
dispatchEvent(), 637
select(), BC662
setActive(), 682
text, BC165
focusNode, selection, BC50
focusOffset, selection, BC50–BC51
font, BC5–BC9, BC233
fonts, h1, 73
fontFamily, BC233
FontName, document.execCommand(), 968
FontSize, document.execCommand(), 968
fontSize, BC233
fontSizeAdjust, BC233
fontSmoothingEnabled, screen, BC404
fontStretch, BC233
fontStyle, 560, BC234
fontVariant, BC234
fontWeight, BC234
font-weight, CSS, 220
footnotes, frames, 196–202
for, 124, 384–388, BC641
, (comma), 431
break, 165, 387–388
childNodes, 550
condition, 384, 386
contains(), 636
continue, 388
control structures, 56
db, BC777
form.elements, 158
i, 385
length, 580
makeTitleRow(), BC670
return, BC734
<script>, 41
style.display, BC677
update expression, 124, 384
var, 384
for(), arrays, 333
forEach(), arrays, 323, 334–335
foreColor, document.execCommand(), 968
for-in, 390–392
for...in, ActiveXObject, 498
FORM, identify(), 170
form, 153–158
action, 175
button, BC131
checkbox, 163–164
container, 155–157
control elements, BC105–BC106
elements(), 157–158
event bubbling, 1049
fieldset, BC123–BC124
form controls, BC103
HTML, 157
label, BC15
legend, BC123–BC124
list, 155
methods, BC119–BC121
object, 155–157
object, BC458
onsubmit, 1050, BC111
parameters, BC106–BC107
properties, 157–158, BC114–BC119
text, BC158–BC159
this.form, BC108
validate.php, 174
<form>
action, BC114
document.forms[], 145
e-mail, BC110
HTML, 158
id, 154
onsubmit, BC122
target, BC119
forms, 153–177
addEventListener(), 174
attributes, BC111
browsers, 160
buttons, BC112
controls
calendars, BC641
fieldset, BC122–BC124
form, BC103
legend, BC122–BC124
maxLength, BC113
objects, 158–170
text, 159–162
Date, 304–309
DOM, 153, 155–157
elements
arrays, BC112
functions, BC106–BC111
e-mail, BC110–BC111
functions, BC106–BC111
getElementById(), BC104
input, 160
lookup tables, BC655–BC662
objects, BC103–BC126
prevalidating, 173–177
preventDefault(), 174
processData(), BC107
submit, 173–177
text, BC153–BC176
format()
calculations, BC699–BC700
eval(), BC699
FormatBlock, document.execCommand(), 968
form.elements[], 157–158
<form>...</form>, BC105–BC106
forms[]
document, 937–939
object checks, BC595
forums, BC821–BC822
forward(), window, 790–791
foundArray, regular expressions, BC473, BC475
foundMatch, BC662
frame
parentNode, 746
properties, 855–861
references, 196
syntax, 855
table, BC326–BC329
<Frame>, 195
frames, 739–880

blank, 745
browsers, 192–193
children, 191–194
child-to-child, 195, 742
child-to-parent, 194–195, 742
creating, 740
CSS, 192
elements, 746
ensuring, 743–744
footnotes, 196–202
HTML, 192
object model, 740–742
parents, 191–194
parent-to-child, 194, 742
preventing, 743
print(), 815
references, 194–195, 742
scripts, 191–206
source code, 745–746
synchronization, 744–745
<frame>, <frameset>, 741
frameBorder
frame, 857
frameset, 867–868
iframe, 872
frameborder, layers, BC432
frameElement, window, 762–763
frames, window, 763–765
frames[], document, 939–940
frameset
properties, 863–868
syntax, 862
<frameset>, 862
<frame>, 741
framesets
cols, 744
cookies, BC714
onload, 744
rows, 744
frameSpacing, frameset, 868
fritzy, 157
fromElement, event, 1086–1088
FrontPage, 27
ftp. *See* File Transfer Protocol
fullName(), onclick, 165
fullScreen, window, 765
fullscreen, window.open(), 806
Function, data types, 110
Function, 55
new, 429
syntax, 437–438
function(), operator precedence, 434
functions
application notes, 447–458
arguments, 441–442
closures, 456–458
creating, 125
custom objects, 437–479
custom validation, BC498–BC499
data-entry validation, BC495–BC501
decimal-to-hexadecimal converter, 273–274
elements, 170–173
filter, data-validation functions, BC495–BC501
form elements, BC106–BC111
forms, BC106–BC111
function call from event handler, 125
global, 481–499
invoking, 447–448
JavaScript, values, 56–57
libraries, 454–456
master validation, BC500–BC501
methods, 446–447
nesting, 440
parameters, 125–126, 441
properties, 441–445
recursion, 454
<script>, 101
script statements, 101
text, 454
utility, strings, 261–267
"Function does not always return a value," error message, BC573
functionName, 438
() (parentheses), 441

G

g, local variables, 452
garbage collection, host environment, 57
GDownloadUrl, Google Maps, BC795
geckoActiveXObject(), window, 799
generators, array comprehensions, 354
generic HTML element objects, 537–738
event handlers, 691–738
methods, 612–691
properties, 541–612
geocoordinates, Google Maps, BC788
GET, unconditional, 898
get, method, BC118
getAdjacentText(), 644–645
getAllResponseHeaders(), BC285
getAttribute(), 547
getAttribute()
case sensitivity, 683
userProfile, BC410
getAttributeNode(), 691–647
getAttributeNodeNS(), 648
getAttributeNS(), 648
getBookmark(), TextRange, BC89
getBoundingClientRect(), 648–654
getClientRects(), 648–654
getColor(), colorButtons, 777
getComputedStyle()
W3C DOM, BC210
window, 799
getContext(), canvas, 1035, 1041
getCountDown(), 301
getData()
clipboardData, 730
dataTransfer, 1085
onpaste, 730
window.clipboardData, 751
getDate(), 185

getDay(), 186
dateObj, 290
getdh3Factor(), BC733
getdh4Performance(), BC740
getElementById(), 91
all[], 544
browsers, BC366
document, 48, 69, 86, 90–91, 519, 969–970
document.all, 91–92
forms, BC104
hr, BC10
id, 154
object detection, BC616
output, 69
W3C, BC617
getElementByTagName(), document, 154
getElementsByName(), document, 970–971
getElementsByTagName(), 133, 654, 690, BC273
arrays, 154
W3C DOM, 911
getElementsByTagNameNS(), 654–655
getErrorObj(), name, 406
getExpression(), 655
getFeature(), 655
getFile(), Java applets, BC539
getFormData(), 838, 843
getFullYear(), 186, 292
dateObj, 290
getGrossOffsetLeft(), BC424, BC425
getGrossOffsetTop(), BC424, BC425
getHours(), 186
dateObj, 290
getItem(), VBArray, 499
getMember()
Java applets, BC547
JSObject, BC544
getMilliseconds(), dateObj, 290
getMinutes(), 186
getMonth(), 186
dateObj, 290
getNetOffsetLeft(), BC425
getnetOffsetTop(), BC425
getObject(), library, BC620
getObject, W3C, BC617
getParentLayer(), BC425
getRangeAt(), selection, BC55
getResponseHeader(), BC285
getSeconds(), 186
dateObj, 290
getSelection(), window, 799–801
getSlot(), JSObject, BC544
getters, BC528–BC529
objects, 468–470
getTime(), 186
authorDate, BC713
cookies, 923
dateObj, 289
getTimeUntil(), 301
getTimezoneOffset(), dateObj, 291
time zones, 293
getuserData(), 655–656
getUTCDate(), dateObj, 290
getUTCDay(), dateObj, 290
getUTCFullYear(), dateObj, 290
getUTCHours(), dateObj, 290
getUTCMilliseconds(), dateObj, 290
getUTCMinutes(), DateObj, 290
getUTCMonth(), dateObj, 290
getUTCSeconds(), dateObj, 290
getWindow(), JSObject, BC544
getYear(), 186
dateObj, 290
Global, 55
global
regular expressions, BC472
RegularExpressionObject, BC484
global functions, 481–499
global scope, 127
host environment, 55
variables, 55
global variables, 127–128, 448–453
_ (underscore), 452
cookies, BC710
currTitle, BC719, BC725
debugging, 452
DHTML cross-browser map puzzle, BC749
Evaluator, 797
IE, 465
behaviors, BC626
inline branching, BC613
table of contents, BC677, BC688
globalAlpha, canvas, 1036–1037
globalCompositeOperation, canvas, 1037
Glow(), BC268
glow(), BC260
GMT. *See* Greenwich Mean Time
go(), history, 901
Google Maps, 8, BC782–BC798
API, BC785–BC788
geocoordinates, BC788
mashups, BC788–BC799
Google Maps Directory, BC785
Google Moon, BC785
goPrev(), BC725
goto, 394
graceful degradation, 22–23, 68
graphics, BC695–BC704
gray(), BC260
Greenwich Mean Time (GMT), 184–186, 285–286, BC711
milliseconds, BC713
time tracking, BC706–BC707
groups, rules, BC331
guid, BC550

H

h1, 65, BC9–BC10, BC641
fonts, 73
Hammond, David, 7
handleApply(), Apply button, 843
handleBottom, componentFromPoint(), 634
handleBottomLeft, componentFromPoint(), 634

handleBottomRight,
componentFromPoint(),
634
handleClick(), setCapture(),
670
handleEvent(), 1047
handleLeft,
componentFromPoint(),
634
handleOK(), 841
window.returnValue,
837–838
handleRight,
componentFromPoint(),
634
handleTop,
componentFromPoint(),
634
handleTopLeft,
componentFromPoint(),
634
handleTopRight,
componentFromPoint(),
634
hard, textarea, BC174
hard reload, 898
hasAttribute(), 656
hasAttributeNS(), 656
hasAttributes(), 701
hasChildNodes(), 525,
656–657
hasFeature(),
DOMImplementation, 942
hash, 884–886, 999
area, 1028
hash tables, arrays, 321–353
hasOwnProperty(), Object, 477
head, 64–65, 81, BC291
DHTML cross-browser map
puzzle, BC749
expansionstate, BC690
link, 73
OPML, BC690
script, 67, 96, 162
headers, td, BC349
height, 572–573, BC247
applet, BC452
<canvas>, 1033
document, 940–941
embed, BC462
frame, 858
iframe, 872
Image, 1012
marquee, BC20
object, BC458–BC459
pixels, 208
screen, BC401–BC402
table, BC329–BC330
td, BC351
tr, BC345–BC346
window.open(), 806
Hello, World script, 31–36
helper applications, 9. *See also*
plug-ins
hexadecimal triplet format, 273
hexidecimal (base-16) integer
values, 273–274
Hickson, Ian, 7
hidden, BC172–BC173
embed, BC462
hide(), BC621
popup, 877–880
hideFocus, 573–574
hideFocus, IE5, 627
hideTip(), BC726
hierarchies. *See* document objects;
node trees
HIERARCHY_REQUEST_ERR, 400
highlight(), setCapture(),
669
history, 82, 900–906
back(), 901
go(), 901
methods, 903–906
properties, 901–903
security restrictions, 881
window, 765
history.back(), 901
history.forward(), 901
history.go(), 905–906
home(), window, 801
host, 886–890, 999
area, 1028
host environment
garbage collection, 57
global scope, 55
JavaScript, 54
memory, 57
hostname, 890, 999
area, 1028
:hover, W3C DOM, 216
hr, BC10–BC14, BC641
properties, BC11–BC12
syntax, BC10
href, 890–892, 999
area, 1028
base, BC292
charset, 999
elements, 997
javascript: pseudo-URL,
219, 247, BC679
link, BC295
styleSheet, BC215
target, 1001
toggle(), BC677, BC681
W3C DOM, 999
hrefLang, link, BC295
hreflang, 1000
hsides, frame, BC327
hspace
applet, BC452–BC453
iframe, 872–873
Image, 1013
marquee, BC20
object, BC459
HTAs. *See* HTML applications
.htc, BC624
.css, BC625
.js, BC625
<script>, BC625
HTML. *See* Hypertext Markup
Language
HTML, event bubbling, 1049
html, 65, 81, BC289–BC291
all[], 544
.html, document.write(), 148
HTML applications (HTAs), 516
HTML Validator, 7, 20, 31
htmlFor
label, BC16, BC126
script, BC300–BC301
htmlText, TextRange,
BC71–BC72

http. *See* Hypertext Transfer Protocol
http://, .js, 38
httpEquiv, meta, BC299
https. *See* HTTP-Secure
https:, location, BC593
HTTP-Secure (https), 893
hybrid tables, BC645
Hypertext Markup Language (HTML), 5–7. *See also* Dynamic HTML; XHTML
// (slash - double), 43
behaviors, BC624
CSS, 6, 267
Decision Helper, BC716
directive elements, BC289–BC302
documents
intelligent, 78
loading process, 82–85
node tree, 88
object model containment hierarchy, 83–88
scripts, 95–107
structure, 79–81
document.write(), 147
DOM, BC609
E4X, 370
elements, 997
embedding E4X in, 370
event binding, 1060
fieldset, BC123
form, 157
<form>, 158
frames, 192
generic element objects, 537–738
Google Maps, BC795
Hello, World script, 32
iframe, BC294
imageName, 207
Java
applets, BC543
plug-ins, BC550–BC551
JavaScript, 37–51
layers, BC432–BC434
li, 216
method, BC118
noscript, 44
order forms, BC667–BC673
<script>, 40
scripts, comments, 43
select, 167
separated development layers and, 24–25
strings, 263–267
style, BC211
style sheets, BC768
table of contents, BC688–BC691
tables, BC303
tags, 77–81
debugging, BC575
positioning, 96
type, 95
target, 997, BC119
trace(), BC586
type, 40
update flags, BC709
UTF-8, 63
validation, 7, 20, 31
version 4.01, 7, 20, 80–81
version 5, 7, 81
W3C specification, 20, 31
Web Forms 2.0, BC126
XML, 364
Hypertext Transfer Protocol (http), 893

I

i
for, 385
loop counter, 385–386, 452
ibound(), VBArray, 499
id, 82, 85, 267, 574, BC550
DOM, 159
<form>, 154
getElementById(), 154
join, 155
span, 221
styleSheet, BC215–BC216
identifiers
hyphen, 514
variable names, 495–496
IE. *See* Internet Explorer
IEEE doubles, BC545
if, 122–123
Boolean value, 163
checked, 165
comparison operators, 374
condition, 374
control structures, 56, 123
debugging, BC587–BC588
decisions, 373–379
document.images, 49
else, 376
indexes, 318
name, 1000
submit, 175
type, 175
window.confirm(), 140
window.prompt(), 141
if (!document.getElementById) return:, 53
if...else, 123
{}, BC571
?:, 431
Boolean value, 163
control structures, 123
debugging, BC588
decisions, 373–379
inline branching, BC611
nesting, 376–377
switch, 380
window.confirm(), 140
window.prompt(), 141
iframe, 196, 868–874
HTML, BC294
layers, BC432
syntax, 869
if-test, 48
ignoreCase, regular expressions, BC472
Image, 511, 1003–1024
event handlers, 1022–1024
memory, 1006
new, 429
properties, 1007–1002
setInterval(), 1006
syntax, 1004–1005
image, BC151–BC152
images, 207–222
basic object model, 511
cache, 208–210
interchangeable, 208
object checks, BC595

objects, 207–216
pre-caching, 208–210
rollovers, 211–216
without scripts, 216–219
select, 210
src, 213
imageDB, BC700
imageLibrary, loadCached(), 210
imageName, HTML, 207
imageOff(), onmouseout, 213
imageOn()
document.images, 212
onmouseover, 213
images, object detection, 49
images[], document, 941–942
imeMode, BC257
img, 1003–1024
controller, 211
div, BC679
<img>, 208
src, 208
toggle(), BC679, BC691–BC692
<img>, 1005
document.write(), BC713
img, 208
<map>, 1025
soft cookies, BC713
src, 208
immediate scripting, BC641
immediate statements, 100
implementation, document, 942
import
signed scripts, BC606–BC607
style sheets, BC209
@import, style, BC209
imports, styleSheet, BC216
in, object operator, 427–428
increaseFontSize, document.execCommand(), 968
Indent, document.execCommand(), 968
indexes
arrays, 129–133, 314–318
if, 318
select, 167
index.html, BC676
Decision Helper, BC718–BC724
window.confirm(), 141
indexOf(), 183, 240
arrays, 323, 335–337
Indterm, document.queryCommand, 972–973
industrial-strength data-entry validation, BC506–BC523
dispatch lookup table, BC506–BC508
sample form for, BC509
sample validations, BC508–BC521
structure, BC506
inheritance
containment, 744
fieldset, BC123
JavaScript, 56
prototype, 56, 472–474
init()
DHTML cross-browser map puzzle, BC761
Java applets, BC547
loadXMLDoc(), BC686
onload, BC630, BC678–BC679, BC769
setExpression(), 686
initArray(), DHTML cross-browser map puzzle, BC761
initAudioAPI(), BC551–BC552
initEvent(), 959
event, 1119–1120
initExpand(), currState, BC691
initial expression, loops, 124
initialCaps(), String, 234
initialize(), event handlers, 162
initializing variables, 111
initKeyEvent(), event, 1119–1120
initMouseEvent(), event, 1119–1120
initMutationEvent(), event, 1119–1120
initState, BC678
initUIEvent(), event, 1119–1120
inline branching, browser compatibility, BC611–BC613
inline characters, strings, 227–228
innerHeight
window, 766–768
window.open(), 806
innerHTML, 513, 574–577, BC310
DHTML, 221–222
empty div, BC687
firstChild, 570
IE4, 975
range, BC25
span, 731
transferHTML(), BC432
userProfile, BC407
W3C, BC116, BC504, BC687
W3C DOM, 528–529, BC307
innerText, 513, 574–577
firstChild, 570
IE, 585–586
setupDrag(), 705
innerWidth
window, 766–768
window.open(), 806
INPUT, identify(), 170
Input, 158, BC671
attributes, BC113
button, 698
buttons, 163
checkbox, BC136
elements, BC113–BC114
forms, 160
label, BC125
RegExp, BC488
textarea, BC174
type, BC115
<input type="button">, BC112
inputEncoding, document, 942
inRange(), BC499, BC501
TextRange, BC89–BC90
insertAdjacentElement(), 657–659

insertAdjacentHTML(), 659–661
IE, BC309
insertAdjacentText(), 659–661
insertBefore(), 525, 661–663
document.createComment(), 957
removeChild(), 675
InsertButton, TextRange.execCommand(), BC79
insertCell(), BC310
tr, BC347
insertData(), Text, BC58–BC63
InsertDateTime(), 68
InsertFieldset, TextRange.execCommand(), BC79
InsertHorizontalRule
document.execCommand(), 968
TextRange.execCommand(), BC79
InsertIFrame, TextRange.execCommand(), BC79
InsertImage, document.execCommand(), 968
InsertInputButton, TextRange.execCommand(), BC79
InsertInputCheckbox, TextRange.execCommand(), BC79
InsertInputFileUpload, TextRange.execCommand(), BC79
InsertInputImage, TextRange.execCommand(), BC79
InsertInputRadio, TextRange.execCommand(), BC79
InsertInputReset, TextRange.execCommand(), BC79
InsertInputSubmit, TextRange.execCommand(), BC79
InsertInputText, TextRange.execCommand(), BC79
InsertMarquee, TextRange.execCommand(), BC80
insertNode(), range, BC41–BC43
InsertOrderedList, TextRange.execCommand(), BC80
InsertParagraph
document.execCommand(), 968
TextRange.execCommand(), BC80
insertRow(), BC310
table, BC336
tr, BC344
insertRule(), styleSheet, BC222
InsertSelectDropdown, TextRange.execCommand(), BC80
InsertTextArea, TextRange.execCommand(), BC80
InsertUnorderedList, TextRange.execCommand(), BC80
insideWindowWidth, DHTML cross-browser map puzzle, BC761
instanceof, object operator, 428
Instant SSL, BC597
integers. *See also* floating-point numbers; numbers
hexadecimal, 273–274
octal format, 273–274
intelligent
documents, 78
web pages, 78
interactive, readyState, 598
interCap format, 112, 125
interfaces
calculations, BC696–BC697
lookup tables, BC654
order form, BC669
internalSubset, document.doctype, 932
international characters, signed scripts, BC607
Internet Explorer (IE)
all, BC613
attachEvent(), BC626
behaviors, BC623–BC635
elements, BC627–BC631
embedding, BC624–BC625
enabling/disabling, BC625
event handlers, BC626
global variables, BC626
methods, BC627
properties, BC627
references, BC625
text rollover, BC631–BC635
compatibility, 17–22
cookies, BC714
createStyleSheet(), 960
deconstructing assignment, 355
detachEvent(), 1062
DOM, BC609, BC611
dominance, 16
elementFromPoint(), 964
error dialog box, 104
event, 1062
event binding, 1061–1062
event handlers, 174
events, properties, 1074–1097
Firefox, 17
global variables, 465
HTTPRequest, BC277
innerText, 585–586
insertAdjacentHTML(), BC309
JSObject, BC548
li, 219
<object>, BC396, BC528
object references, 90
plug-ins, BC396–BC399
<script>, 41
security, 17
select, 170
select(), BC662

style, BC613
style sheets, BC623–BC624
styleSheet, 960
type, 41
version 4, 22, 511–515
CSS and, 515
document.all, 49
event, 1074
event bubbling, 1047–1049
events, 1044, 1047–1052, 1071–1074
filter, BC259–BC262
iframe, 196
innerHTML, 975
outerHTML, 975
outerText, 975
revealClip(), BC416
select, BC180–BC186
setColor(), BC414
srcElement, BC414
tables, BC310
version 5, 19, 515–516
currentStyle, BC210, BC416, BC618
events, 1052
filter, BC265–BC271
hideFocus, 627
Mac OS X, BC309
W3C DOM, BC610
window, 835
window.event, 1060
window.open(), 812
XMLHttpRequest, BC687
Internet resources, BC818–BC822
Internet service providers (ISPs), BC105
CGIs, BC110
e-mail, BC110
intersectsNode(), range, BC43
invert(), BC260
Iris(), BC268
isChar, event, 1111–1112
isCollapsed, selection, BC51
isDefaultNamespace(), 663
isDisabled, 578
isDone(), DHTML cross-browser map puzzle, BC758
isEmpty(), BC496
isEqual(), TextRange, BC90
isEqualNode(), 663
isFinite(), 485
isInteger(), BC497
isMap, Image, 1013
isMultiLine, 578–579
isNaN(), 276, 281, 485–486
isNumber(), BC498, BC500, BC501
isOpen, popup, 877
isPointInRange(), range, BC43
isPosInteger(), BC497
isPrototypeOf(), Object, 477
ISPs. *See* Internet service providers
isSameNode(), 663
isSupported(), 525, 663–664
isTextEdit, 579
isTrusted, event, 1112
isValid(), BC500, BC501
lookup tables, BC660
Italic, TextRange.execCommand(), BC80
item(), 664–665
Enumerator, 498
select, BC197
iterators, array comprehensions, 354

J

JAR Packager, BC599
Java
applets, 10, BC524–BC563
data type conversion, BC542, BC545
doNewWindow(), BC547
eval(), BC547
faceless, BC537–BC542
fetchText(), BC539
getFile(), BC539
getMember(), BC547
HTML, BC543
init(), BC547
limitations, BC536–BC537
mainwin, BC547
object checks, BC595
permissions, BC543
privacy, BC592
properties, BC528–BC529
source code, BC538–BC539
window, BC543
XHTML, BC527–BC528
classes, BC525
direct scripting, BC562–BC563
NPAPI, BC562
packages, BC562
classpath, BC542–BC543
methods, BC525–BC528
mainString, BC563
<object>, BC526
objects, BC525
plug-ins, BC524–BC563
ActiveX, BC550
API, BC551–BC558
HTML, BC550–BC551
library, BC551–BC552
sandbox, BC591–BC592
String, BC529
void, BC526
javaEnabled(), navigator, BC381
java.lang, String, BC563
JavaPluginCocoa.bundle, BC542
JavaScript
animation, 48
arguments, 56
built- objects, 55
case sensitivity of, 86
control structures, 56
cookies, 922
core
document objects, separation of, 17–18
standard, 18–19
database, BC652–BC653
sort, BC771–BC777
ECMA-262 standard, 109, 179
error trapping, 57
expressions, embedding, 366
functions, values, 56–57
history, 10–12
host environment, 54
HTML, 37–51

JavaScript (*continued*)
- inheritance, 56
- limitations, 13
- loosely typed language, 55
- objects, quick reference, 1125–1132
- OO, objects, 458–470
- operators, 56
- prototype, 56
- reserved words, BC800
- separated development layers and, 24–25
- statements, 99–100
- static objects, 56
- updated flags, BC710
- user agent, 42–46
- uses, 12–13
- variables, 56
- versions, 11, 18–19, 39–40
- W3C DOM, 179
- web-authoring technologies, 3–9

javascript: pseudo-URL, 219, BC582–BC583, BC727
- calendars, BC651
- href, 447, BC679

JavaScript Object Notation (JSON), 357–361
- {} (curly brackets), 357
- [] (square brackets), 357
- data types, 358–359
- ECMAScript, 359
- eval(), 359, 361
- object, 361
- parse(), 360
- replacer, 360
- security, 361
- sending/receiving data, 359–360
- space, 360
- stringify(), 360
- toJSON(), 360
- XMLHttpRequest(), 361

javascript.enabled, BC382

javascript:functionName(), 997
- target, 1001

join(), arrays, 323, 337–339

join, id, 155

join(), strings, 227

.js
- API, BC618
- arrays, BC663
- DHTML cross-browser map puzzle, BC752
- .htc, BC625
- http://, 38
- libraries, 38–39
- MIME, 39

jsb-global.js, 508
- addEvent(), 138, 162
- Content/, 138

JScript, 11–12, 109

JSObject, BC543–BC545
- IE, BC548
- netscape.javascript, BC546
- subwin, BC547

JSObject.getWindow(), BC543

JSON. *See* JavaScript Object Notation

JSON, 55

jukebox, BC554–BC558

JustifyFull, document.execCommand(), 968

JustifyLeft, document.execCommand(), 968

JustifyParagraph, document.execCommand(), 968

JustifyRight, document.execCommand(), 968

K

Key(), Enumerator, 498

keyboard
- character codes, 719
- trapping, data-entry validation, BC493–BC494

KeyboardEvent, Mozilla, 959

keyCode
- event, 721, 1088–1091, 1101–1104
- events, 1069

KeyEvents, Mozilla, 959

Keys(), Dictionary, 498

keywords
- const, 493–498
- objects, 460
- this, 170
- variable names and, 112

KHTML, 11

L

LABEL, identify(), 170

label, BC14–BC16
- form, BC125–BC126
- options, BC201

labeled statements, 393–396

labeloptgroup, BC202–BC204

lambda expressions, 439

lang, 579–580

Language, 562

language, 40–41, 580
- navigator, BC375
- <script>, 96–99

lastChild, 522, 570–572
- appendChild(), 620
- W3C DOM, 647

lastChild(), TreeWalker, 992

lastIndex
- regular expressions, BC472, BC473
- RegularExpressionObject, BC484–BC485

lastIndexOf(), arrays, 323, 335–337

lastMatch, RegExp, BC489

lastModified, document, 943–944

lastPage(), table, BC336

lastParen, RegExp, BC489

lastVisit, BC713
- soft cookies, BC710

layer
- captureEvents(), 1046
- releaseEvents(), 1046

layers, BC411–BC412
- clipping, BC416–BC423
- development, 24–25
- dragging, BC439–BC447
- frameborder, BC432

HTML, BC432–BC434
iframe, BC432
nesting, BC424–BC431
resizing, BC439–BC447
stacking order, BC436–BC439
visibility, BC434–BC436
zIndex, BC436–BC439
<layer>, Navigator 4, 52
layerelement, NN4, BC411
layer.moveBy()
Navigator 4, BC620
shiftBy(), BC620
layerObject.above, BC436
layerObject.below, BC436
layers[], document, 944
layers, Navigator 4, 49, BC615
layerX, event, 1104–1108
layerY, event, 1104–1108
Layout:Complete, 812
layoutGrid, BC241
layoutGridChar, BC242
layoutGridLine, BC242
layoutGridMode, BC242
layoutGridType, BC242
leading minus sign, test for, BC497
leaf node, 87
left, BC246, BC622
iframe.align, 871
style, BC617
TextRectangle, BC100–BC102
window.open(), 806
leftContext
RegExp, BC489
regular expressions, BC474
leftMargin, body, 984–985
legend, BC103
form, BC123–BC124
form controls, BC122–BC124
length, 231–232, 580–581
arrays, 129, 133, 323–324
firstChild, 570
form, BC118
functions, 476
plugin, BC390–BC391
radio, BC147
select, BC188
String, 116, 275
letterSpacing, BC234
lhs, frame, BC327
li, BC356–BC357
appendChild(), 620
HTML, 216
IE, 219
ol, BC352
libraries
functions, 454–456
getObject(), BC620
Java plug-ins, BC551–BC552
.js, 38–39
object checks, BC595
validation functions, BC495–BC501
Light(), BC268–BCC269
light(), BC260
line terminators, statement delimiters, 57
lineBreak, BC234
lineCap, canvas, 1037
lineHeight, BC234–BC235
lineJoin, canvas, 1037
lineNumber, error.Object, 409
lineTo(), canvas, 1041
lineWidth, canvas, 1037
link, BC293–BC297
body, 983
event handlers, BC297
head, 73
onload, BC297
properties, BC294–BC297
<link>, BC208
linkColor, document, 912–915
links
objects, 995–1002
onclick, 997
links[], document, 944–945
list, form, 155
list objects, tables, BC303–BC359
listStyle, BC253
listStyleImage, BC253
listStylePosition, BC253
listStyleType, BC253
setCapture(), 669
literal notation, 314
LiveScript, 10–11, BC695
LN2, Math, 277
LN10, Math, 277
load, window, 141–142
loadCached(), imageLibrary, 210
loadDecisionName, cookies, BC728
loaded, readyState, 598
loading, readyState, 598
loadOuter(), BC432
loadXML(), XMLHttpRequest, BC280
loadXMLDoc()
init(), BC686
makeHTML(), BC688
XMLHttpRequest, BC769
local scope, 127
variables, 56
local variables, 127, 448–453
localeCompare(), 241
localName, 522, 581–582
location, 82, 142–143, 881–900
document, 945–948
https:, BC593
methods, 881
object checks, BC594
properties, 884–897
references, 883
same origin security policy, BC593
search, 999
security restrictions, 881–883
window, 768
window.open(), 806
locationbar, window, 758–760
location.hash, BC725
location.href, 143, 893, 997
calendars, BC641
select, 168
location.protocol, 893
location.reload(), 898
location.search, 893
log(), 278
LOG2E, Math, 277
LOG10E, Math, 277
longDesc
frame, 858
iframe, 873
Image, 1013–1014

lookup tables, BC652–BC664
  database, BC652–BC653
  forms, BC655–BC662
  interface, BC654
  isValid(), BC660
  onsubmit, BC656
  parallel arrays, BC654
  parseInt(), BC660
  script, BC655
  search(), BC661
lookupNamespaceURI(), 665
lookupPrefix(), 665
loop
  Image, 1014
  marquee, BC20
loops, 124. *See also* for; while
  arrays, 318–320
  counter, i, 385–386, 452
  debugging, BC587–BC588
  decisions and, 121–122
  document, 937–938
  do-while, 390
  for-in, 390–392
  initial expression, 124
  properties, 390–392
  submit, 175
loosely typed language, JavaScript, 55, 225
lowsrc
  Image, 1014
  images, BC595
lowSrc, Image, 1014

M

Mac OS X
  IE5, BC309
  web-authoring environment, 27, 30
macro target, BC601
mailto, 893
mailto:URL, BC111
  method, BC118
mainString, 245, BC563
  Java methods, BC563
  regular expressions, BC475
mainwin, Java applets, BC547
makeHot(), onmouseover, BC631
makeHTML()
  body, BC687
  calcBlockState(), BC690, BC691
  div, BC691
  loadXMLDoc(), BC688
makeNewWindow(), 138, 629
  onload, 150
  subWrite(), 150
makeNormal(), onmouseout, BC631
makeOneRow(), BC671
makeTitleRow(), BC671
  for, BC670
manifest, BC599
map, 1028–1032
  properties, 1029–1032
  syntax, 1028
<map>, <img>, 1025
map()
  array comprehensions, 353
  arrays, 323
  Firefox, 340
margin, BC250–BC251
marginBottom, BC251
marginHeight
  frame, 858–859
  iframe, 873
marginLeft, BC251
marginRight, BC251
marginTop, BC251
marginWidth
  frame, 858–859
  iframe, 873
markerOffset, BC242
marks, BC242
marquee, BC16–BC21
  event handlers, BC21–BC22
  methods, BC21
  properties, BC17–BC21
  syntax, BC16
mashups, BC783
  Google Maps, BC788–BC799
mask(), BC260
MaskFilter(), BC269
master validation function, BC500–BC501
matchArray, regular expressions, BC478–BC479
matching tags, XML, 363
Math, 55, 183–184, 276–280
  methods, 183–184, 278
    shortcut, 279–280
  properties, 183–184, 277
max(), 184, 278
maxHeight, BC243
maximize(), 508
maxLength
  form controls, BC113
  text, BC159
maxValue, 124
MAX_VALUE, 281
maxWidth, BC243
mayscript, BC543
  <object>, BC548
media
  document, 948
  link, BC295
  style, BC211–BC212
  styleSheet, BC216
memory
  host environment, 57
  Image, 1006
  leaks, closures, 458
menu, BC359
menubar
  window, 758–760
  window.open(), 806
mergeAttributes(), 665–667
message, error.Object, 409
meta, 81, BC297–BC299
  properties, BC298–BC299
  syntax, BC297
metacharacters, regular expressions, BC468–BC470
metadata, BC297
metaKey, event, 1099–1100
method
  form, 157
  get, BC118
  HTML, BC118
  mailto:URL, BC118
  post, BC111, BC118
methods, 90
  () (parentheses), 91, 507
  arrays, 326–355
  body, 987–989
  Boolean, 284

button, BC132–BC133
canvas, 1039–1042
checkbox, BC140
custom objects, 460–462
Date, 186, 289–292
dateObj, 289–290
document, 954–980
E4X, 370–372, BC288
event, 1065–1066, 1119–1121
form, BC119–BC121
functions, 446–447
generic HTML element objects, 612–691
history, 904–906
IE behaviors, BC627
invoking, 91
Java, BC525–BC528
  mainString, BC563
Java plug-ins API, BC553–BC554
location, 881
marquee, BC21
Math, 183–184, 278
navigator, BC381–BC384
Number, 282–284
objects, 56
  object detection, 50
  this, 56
parameters, 91
parsing, strings, 234–261
popup, 877–880
radio, BC148–BC149
range, BC30–BC49
RegularExpressionObject, BC485–BC487
select, BC196–BC197
selection, BC53–BC56
static objects, 56
strings, 181–183
styleSheet, BC220–BC222
table, BC335–BC337
tables, BC777
Text, BC58–BC64
text, BC164–BC167
textarea, BC176
TextNode, BC58–BC64
TextRange, BC72–BC99
toString(), 487
  tr, BC347
  TreeWalker, 992–993
  userProfile, BC407–BC410
  window, 789–848
  XMLHttpRequest, BC284–BC286
method.extractContents(), range, BC40
Methods, 1000
Meyer, Eric, 8
Microsoft. *See also* Internet Explorer
  Script Debugger, BC579
  trusted ActiveX, 54
  Word, 28
middle, iframe.align, 871
milliseconds, 289, 296
  GMT, BC713
MIME. *See* Multipurpose Internet Mail Extension
mime, verifying, BC393–BC395
mimeType, 1000
  document, 948–949
  Image, 1015
  navigator, BC385–BC388
  type, 1002
  using, BC392
mimeTypes, navigator, BC375–BC376
min(), 278
minHeight, BC243
minimizable, window.open(), 807
minus sign, test for, BC497
MIN_VALUE, 281
minWidth, BC243
miterLimit, canvas, 1038
mobile devices, 79
modal, window.open(), 807
modal dialog, 34
modifier keys, events, 1047, 1066–1068
modules, 24
month lengths test, BC519
mothers, daughters, BC677
MotionBlur(), BC269
mouse. *See also specific commands*
  button, events, 1068–1069
  event bubbling, BC439
  rollovers, 19
mousedown, layers, BC439–BC440
MouseEvent, Mozilla, 959
MouseEvents, Mozilla, 959
mousemove, layers, BC439
mouseover
  palette, BC414
  status, 788
mouseup, layers, BC439–BC440
move(), TextRange, BC90–BC92
moveBy(), window, 801–805
moveEnd(), TextRange, BC92–BC93
moveFirst(), Enumerator, 498
moveHelp(), DHTML cross-browser map puzzle, BC759
moveNext(), Enumerator, 498
moveRow(), table, BC336–BC337
moveStart(), TextRange, BC92–BC93
moveTo()
  canvas, 1041
  Navigator 4, BC617
  window, 801–805
moveToBookmark(), TextRange, BC93
moveToElementText(), TextRange, BC93–BC94
moveToPoint(), TextRange, BC94–BC95
mozBorderRadius, BC251
Mozilla, 16. *See also* Firefox; Netscape
  Code Base Principles, 58
  cookies.txt, 921
  event, 1097–1121
    properties, 1099–1119
    syntax, 1098
  events, 959
  HTTPRequest, BC277
  iframe, 196
  Privilege Manager, BC598
  privileges, BC600–BC606
  signed scripts, 54
  SignTool, BC599
  type, 40

Mozilla, *(continued)*
Venkman, BC580
W3C DOM, 196
window.open(), 808–811
Mozilla Foundation, 15, 19
mozOpacity, BC243
MSObject, data type conversion, BC545
MSXML. *See* XML Core Services
Msxml2.XMLHTTP, BC687
multidimensional arrays, 320–321
multiline
RegExp, BC488–BC489
RegularExpressionObject, BC485
multiple, select, 655, BC179, BC189
multiple sounds, BC558–BC562
multiple windows
references, 202–206
scripts, 191–206
multiplier, calculations, BC698
Multipurpose Internet Mail Extension (MIME)
E4X, BC287
enctype, BC118
.js, 39
<script>, 39
Type, 1000
type, 1002
MutationEvents, Mozilla, 959
MutationNameEvent, Mozilla, 959
myAge, 111–112, 123

N

\n, 392
new line inline character, 228
textarea, BC175
{n,}, regular expressions metacharacter, BC469
{n}, regular expressions metacharacter, BC469
name, 89, 1000
applet, BC453
button, BC131–BC132
document.doctype, 932
document.forms[], 145
embed, BC463
error.Object, 409–410
form, 157, BC119
frame, 859
getErrorObj(), 406
iframe, 873
Image, 1015
map, 1032
meta, BC299
object, BC459
oXML.first, 365
plugin, BC390–BC391
radio, BC148
text, BC159–BC160
W3C DOM, 647
window, 769
names
cookies, 923
filenames, error messages, BC567
functions, 125
object, BC208–BC209
variables, 111–112
identifiers, 495–496
name(), XML, 371
namedItem(), select, BC197
nameProp, 1001
document, 949
Image, 1015–1016
Namespace, 55
namespaces[], document, 949
namespaceURI, 522, 581–582
NaN (Not a Number), 274–275, 281–282. *See also* isNaN()
naturalHeight, Image, 1016
naturalWidth, Image, 1016
navigate(), window, 805
NavigateAndFind(), window.external, 762
navigation bar, Decision Helper, BC716
Navigator. *See* Netscape Navigator
navigator, 82, 143, BC361–Bc384
browserLanguage, BC373
methods, BC381–BC384
mimeType, BC385–BC388
plugin, BC389–BC391
properties, BC362–BC381
screen, BC400–BC404
syntax, BC361
userAgent, BC367
userProfile, BC380–BC381, BC404–BC410
window, 769
navigator.appName, 114
navigator.appname, Evaluator, BC612
navigator.appVersion, 487
navigator.userAgent, 143
 , select, BC703
NEGATIVE_INFINITY, 280–281
nesting
functions, 440
if...else, 376–377
layers, BC424–BC431
objects, prototype inheritance, 472–474
Netscape. *See also* Navigator
digital certificates, BC597–BC598
document.layers, 49
same origin security policy, BC593–BC595
security policies, BC592–BC597
signed script policy, BC595–BC600
signed scripts, security, BC590–BC607
targets, BC601–BC603
window.open(), 808–811
Netscape, appName, BC612
netscape, window, 769–770
Netscape Navigator (NN), BC360–BC410
JavaScript versions, 40
version 2, 303
version 3, select, BC180–BC185
version 4, 19, 511–512
arrays, 49
events, 1045–1047
<layer>, 52
layerelement, BC411
layer.moveBy(), BC620
layers, 49, BC615
moveTo(), BC617

version 6, 11, 19
event, 1097–1121
events, 1071–1074
tables, BC310
Netscape Plugin Application Programming Interface (NPAPI), BC524–BC525
browsers, BC542
Java classes, BC562
xpconnect, 750
netscape.javascript, JSObject, BC546
netscape.security, BC600
network.cookie .cookieBehavior, BC382
network.cookie .warnAboutCookies, BC382
new, 129, 180
object operator, 429
operator precedence, 434
new Date(), 288
newAsOf()
document.write(), BC709
time stamp, BC713
newsgroups, BC821–BC822
newWindow, 138
nextNode(), TreeWalker, 992–993
nextPage, event, 1091–1092
nextPage(), table, BC337
nextPrevVisit, BC711
nextSibling, 522, 582–583
W3C DOM, 647
nextSibling(), TreeWalker, 992
Nielsen, Jakob, 192
{n,m}, regular expressions metacharacter, BC469
NN. *See* Netscape Navigator
Node, properties, 522–23
nodes, 87–88. *See also specific nodes*
definition of, 520
DOM, 263
XML, BC273–BC274
node trees, 88
nodeName, 522, 582–583
document.doctype, 932
W3C DOM, 647
nodeType, 522, 583–584
childNodes, 550
document.doctype, 932
W3C DOM, 647
nodeValue, 522, 585–586
W3C DOM, 647
noHref, area, 1028
NO_MODIFICATION_ALLOWED_ERR, 400
non-DHTML browsers, BC619
none, rules, BC331
noResize
frame, 859
iframe, 873
normalize(), 667
noscript, 44–46
HTML, 44
script, 44
noShade, hr, BC13
Not a Number. *See* NaN
notations, document.doctype, 932
nowGMT, BC711
noWrap
body, 985
td, BC351–BC352
NPAPI. *See* Netscape Plugin Application Programming Interface
Null, data type, 110
null
conditional expressions, 374
cookies, BC723
JSON, 359
TextRange, 595
window.prompt(), 141
null/empty entry, test for, BC496
Number, data type, 110
Number, 55, 280–284
methods, 282–284
new, 429
properties, 281–282
number, error.Object, 410
Number(), 486–487
numbers, 269–286. *See also* floating-point numbers; integers; NaN
concatenation, 414
conversion
numbers to strings, 116, 275–276
strings to numbers, 115–116, 274–275
JSON, 358
number-formatting routine, 271–272
positive/negative, test for, BC498
random, generation of, 184, 279
strings, 413–415
numberState[], BC663

O

Object, 55, 474–479
constructor, 476–477
new, 429
syntax, 474
object, BC453–BC460
ActiveX control, BC550
applet, BC453
classid, BC396
constructor, 228
JSON, 361
object, BC459
properties, BC455–BC460
switch, 380
syntax, BC454
VBScript, BC399
<object>
<applet>, BC550
browsers, BC528
<embed>, BC396
faceless applets, BC537
IE, BC396, BC528
Java, BC526
mayscript, BC548
security, BC543
objects. *See also specific objects*
abstract, 88
anchors, 995–1002
arrays, 311–355, 465–466
automation, 496
body, 907–993
body text, BC2–BC102
core language objects, 179–188

O

objects. *See also specific objects (continued)*
- custom
  - functions, 437–479
  - methods, 460–462
  - properties, 460
- data types, 110
- document, 907–993
  - arrays, 133
  - core JavaScript, separation of, 17–18
  - definition, 18
  - hierarchy, 503–505
- embedded, 467–468, BC448–BC464
- encapsulation, 465
- error, 407–410
  - syntax, 408
- exception handling, 404–405
- `form`, 155–157
- form controls, 158–170
- forms, BC103–BC126
- generic HTML element, 537–738
- getters, 468–470
- images, 207–216
- Java, BC525
- JavaScript, quick reference, 1125–1132
- JavaScript OOP, 458–470
- JSON, 359
- keywords, 460
- links, 995–1002
- list, tables, BC303–BC359
- methods, 56
  - object detection, 50
  - `this`, 56
- names, BC208–BC209
- nesting, prototype inheritance, 472–474
- operators, 412, 425–430
- positioned, BC411–BC447
- properties, 56, 460
  - binding events, 1061
- prototype, 56
- setters, 468–470
- static, 184
  - defined, 184
  - `Event`, 1045, 1098
  - JavaScript, 56
  - methods, 56
  - properties, 56
  - scripts, 56
- `String`, 228–229
- style, BC207–BC271
- table, W3C DOM, BC319–BC320
- top-level, 135–152
- XML, 364–370
  - serialization, 370

object checks, images, BC595

object detection
- browsers
  - compatibility, 49–51, BC615–BC617
  - versions, 51
- conditional expressions, 51
- `images`, 49
- object methods, 50

"Object doesn't support this property or method," error message, BC571

"Object expected," error message, BC569

object IDs. *See* identifiers

object literal notation, 474

object models. *See* document object model; W3C DOM

object referencing. *See* references

object-oriented programming (OO)
- `Function`, 437
- JavaScript, 55
  - objects, 458–470
- prototype, 470–474
- W3C DOM and, 523–532

`object.property`, BC287
- XML elements, 364

`object[property]`, XML elements, 364

octal (base-8) format, 273–274

`offscreenBuffering`, `window`, 770

`offsetHeight`, 572, 586–587

`offsetLeft`, 558, 587–588, BC424

`offsetParent`, 588–590

`offsetTop`, 558, 587–588, BC424
- `div`, 558

`offsetWidth`, 572, 586–587

`offsetX`, `event`, 1078–1084

`offsetY`, `event`, 1078–1084

`oForm.elements[i]`, 158

`oForm.elements.length`, 158

`oInput`, `converter`, 162

`OKToTest`, BC552

`ol`, BC352–BC355
- `replace()`, 681
- syntax, BC352

`onabort`, `Image`, 1022–1023

`onactivate`, 691–692

`onafterprint`, 815
- `body`, 989
- `window`, 848–849

`onafterupdate`, 692
- `text`, BC167

`onbeforecopy`, 692–693
- `onbeforecut`, 701

`onbeforecut`, 693–694
- `onbeforecopy`, 701

`onbeforedeactivate`, 694

`onbeforeeditfocus`, 694–695

`onbeforepaste`, 695

`onbeforeprint`, 815
- `body`, 989
- `window`, 848–849

`onbeforeunload`, `window`, 849–850

`onbeforeupdate`, 692
- `text`, BC167

`onblur`, 695–697
- `text`, BC168–BC170

`onbounce`, `marquee`, BC21–BC22

`oncellchange`, 697

`onchange`
- `calculate()`, BC671
- calculations, BC696
- calendars, BC641
- `click`, 173
- cookies, BC729
- data-entry validation and, BC493-BC494, BC509
- `search()`, BC661
- `select`, 168, BC180, BC198–BC199, BC641

text, BC170–BC171
upperMe(), 162
onclick, 219, 697–700
button, BC133–BC134
checkbox, 164
checkbox, BC140–BC143
click(), 631
clickevent, 1043
fullName(), 165
links, 997
onmousedown, 725
onmouseup, 725
processData(), BC107
radio, BC149–BC150
return false, 998
search(), BC661
setCapture(), 669
submit, 175
toggle(), BC691
oncontextmenu, 700
setCapture(), 669
oncontrolselect, 700
oncopy, 700–703
oncut, 700–703
ondataavailable, 703
ondatasetchanged, 703
ondatasetcomplete, 703
ondblclick, 703–704
onclick, 697
ondeactivate, 691–692
onblur, 696
ondrag, 704–710
ondragend, 704–710, 710
ondragenter, 710–712
ondragleave, 710–712
ondragover, 710–712
ondragstart, 704–710, 712
ondrop, 712
One True Brace Style (1TBS), 377
1TBS. *See* One True Brace Style
oneDate, 288
onerror, 397
Image, 1022–1023
window, 770–773, 850
onerrorupdate, 712
text, BC167
onevent, ; (semicolon), BC626
onfilterchange, 712–714
onfinish, marquee, BC22
onfocus, 714–715
text, BC168–BC170
onfocusin, 715
onfocusout, 715
onhelp, 715–717
window, 850–851
onkeydown, 717–719
events, 1069
onkeypress, 717–719, 841
checkForEnter(), 720
onkeyup, 717–719, BC494
events, 1069
onlayoutcomplete, 724
onLine, navigator, BC376
online documentation, BC819
onload
addEvent(), 1064
addEvent, BC697
body, BC551
calendars, BC644
cookies, BC706
faceless applets, BC537
framesets, 744
Image, 1023–1024
init(), BC630, BC678–BC679, BC769
link, BC297
makeNewWindow(), 150
setExpression(), 686
target, 1002
testValues(), 451
window, 68, 851–852, BC729, BC733
window.open(), 808
onlosecapture, 669, 724–725
onmousedown, 725–727
button, BC134
onclick, 697
onmouseenter, 727
onmouseleave, 727
onmousemove, 727
onmouseout, 213–215, 728–729
imageOff(), 213
makeNormal(), BC631
return, 215
setCapture(), 669
onmouseover, 728–729
document, 964–965
imageOn(), 213
makeHot(), BC631
return, 215
setCapture(), 669
onmouseup, 725–727
button, BC134
onclick, 745
selection, BC50
onmousewheel, 729
onmove, 729–730
onmoveend, 730
onmovestart, 730
oNow, 69
onpaste, 730–733
onpropertychange, 733–735
propertyName, 1092
onreadystatechange, 735, BC278
onreset, form, BC121–BC123
onresize, 735–736
window, 852
onresizeend, 736
onresizestart, 736
onrowenter, 736
onrowexit, 736
onrowsdelete, 736
onrowsinserted, 736
onscroll, 736
body, 989
window, 852–853
onselect, text, BC168–BC170
onselectionchange, document, 980
onselectstart, 736–738
onstart, marquee, BC22
onstop, document, 980–981
onsubmit, 160, BC493-BC494, BC521
checkForm(), 175
form, 1050, BC111, BC122
lookup tables, BC656
submit(), 174
onTarget(), DHTML cross-browser map puzzle, BC756
onunload
<body>, 1018
resetSelects, 1018
restore(), 759

onunload *(continued)*
window, 853–854
window.open(), 808
OO. *See* object-oriented programming
opacity, BC243
open()
document, 971–972
window, 805–814
XMLHttpRequest, BC285–BC286
openDialog(), window, 814–816
opener, window, 202, 774–777
Opera, 16
.all, BC615
debugging, BC580–BC581
deconstructing assignment, 355
Dragonfly, BC580–BC581
E4X, BC272
error messages, BC566
HTTPRequest, BC277
object literal notation, 474
W3C, BC615
operator precedence, 434
operators, 116–117, 411–435
arithmetic, 117
assignment, 412, 418–420
bitwise, 412, 424–425
categories, 411–412
comparison, 117, 412–413
if, 374
connubial, 412, 415–418
JavaScript, 56
object, 412, 425–430
precedence, 432–435
custom validation function, BC499
OPML. *See* Outline Processor Markup Language
optgroup, BC201–BC204
option, BC199–BC201
select, BC701
<option>
select, 167
selected, BC180
switch, 381
options[], select, BC189
options.add(), select, BC197
options[].defaultSelected, select, BC189–BC190
options[].index, select, BC190
options.remove(), select, BC197
options[].selected, select, BC190–BC191
options[].text, select, BC191–BC192
options[].value, select, BC193–BC194
OR
bitwise operator, 425
Boolean operator, 421
order forms, BC665–BC673
HTML, BC667–BC673
interface, BC669
scripts, BC667–BC673
String.split(), BC666
origin checks, BC594–BC595
originalTarget, event, 1112
orphans, BC255
oscpu, navigator, BC376–BC377
oTarget.firstChild, while, 71
Outdent, document.execCommand(), 968
outerHeight
window, 766–768
window.open(), 807
outerHTML, 518, 590–592
IE4, 975
outerText, 513, 590–592
IE4, 975
outerWidth
window, 766–768
window.open(), 807
outline, BC251
OPML, BC690
Outline Processor Markup Language (OPML), BC682–BC684
body, BC687
head, BC690
outline, BC690
parent-child relationship, BC684
outlineColor, BC251
outlineOffset, BC252
outlineStyle, BC252
outlineWidth, BC252
output, 65
getElementById(), 69
#output, 74
output span, 69
output streams, document.write(), 148
outside, componentFromPoint(), 634
overflow, BC243
overflow: auto, CSS, 192
overflowX, BC244
overflowY, BC244
OverWrite, TextRange.execCommand(), BC80
ownerDocument, 522, 593
W3C DOM, 647
ownerNode, styleSheet, BC216–BC217
ownerRule, styleSheet, BC217
owningElement, styleSheet, BC217
oXML.first, name, 365

P

p, paragraph element, 65–66, 73
<p>, tabindex, 715
packages, Java classes, BC562
padding, BC252–BC253
page, BC255
pageBreakAfter, BC255–BC256
pageBreakBefore, BC255–BC256
pageBreakInside, BC256
pages, styleSheet, BC218
pageX, event, 1104–1108
pageXOffset, window, 777
pageY, event, 1104–1108
pageYOffset, window, 777
palette, mouseover, BC414

parallel arrays, 131–133, 315–320
lookup tables, BC654
param, BC463–BC464
parameters
arguments, BC677
cookies, 925
form, BC106–BC107
functions, 125–126, 441
methods, 91
references, BC111
values, BC111
variables, 453–454
parent
Decision Helper, BC716
top, 741, 743
window, 778–780
parents
frames, 191–194
nodes, 87–88
_parent, <form>, BC119
parent-child relationship
OPML, BC684
XML, BC684
parentElement, 593–594
parent.frames[0].location, 883
parent.frames[1].location, 883
parent.frames[i], BC578
parent.location, 883
parentNode, 522, 594–595
frame, 746
W3C DOM, 647
parentNode(), TreeWalker, 992
parent.otherFrameName .location, 883
parentStyle
cssRule, BC224
styleSheet, BC209
parentStyleSheet, styleSheet, BC218
parentTextEdit, 595–597
parent-to-child, frames, 194, 742
parentWindow, document, 949
parse(), JSON, 360
parseFloat(), 116, 275, 486–487
floating-point numbers, BC669
parseInt(), 116, 275, 486–487, BC499, BC502, BC515, BC622
height, 572
lookup tables, BC660
parsing methods, strings, 234–261
password, BC171–BC172
passwords, throwaway, 922
Paste, TextRange.execCommand(), BC80
pasteHTML(), TextRange, BC96–BC97
path, cookies, 924
pathname, 892, 999
area, 1028
Perl, 77
permissions, Java applets, BC543
personalBar, window.open(), 807
personalbar, window, 758–760
phantom page syndrome, 514
PHP, 77
PI (Math.PI), 277
"Picking a Rendering Mode" (Meyer), 8
pixels
height, 208
width, 208
pixelBottom, BC247
pixelDepth, screen, BC403–BC404
pixelHeight, BC247
pixelLeft, BC247
pixelRight, BC247
pixelTop, BC247
pixelWidth, BC247
placeholder, span, 221
placeholder document, 889
placeHolderElement, 451
planet web page exercise, 133–134
platform, navigator, BC377–BC378
platform equivalency, browser compatibility, BC613–BC614
PlayImage, TextRange.execCommand(), BC80
plugin
navigator, BC389–BC391
using, BC392
pluginIsReady(), BC394
-plugin.jar, BC542
plugins, navigator, BC378
plug-ins, 9
IE, BC396–BC399
installation, BC395–BC396
Java, BC549–BC563
ActiveX, BC550
API, BC551–BC558
HTML, BC550–BC551
library, BC551–BC552
verifying, BC393–BC395
versions, BC398–BC399
plugins[], document, 949
pluginspage, embed, BC463
poly
coords, 999
shape, 1027
polygon, shape, 1027
pop(), arrays, 323, 340–341
popup, 875–880
methods, 877–880
properties, 876–877
syntax, 875
pop-up windows, BC545
calendars, BC651
CSS, BC693
port, 893, 999
area, 1028
posBottom, BC247
posHeight, BC247
position, BC247
positioned objects, BC411–BC447
position.top, 73
positive connubial operator, + *val*, 415, 416
positive integers, test for, BC497
POSITIVE_INFINITY, 280–281
posLeft, BC247
posRight, BC247
post, method, BC111, BC118
posTop, BC247
posWidth, BC247

pow(), 184, 278
pre-caching, images, 208–210
preference(), navigator, BC381–BC384
PreferencesRead, BC602
PreferencesWrite, BC602
prefix, 581–582
Prefix, 522
prefsDlog, 841
preserveIDs, mergeAttributes(), 665
preventDefault()
    dispatchEvent(), 637
    event, 1120–1121
    forms, 174
    W3C DOM, 1057
prevention, debugging, BC587–BC588
previousNode(), TreeWalker, 992–993
previousPage(), table, BC337
previousSibling, 522, 582–583
    W3C DOM, 647
previousSibling(), TreeWalker, 992
primitive target, BC601
print(), frames, 815
printEvalulator(), BC581
privacy
    Java applets, BC592
    submit(), BC121
Privilege Manager, Mozilla, BC598
PrivilegeManager, enablePrivilege(), BC606
privileges
    errors, try...catch, BC606
    Mozilla, BC600–BC606
    scripts, BC603
processData()
    forms, BC107
    onclick, BC107
    this, 170–173
processRequest(), XMLHttpRequest, BC279, BC280
product, navigator, BC379
productSub, navigator, BC379
profile, head, BC291
programming, *vs.* scripts, 105–106
progressive enhancement, 22–23, 68
prompt(), window, 816–818
prompter, window, 750
properties, 89–90, 1099–1119. *See also specific properties*
    applet, BC451–BC453
    area, 1027–1028
    arrays, 323–326
    body, 983–987
    Boolean, 284
    button, BC131–BC132
    canvas, 1035–1039
    checkbox, BC136–BC140
    clientInformation, BC362–BC381
    CSS, 220
    cssText, BC223–BC226
    custom objects, 460
    document, 912–954
    embed, BC462–BC463
    event, 1065–1066
        Mozilla, 1099–1119
        NN6, 1099–1119
    form, 157–158, BC114–BC119
    frame, 855–861
    frameset, 863–868
    functions, 441–445
    generic HTML element objects, 541–612
    history, 801–903
    hr, BC11–BC12
    IE
        behaviors, BC627
        events, 1074–1097
    Java applets, BC528–BC529
    link, BC294–BC297
    location, 884–897
    loops, 390–392
    map, 1029–1032
    marquee, BC17–BC21
    Math, 183–184, 277
    meta, BC298–BC299
    navigator, BC362–BC381
    Number, 281–282
    object, BC455–BC460
    objects, 56, 460
        binding events, 1061
        static, 56
    popup, 876–877
    radio, BC145–BC148
    range, BC26–BC29
    RegExp, BC488–BC490
    RegularExpressionObject, BC484–BC485
    script, BC300–BC301
    select, BC188–BC196
    selection, BC50–BC53
    static objects, 56
    strings, 231–261
    style, BC211–BC212
    styles, BC210
    styleSheet, BC213–BC220
    table, BC321–BC334
    Text, BC58
    text, BC157–BC164
    textarea, BC175–BC176
    TextNode, BC58
    TextRange, BC68–BC72
    title, BC302
    tr, BC345–BC346
    TreeWalker, 991–992
    window, 136–137, 750–789
    XMLHttpRequest, BC281–BC284
property picker, 886–889
propertyIsEnumerable(), Object, 477–478
propertyName, 321
    onpropertychange, 1092
protocol, 892–893, 999
    area, 1028
    document, 950
    Image, 1016–1017
prototype
    inheritance, 56, 472–474
    JavaScript, 56
    objects, 56
    OOP, 470–474
prototype
    Array, 324–326
    arrays, 324–326
    functions, 445
    Number, 280
    RegExp, BC490

String, 232–234
pseudo-URL. *See* javascript: pseudo-URL
public:, XML tags, BC627
public:attach, BC626
publicId, document.doctype, 932
push(), arrays, 323, 340–341

Q

q, BC3
QName, 55
quadraticCurveTo(), canvas, 1039
qualifier, event, 1076–1077
queryCommand
  document, 972–973
  TextRange, BC97
quick reference, JavaScript objects, 1125–1132
quirks mode, 21, 22, 534
  compatMode, 920
quotes, BC235

R

\r
  carriage return inline character, 228
  textarea, BC175
radio, BC143–BC150
  event handlers, BC149–BC150
  methods, BC148–BC149
  properties, BC145–BC148
  syntax, BC143–BC144
radio buttons, 165–167
  checked, BC112
radix, 274
random(), 279
random number generation, 184, 279
RandomBars(), BC270
RandomDissolve(), BC269–BC270
Range
  TextRange, BC68
  W3C DOM, BC2
range, BC22–BC49
  methods, BC30–BC49
  properties, BC26–BC29
  syntax, BC23
  W3C DOM, BC23
rangeCount, selection, BC51
RangeError, 55
readColor(), select, BC633
readOnly
  cssRule, BC224
  styleSheet, BC218–BC219
  text, BC160–BC161
readyState, 597–598, BC281
  addBehavior(), 613
real-time validation, BC492–BC494, BC504–BC505
reason, event, 1076–1077
recalc(), document, 973–974
recordNumber, 561, 598–601
recordset, event, 1076–1077
rect(), canvas, 1041
rect
  coords, 999
  shape, 1027
rectangle, shape, 1027
recursion, functions, 454
reduce(), arrays, 323, 341–344
reduceRight(), arrays, 323, 341–344
ReferenceError, 55
references
  events, 1059
  frame, 196
  frames, 194–195, 742
  IE behaviors, BC625
  location, 883
  multiple windows, 202–206
  objects, 85–86, 90
  parameters, BC109
  values, 56–57
referrer, document, 950–951
Refresh
  document.execCommand(), 969
  TextRange.execCommand(), BC80
refresh()
  plugin, BC391
  table, BC337
regedit, BC397
RegExp, 55, BC465–BC490
  new, 429
  properties, BC488–BC490
  syntax, BC487
Registry Editor, BC397
regular expressions, BC465–BC490
  \ (backslash), 245
  arrays, 352
  backreferencing, BC470–BC471
  grouping, BC470–BC471
  matches, BC476–BC480
  metacharacters, BC468–BC470
  patterns, BC465–BC467
  string.match(), 241–244
  string.replace(), 244–249
  strings, BC480–BC482
  string.search(), 249
  string.split(), 253
RegularExpressionObject, BC483–BC487
  methods, BC485–BC487
  properties, BC484–BC485
  syntax, BC483
rel, 1001
  link, BC295–BC296
relatedTarget, event, 1112–1114
release(), DHTML cross-browser map puzzle, BC756
releaseCapture, 667
releaseCapture(), 1052
releaseEvents()
  document, 974, 1046
  layer, 1046
  window, 1046
reload(), 897–898
remove(), select, BC196–BC197
Remove(), Enumerator, 498
RemoveAll(), Enumerator, 498
removeAllRanges(), selection, BC55
removeAttribute(), 672–673
removeBehavior(), 674, BC625

removeChild(), 71, 525, 675, BC310
  appendChild(), 620
removeEventListener(), 531, 615–620
  event binding, 1062–1063
removeExpression(), 675–676
RemoveFormat, document.execCommand(), 969
removeMember(), JSObject, BC544
removeNode(), 676–677
  removeChild(), 675
removeRange(), selection, BC55
removeRule, styleSheet, BC220–BC221
repeat, event, 1092
repeat loops. *See* loops
repeatCustomer, BC712
repetition blocks, BC126
replace()
  location, 899–900
  ol, 681
  regular expressions, BC481, BC482
replaceAdjacentText(), 677–678
replaceChild(), 525, 678, BC307
  appendChild(), 620
  removeChild(), 675
replaceData(), Text, BC58–BC63
replaceNode(), 678–681
replacer, JSON, 360
reserved words, JavaScript, BC800
Reset(), 566
reset(), form, BC119–BC120
resetSelects, onunload, 1018
Resistor Calculator, BC696–BC697
resistor.html, BC697
resizable
  IE, 835
  window.open(), 807
resizeBy(), window, 818–821
resizeIt()
  layers, BC443–BC446
  style.height, BC443
  style.width, BC443
resizeTo(), window, 818–821
responseText, BC281–BC282
responseXML, BC282
restore()
  canvas, 1041
  onunload, 759
result, Date, 287
return
  for, BC734
  onmouseout, 215
  onmouseover, 215
  onreset, BC122
return false, onclick, 998
return true, setMsg(), 215
returnValue, 531
  event, 1050, 1092–1093
  fireEvent(), 641
  setCapture(), 669
  window, 780
rev, 1001
  link, BC295–BC296
revealClip(), IE4, BC416
revealTrans(), BC261–BC262
reverse(), arrays, 323, 344–346
rhs, frame, BC327
right, BC246
  iframe.align, 871
  TextRectangle, BC100–BC102
rightContext, RegExp, BC489
rightMargin, body, 984–985
rollovers
  CSS, 216–219
  images, 211–219
  mouse, 19
  text, IE behaviors, BC631–BC635
root, TreeWalker, 991–992
rotate(), canvas, 1042
round(), 184, 278
routeEvent(), 1047
  document, 974
RowDelim, 562
rowIndex, tr, BC346
rows, tables, modifying, BC309–BC316
rows
  frameset, 864–867
  framesets, 744
  rules, BC331
  table, BC330–BC331
  textarea, BC175
rowSpan, td, BC350
RSS feeds, BC820–BC821
rubyAlign, BC235
rubyOverhang, BC235
rubyPosition, BC235
rule, BC208, BC223–BC226
rules
  styleSheet, BC219
  table, BC331–BC333
runtime errors
  DOM, 174
  syntax, BC564–BC565
runtimeStyle, BC226–BC257
  style sheets, 601

S

\S, regular expressions metacharacter, BC468
\s, regular expressions metacharacter, BC468
Safari, 16
  compatibility, 17–22
  debugging, BC580
  document.lastModified, 943
  Drosera, BC580
  E4X, BC272
  error messages, BC566
  HTTPRequest, BC277
  iframe, 196
  object literal notation, 474
  select, 170
safe arrays, Visual Basic, 498
same origin security policy, Netscape, BC593–BC595
sandbox, Java, BC591–BC592
save(), canvas, 1041
saveCurrentVisit, BC711
saveType, event, 1093
scale(), canvas, 1042
scope
  global, 127

host environment, 55
variables, 55
local, 127
variables, 56
variables, 127–128, 448–453
global, 55
local, 56
scopeName, 601–602
screen, 82
navigator, BC400–BC404
window, 780–781
screenLeft, window, 781
screenTop, window, 781
screenX
event, 1078–1084, 1104–1108
window, 781–782
window.open(), 807
screenY
event, 1078–1084, 1104–1108
window, 781–782
window.open(), 807
script, 34, BC299–BC301
addEvent(), 162
browsers, 95
DHTML cross-browser map puzzle, BC749
head, 67, 162
lookup tables, BC655
noscript, 44
properties, BC300–BC301
<script>, 37–41
for, 41
attributes, 95
<body>, 97
debugging, BC587
defer, 575
event, 41
functions, 101
<head>, 96
.htc, BC625
HTML, 40
IE, 41
inline branching, BC611
language, 96–99
MIME, 39
src, 38, BC595
tags, 43
type, 39, 40
scripts. *See also* JavaScript; signed scripts
browser compatibility, 47–51
comments
// (double slash), 100
/* (slash asterisk), 100
HTML, 43
XHTML, 43–44
deferred, 102–104, BC641
errors, viewing, 104–105
events, 1044
frames, 191–206
hiding, 46–47, 98
HTML documents, 95–107
image rollovers without, 216–219
multiple windows, 191–206
order forms, BC667–BC673
privileges, BC603
vs. programming, 105–106
run as another function, 104
scripting strategy, 22–25
statements, 100–104
called by another function, 104
functions, 101
run by user, 103–104
running after loading, 101–103
static objects, 56
style sheets, BC623–BC624
XHTML, 98–99
Script Debugger, Microsoft, BC579
script for, 41
ScriptableClock.java, BC529–BC536
scripting.com, BC682
scripts[], document, 951–952
<script><script>, 37
src, 38
scroll
body, 985–986
IE, 835
scroll(), 115
window, 821
scrollAmount, marquee, BC20–BC21
scrollbar, BC254
scrollbarDown, componentFromPoint(), 634
scrollbarHThumb, componentFromPoint(), 634
scrollbarLeft, componentFromPoint(), 634
scrollbarPageDown, componentFromPoint(), 634
scrollbarPageLeft, componentFromPoint(), 634
scrollbarPageRight, componentFromPoint(), 634
scrollbarPageUp, componentFromPoint(), 634
scrollbarRight, componentFromPoint(), 634
scrollbars
window, 758–760
window.open(), 807
scrollbarUp, componentFromPoint(), 634
scrollbarVThumb, componentFromPoint(), 634
scrollBy(), window, 822–824
scrollByLines(), window, 824–825
scrollByPages(), window, 824–825
scrollDelay, marquee, BC20–BC21
scrollHeight, 602
scrolling
frame, 859–861
iframe, 873–874
scrollIntoView(), 681–682, 852–853
focus(), 627

scrollLeft, 603–604, 852–853
  body, 986–987
scrollMaxX, window, 782
scrollMaxY, window, 782
scrollSpeed, 829
scrollTo(), window, 822–824
scrollTop, 603–604, 852–853
  body, 986–987
scrollWidth, 602
scrollX, window, 782–783
scrollY, window, 782–783
sDateTime, 69
search, 893, 999
  area, 1028
search()
  lookup tables, BC661
  onchange, BC661
  onclick, BC661
  regular expressions, BC476–BC477
search engines, 79
searchInFrames, window.find(), 798
sectionRowIndex, tr, BC346
SECURE, cookies, 924
security
  history, 905
  IE, 17
  JSON, 361
  location, 881, 883
  Netscape
    policies, BC592–BC597
    signed scripts, BC590–BC607
  <object>, BC543
  onsubmit, BC122
  same origin security policy, Netscape, BC593–BC595
  submit(), BC121
security, document, 952
security.enable_Java, BC382
securityPolicy, navigator, BC379–BC380
select, 158–159, 167–170, BC177–BC199, BC700–BC701
  addTotals(), BC670, BC673
  arrays, 322
  data-validation filter, BC495, BC501
  DOM, 167
  event handlers, BC198–BC199
  HTML, 167
  IE4, BC180–BC186
  images, 210
  indexes, 167
  label, BC125
  location.href, 168
  methods, BC196–BC197
  multiple, 655, BC179, BC189
   , BC703
  NN3, BC180–BC185
  onblur, 695
  onchange, 168, BC641
  <option>, 167
  properties, BC188–BC196
  readColor(), BC633
  selectedIndex, 153, 167–168, 316
  setAttribute(), BC11
  setSelected(), 837
  setTimeout(), BC662
  src, BC703
  syntax, BC178
  text, 168
  value, 170
  W3C DOM, BC187–BC188
select()
  focus(), 627, BC662
  IE, BC662
  text, BC165–BC167
  TextRange, BC97
SelectAll, document.execCommand(), 969
selectAllChildren(), selection, BC55
selected, <option>, BC180
selectedIndex
  checkTimer(), 1018
  select, 153, 167–168, 316, BC194–BC195
selectedObj, DHTML cross-browser map puzzle, BC753
selectedState, 132
selectedStateLabel, DHTML cross-browser map puzzle, BC754
selection, BC49–BC56
  document, 952–953
  methods, BC53–BC56
  onmouseup, BC50
  properties, BC50–BC53
  syntax, BC49
selectNode(), range, BC25, BC43–BC44
selectNodeContents(), range, BC25, BC43–BC44
selectorText, cssRule, BC224–BC225
self, window, 137, 191, 783
_self, <form>, BC119
self.blur(), 628
self.close(), 138
send(), XMLHttpRequest, BC286
separation of development layers, 38
serialization, XML objects, 370
serverless CGIs, 78, 312
server-side programming, 8
setActive(), 682–683
  focus(), 682
setAttribute(), 547
setAttribute(), 525, 683–684
  case sensitivity, 646, 683
  onpropertychange, 733
  select, BC11
  W3C DOM, 646
setAttributeNode(), 673–674, 684
setAttributeNodeNS(), 684
setAttributeNS(), 684
setBGColor(), backgroundColor, BC621
setCapture(), 668–672, 1052
setColor()
  IE4, BC414
  W3C DOM, BC414
setCookie(), BC720
setCookieData(), BC711
setData()
  dataTransfer, 1085
  window.clipboardData, 751

setDate(), 186, 299
  dateObj, 290
setDay(), 186
  dateObj, 291
setdh3Importance(), BC734
setEnd(), range, BC44–BC46
setEndAfter(), range, BC46–BC47
setEndBefore(), range, BC25, BC46–BC47
setEndPoint(), TextRange, BC97–BC99
setExpression(), 685–689
setExpression, style, 675
setFullYear(), dateObj, 290
setHotColor(), applyVals(), BC633
setHours(), dateObj, 291
setHRAttr(), value, BC11
setInitialColor(), addBehavior(), 613
setInnerPage(), BC424
setInterval()
  clearInterval(), 825
  DHTML cross-browser map puzzle, BC759-BC760
  Image, 1006
  setTimeout(), 825
  startScroll(), 829
  stepClip(), BC416
  window, 825–829
setMember(), JSObject, BC544
setMilliseconds(), dateObj, 291
setMinutes(), 186
  dateObj, 291
setMonth(), 186, 299
  dateObj, 290
setMsg(), return true, 215
setOuterPage(), BC424
setRequestHeader(), XMLHttpRequest, BC286
setSeconds(), dateObj, 291
setSelect, DHTML cross-browser map puzzle, BC753
setSelected(), select, 837
setSelectedMap(), DHTML cross-browser map puzzle, BC754
setSlot(), JSObject, BC544
setStart(), range, BC44–BC46
setStartAfter(), range, BC25, BC46–BC47
setStartBefore(), range, BC46–BC47
setTens(), select, BC701
setters, BC528–BC529
  objects, 468–470
setTime(), 186
  cookies, 923
  dateObj, 291
    date, 295
setTimeout(), 458, 821
  focus(), 627
  select, BC662
  setExpression(), 687
  setInterval(), 825
  window, 829–832
setUserData(), 689
setUTCDate(), dateObj, 291
setUTCDay(), dateObj, 291
setUTCFullYear(), dateObj, 291
setUTCHours(), dateObj, 291
setUTCMilliseconds(), dateObj, 291
setUTCMinutes(), dateObj, 291
setUTCMonth(), dateObj, 291
setUTCSeconds(), dateObj, 291
setWinWidth(), DHTML cross-browser map puzzle, BC761
setYear(), 186
  dateObj, 290
setZIndex(), BC621
SGML. *See* Standard Generalized Markup Language
Shadow(), BC270
shadow(), BC260
shadowBlur, canvas, 1038
shadowColor, canvas, 1038
shadowOffsetX, canvas, 1038
shadowOffsetY, canvas, 1038
shape, area, 1027–1028
sheet, link, BC296
shift(), arrays, 323, 340–341
shiftBy(), layer.moveBy(), BC620
shiftKey, event, 1074–1075, 1099–1100
shiftLeft, event, 1075
shiftTo(), DHTML, BC620
  cross-browser map puzzle, BC756, BC758
shortWidth, 686
show(), BC621
  popup, 877–880
showArrays(), 326
showContextMenu(), setCapture(), 669
showDialog, window.find(), 798
showGlobal(), 451–452
showHelp()
  DHTML cross-browser map puzzle, BC759, BC760
  window, 832–833
showModalDialog(), window, 833–847
showModelessDialog()
  window, 833–847
  window.dialogArguments, 843
showModelessWindow(), 841
showProps(), 392
showTip(), BC726
sidebar, window, 750
signed scripts
  export, BC606–BC607
  import, BC606–BC607
  international characters, BC607
  locking down, BC607
  Mozilla, 54
  Netscape, BC595–BC600
SignTool, Mozilla, BC599
sin(), 278
size, BC256
  columnWidths, BC673
  font, BC7–BC8
  hr, BC13–BC14
  select, BC195
  text, BC161
sizeToContent(), window, 847–848

slice(), arrays, 323, 346
soft, textarea, BC174
soft cookies, BC706
<img>, BC713
lastVisit, BC710
soft reload, 898
Soft Reload/Refresh button, 898
some(), arrays, 323, 331–333
Something(), evt, 1061
"<Something> is not a function", error message, BC570–BC571
"<Something> can't be set by assignment," error message, BC573
"<Something> has no properties," error message, BC572
"<Something> has no property indexed by [i]", error message, BC573
"<Something> has no property named...", error message, BC572
"<Something> is null or not an object," error message, BC572
"<Something> is undefined," error message, BC570
Sort, 565
sort()
arrays, 323, 346–351
compare(), 347
sort, JavaScript database, BC771–BC777
source
regular expressions, BC472
RegularExpressionObject, BC485
source code
debugging, BC575–BC576
frames, 745–746
Java applets, BC538–BC539
space, JSON, 360
space characters, variable names and, 111
span, 65, 707
colgroup, BC342–BC343
display, BC691
div, BC692
id, 221
innerHTML, 731
nextSibling, 582
placeholder, 221
specified, W3C DOM, 647
splice(), arrays, 315, 323, 352
split(), 253
splitText(), Text, BC63–BC64
sqrt(), 278
SQRT1_2 (Math.SQRT1_2), 277
SQRT2 (Math.SQRT2), 277
src, 511, BC276
base, BC292
embed, BC463
file:, BC595
frame, 861
iframe, 874
Image, 1017–1021
image, BC152
images, 213, BC595
img, 208
<img>, 208
script, BC301
<script>, 38, BC595
<script><script>, 38
select, BC703
srcElement
event, 1093–1095
fireEvent(), 1050
IE4, BC414
srcUrn, event, 1096
Standard Generalized Markup Language (SGML), 5
standby, object, BC460
start
Image, 1021
ol, BC353
start(), marquee, BC21
startContainer, range, BC27–BC28
startOffset, range, BC28–BC29
startScroll(), setInterval(), 829
StartToEnd, setEndPoint(), BC98
StartToStart, setEndPoint(), BC98
startValue placeholder, 124
state, DHTML cross-browser map puzzle, BC752–BC753, BC758
statement, with, 392–393
statements, 492–499
; (semicolon), 99
delimiters, line terminators, 57
immediate, 100
JavaScript, 99–100
labeled, 393–396
scripts, 100–104
called by another function, 104
functions, 101
run by user, 103–104
running after loading, 101–103
states[], DHTML cross-browser map puzzle, BC761
statesIndexList, DHTML cross-browser map puzzle, BC753, BC758
static objects, 184
Event, 1045, 1098
JavaScript, 56
methods, 56
properties, 56
scripts, 56
static tables, BC638–BC641
status, BC282
IE, 835
mouseover, 788
window, 784–788
status bar, debugging, BC578
statusbar, window, 758–760
statusText, BC284
stepClip(), setInterval(), BC416
stop()
marquee, BC21
window, 848
StopImage, TextRange.execCommand(), BC80
stopPropagation(), event, 1121
strictErrorChecking, document, 953

String, 55, 180–183
  initialCaps(), 234
  Java, BC529
  java.lang, BC563
  length, 116, 275
  new, 429
  objects, 228–229
  prototype, 232–234
  value, 227
strings, 225–267
  case changes, 184
  concatenation, 114, 116–117, 180–181, 226, 414
  conversion
    numbers to strings, 116, 275–276
    strings to numbers, 115–116, 274–275
  data type, 110
  Date, 293
  definition of, 110
  exception handling, 403–404
  extracting characters from, 182–183
  HTML, 263–267
  inline characters, 227–228
  JSON, 358
  methods, 181–183
  numbers, 413–415
  parsing methods, 234–261
  properties, 231–261
  regular expressions, BC480–BC482
  searches, 183
  URLs, 267
  utility functions, 261–267
  value, 160
  variables, 226–227
string literals, 180
  variables, 227
String(), constructor, 231
string.anchor(), 266
string.big(), 264
string.blink(), 264
string.bold(), 264
string.charAt(), 234–235
string.charCodeAt(), 235–239
string.concat(), 239
string.escape(), BC175
string.fixed(), 264
string.fontcolor(), 264, 266
string.fontsize(), 264, 266
stringify(), JSON, 360
string.indexOf(), 239–240
string.italics(), 264
string.link(), 264
string.localecompare(), 241
string.match(), 241–244
stringObject, 230
string.prototype, constructor, 230
string.replace(), 244–249
  regular expressions, BC480, BC481
string.search(), 249
string.slice(), 249–253
  string.substring(), 249
string.small(), 264
string.split()
  cookies, 923
  order forms, BC666
string.split(), 253–254
  arrays, BC664, BC667
  Evaluator, 253
  regular expressions, 253
string.strike(), 264
string.sub(), 264
string.substr(), 254–257
  string.substring(), 254
string.substring(), 254–257
  string.charAt(), 235
  string.slice(), 249
  string.substr(), 254
string.sup(), 264
string.toLocaleLowerCase(), 260
string.toLocaleUpperCase(), 260
string.toLowerCase(), 260
string.toString(), 260–261
string.toUpperCase(), 260
string.valueOf(), 260–261
Stripes(), BC270
stroke(), canvas, 1042
strokeRect(), canvas, 1042
strokeStyle, canvas, 1038–1039
style, 604–605, BC208, BC211–BC212, BC226–BC257
  behaviors, BC625
  CSS, 220
  cssRule, BC225
  DHTML cross-browser map puzzle, BC750
  IE, BC613
  @import, BC209
  left, BC617
  object detection, BC616
  properties, BC211–BC212
  setExpression, 675
  style sheets, BC618
  top, BC617
  W3C, BC412
  W3C DOM, BC610
styles
  objects, BC207–BC271
  properties, BC210
style sheets, BC207–BC271, BC768–BC769. *See also* Cascading Style Sheets
  DHTML, 220
  elements, 263
  fieldset, BC123
  HTML, BC768
  IE, BC623–BC624
  import, BC209
  runtimeStyle, 601
  scripts, BC623–BC624
  <style>, BC618
style.backgroundColor, BC414
style.backgroundImage, BC412
style.clip, BC420
style.display, BC679
  for, BC677
styleFloat, BC244
style.height, resizeIt(), BC443
style.left, 587, BC420, BC425
style.right, BC420
styleSheet, BC208, BC212–BC222, BC768
  IE, 960
  methods, BC220–BC222
  parentStyle, BC209
  properties, BC213–BC220

styleSheets[], document, 953
style.top, 587, BC425
style.visibility,
BC434–BC436
style.width, resizeIt(),
BC443
style.zIndex, BC436
SubEthaEdit, 28
submit
forms, 173–177
if, 175
loops, 175
onclick, 175
submit(), 177
submit(), forms, BC121
Submit button, BC112
CGIs, BC121
onsubmit, BC121
substr(), 182
substring(), 183–184
cookies, 925
substringData(), Text,
BC58–BC63
subTotal, BC668–BC669
subwin
eval(), BC547
JSObject, BC547
subWindow, 138
subWrite()
closed, 150
makeNewWindow(), 150
suffixes, mimeType, BC388
summary, table, BC333–BC334
sUpperCaseValue, 162
Supported,
document.queryCommand,
972–973
surroundContents(), range,
BC47–BC48
swapNode(), 689
removeChild(), 675
replaceChild(), 678
swapState(), currState,
BC692
switch
break, 381
case, 381
expression, 381
if...else, 380
object, 380
<option>, 381
statements, 380–383
syntax, 380–381
variable, 380
syntax
for, 124
applet, BC450
area, 1025
body, 982
Boolean, 284
button, BC129
canvas, 1033
checkbox, BC135
clientInformation, BC361
document, 910
dot syntax form, 86, 504
embed, BC461
error objects, 408
event, 1074
Mozilla, 1098
NN6, 1098
frame, 855
frameset, 862
Function, 437–438
functions, 125
history, 900
hr, BC10
if, 122
if...else, 123
iframe, 869
Image, 1004–1005
link, BC293
location, 882
map, 1028
marquee, BC16
Math, 276
meta, BC297
navigator, BC361
Number, 280
Object, 474
object, BC454
ol, BC352
popup, 875
radio, BC143–BC144
range, BC23
RegExp, BC487
RegularExpressionObject,
BC483
runtime errors, BC564–BC565
script, BC300
select, BC178
selection, BC49
switch, 380–381
table, BC321
td, BC348
text, BC154
textarea, BC173
TextRange, BC65
th, BC348
title, BC302
tr, BC344
TreeWalker, 990
userProfile, BC404
window, 748–749
xml, BC275
XMLHTTPRequest, BC277
SyntaxError, 55
system requirements, 1133
systemId, document.doctype,
932
systemLanguage, navigator,
BC380
Systems Hungarian notation, 69

T

\t, tab inline character, 227
tabIndex, 573–574, 605–607
tabindex, <p>, 715
table, BC777
family hierarchy,
BC304–BC320
methods, BC335–BC337
properties, BC321–BC334
syntax, BC321
tables, BC637–BC651
cells
col, BC316
colgroup, BC316
modifying content,
BC306–BC309
populating, BC306
columns, modifying,
BC316–BC319
CSS, BC303
DHTML, BC646–BC651
dynamic, BC641–BC645

hash, arrays, 321–353
HTML, BC303
hybrid, BC645
IE4, BC310
list objects, BC303–BC359
lookup, BC652–BC664
database, BC652–BC653
forms, BC655–BC662
interface, BC654
isValid(), BC660
onsubmit, BC656
parallel arrays, BC654
parseInt(), BC660
script, BC655
search(), BC661
methods, BC777
NN6, BC310
objects, W3C DOM, BC319–BC320
rows, modifying, BC309–BC316
static, BC638–BC641
W3C, BC310
table of contents, BC674–BC694
Ajax, BC682–BC687
CSS, BC677
global variables, BC677, BC688
HTML, BC688–BC691
toggle(), BC691–BC693
XML, BC682–BC687
tableLayout, BC255
tabs, browsers, 136
Tabular Data Control (TDC), 561
tagName, 607–608
DOM, 159
tagName:P, identify(), 170
tags. *See also* Hypertext Markup Language
attributes, event binding, 1059–1060
browsers, 43
XML, 363
tags(), 689–690
tagUrn, 601–602
tan(), 278
target, 1001–1002
area, 1028
base, BC292–BC293
elements, 997
event, 1114–1117
event handlers, 997
form, 157, BC119
HTML, 997, BC119
link, BC296
window.open(), 137
targets, Netscape, BC601–BC603
tbodies, table, BC334
tbody, BC311, BC338–BC340
tables, BC305
td, BC347–BC352
syntax, BC348
tr, BC344
<td>, BC638
TDC. *See* Tabular Data Control
<td>...</td>, for, BC641
test()
regular expressions, BC481
RegularExpressionObject, BC486–BC487
"Test for equality(==) mistyped as assignment (=)? Assuming equality test," error message, BC573
testing, debugging, BC588–BC589
testValues(), 451–452
text
editors, 27–28
error messages, BC569–BC574
form controls, 159–162
forms, BC153–BC176
nodes, 87
W3C DOM, 962–963
rollovers, IE behaviors, BC631–BC635
value, 160–162
Text, BC57–BC64
document.queryCommand, 972–973
methods, BC58–BC64
properties, BC58
text, BC153–BC171
body, 983
event handlers, BC167–BC171
functions, 454
methods, BC164–BC167
properties, BC157–BC164
script, BC301
select, 168
syntax, BC154
TextRange, BC72
title, BC302
text boxes, numbers allowed (keyboard trapping), BC493–BC494
textAlign, BC235
textAlignLast, BC236
textarea, 158–159, BC173–BC176
cookies, BC729
focus(), 627
label, BC125
methods, BC176
properties, BC175–BC176
syntax, BC173
trace(), BC584
textAutospace, BC236
textContent, 608
textDecoration, BC236
TextEdit, 28
TextEvent, Mozilla, 959
textIndent, BC236
textJustify, BC237
textJustifyTrim, BC237
textkashidaSpace, BC237
TextNode, BC57–BC64
methods, BC58–BC64
properties, BC58
textOverflow, BC237
TextQualifier, 562
TextRange, BC2, BC64–BC99
browser compatibility, BC67–BC68
methods, BC72–BC99
null, 595
properties, BC68–BC72
Range, BC68
syntax, BC65
TextRectangle, 664, BC99–BC102
textShadow, BC237
texttop, iframe.align, 871
textTransform, BC237
textUnderlinePosition, BC238
text/xml, XMLHttpRequest, BC279

T

tfoot, BC311, BC338–BC340
  table, BC334
th, BC347–BC352
  syntax, BC348
<th>, columnHeads, BC670
Thawte Digital Certificate Services, BC597
thead, BC311, BC338–BC340
  table, BC334
  tables, BC305
thinking web pages, 79
this
  buildMsg(), 334
  click, 170
  element, BC625
  form, BC108
  keywords, 179
  object methods, 56
  object operator, 429–430
  parameters, BC109
  processData(), 170–173
  propertyName, BC109
  value, BC109-BC110
  verifySong(), 170–173
throw
  catch, 402
  exception handling, 402–407
throwaway passwords, 922
time
  arithmetic, 296–298
  validation, BC501–BC505
    date range-checking functions, BC513–BC519
    date validation, BC502–BC505, BC512–BC519
time stamps
  browsers, BC707
  newAsOf(), BC713
time tracking, GMT, BC706–BC707
time zones, 285–286
  dateObj .getTimezoneOffset(), 293
timeIt(), ondrag, 705
timeStamp, event, 1117–1118
timing problems, debugging, BC576–BC577
tipStyle, div, BC724
title, 81, 608–609, BC301–BC302
  all[], 544
  document, 953–954
  document.title, BC302
  document.write(), 979
  properties, BC302
  styleSheet, BC219–BC220
  syntax, BC302
titlebar, window.open(), 807
toArray(), VBArray, 499
toc.html, BC676
toDateString(), dateObj, 291
toElement, event, 1086–1088
toExponential(), 276, 282–283
toFixed(), 276, 282–283
toggle()
  display, BC679
  href, BC677, BC681
  img, BC679, BC691–BC692
  onclick, BC691
  table of contents, BC691–BC693
toggleBar(), 759
toggleColor(), BC547
toggleComplete(), 656
toggleEdit(), contentEditable, 559
toGMTString(), dateObj, 291, 293
toHex(), 274
toJSON(), 360
tolerance, calculations, BC698
toLocaleDateString(), 293
  dateObj, 291
toLocaleString(), 284
  arrays, 323, 353
  Date, 69
  dateObj, 291, 293
toLocaleTimeString(), 293
  dateObj, 291
toLowerCase(), 181, 260, BC563
"Too many JavaScript errors," error message, BC574
toolbar
  window, 758–760
  window.open(), 807
top, BC246, BC622
  iframe.align, 871
  parent, 741, 743
  style, BC617
  TextRectangle, BC100–BC102
  window, 788
  window.open(), 807
_top, <form>, BC119
top-level objects, 135–152
topMargin, body, 984–985
toPrecision(), 276, 282–283
toSource(), 474
  Object, 478
toString(), 260–261, 293, 690
  arrays, 323, 353
  Boolean, 284
  dateObj, 292
  error.Object, 410
  functions, 447
  JSObject, BC544
  methods, 487
  Number, 276, 283–284
  Object, 474, 478
  range, BC49
  selection, BC55
  XML, 372
toUpperCase(), 162, 181, 260, BC526, BC563
toUTCString(), dateObj, 292
toXMLString(), BC286, BC288
  XML, 372
tr, BC311, BC343–BC347
  methods, BC347
  properties, BC345–BC346
  syntax, BC344
TRACE, BC585
trace(), BC583–BC584
  HTML, BC586
  invoking, BC585–BC586
trace.js, BC584–BC585
transferHTML(), innerHTML, BC432
translate(), canvas, 1042
trees. *See* node trees
TreeWalker, 963–964, 990–993

methods, 992–993
properties, 991–992
syntax, 990
troubleshooting, 1135
true, Boolean math, 420–424
trueSpeed, marquee, BC21
trusted ActiveX, Microsoft, 54
try, 399
try...catch, privilege errors, BC606
try/catch, XMLHttpRequest, BC687
try-catch-finally, exception handling, 398–402
turnOn(), addBehavior(), 613
type, 1002
browsers, 40, 95, 1002
button, BC132
checkbox, BC138
cssRule, BC209, BC226
document.selection, BC50
event, 1096–1097, 1118–1119
HTML, 40
tags, 95
IE, 40
if, 175
image, BC152
<input>, BC115
li, BC357
link, BC296–BC297
mimeType, BC387–BC388
Mozilla, 40
object, BC460
ol, BC353–BC354
radio, BC148
script, BC301
<script>, 38–39, 40
select, BC196
selection, BC51–BC53
style, BC212
text, BC161
ul, BC355–BC356
W3C, 40
Type, MIME, 1000
typeDetail, selection, BC53
TypeError, 55
typeof
constructor, 231
operators, 431–432
precedence, 434

U

UIEvent, Mozilla, 959
UIEvents, Mozilla, 959
ul, BC355–BC356
UnBookmark, document.execCommand(), 969
unbound(), VBArray, 499
"Uncaught exception," error message, BC574
unconditional GET, 898
Underline, TextRange.execCommand(), BC80
unescape(), 483–484, 891
encodeURIComponent(), 482
unhighlight(), setCapture(), 669
unicodeBidi, BC238
Uniform Resource Name (URN), 1002
uninitialized, readyState, 598
uniqueID, 609–611
units, embed, BC463
UniversalBrowserAccess, BC602
UniversalBrowserRead, BC602
document, BC603
UniversalBrowserWrite, 808, BC602
UniversalFileAccess, BC602
UniversalFileRead, BC602
UniversalFileWrite, BC602
UniversalPreferencesRead, BC602
UniversalPreferencesWrite, BC602
UniversalSendMail, BC602
Unlink, document.execCommand(), 969
Unselect, document.execCommand(), 969
unselectable, 611–612
unshift(), arrays, 323, 340–341
"Unspecified error," error message, BC574
"Unterminated string constant," error message, BC571
"Unterminated string literal," error message, BC571
unwatch(), 474, 492
Object, 478
update expression, for, 124, 384
update flags, BC705–BC714
HTML, BC709
updateClock(), setExpression(), 687
updated flags, JavaScript, BC710
updateInterval, screen, BC404
upperMe(), onchange, 162
URIError, 55
URL, document, 945–948
url, meta, BC299
URLs
% (percent sign), 483
"Access is denied," BC591
location, 142–143
strings, 267
window.open(), 137
URLUnencoded, document, 954
URN. *See* Uniform Resource Name
urn, 1002
urns(), 690–691
U.S. states
array, 130–133, BC510–BC511
name field (validation), BC509–BC512
validation function for, BC511–BC512
UseHeader, 562
useMap
Image, 1021–1022
object, BC460
usemap, # (hash symbol), 1025
user agent, JavaScript, 42–46
userAgent, BC362–BC363, BC366–BC371
browsers, BC368–BC371
navigator, BC367, BC380

UserLand Software, Inc., BC682
userLanguage, navigator, BC380
userProfile
- methods, BC407–BC410
- navigator, BC380–BC381, BC404–BC410
- syntax, BC404

Using experimental feature: HTML5 Conformance Checker, 72
UTC. *See* Coordinated Universal Time
UTF-8, HTML, 63
utility functions, strings, 261–267

V

val1, condition, 379
validate(), BC506, BC507, BC508, BC512, BC520
validate.php
- action, 379
- form, 174

validation. *See also* data-entry validation; industrial-strength data-entry validation
- batch mode, BC494–BC495
- cross-confirmation, BC519–BC521
- CSS Validation Service, 31
- custom functions, BC498–BC499
- data-entry, 77, BC492–BC523
- Date, forms, 304–309
- HTML Validator, 7, 20, 31
- master validation function, BC500–BC501
- real-time, BC492–BC494, BC504–BC505

Validation Service, CSS, 31
validator.w3.org, 72
validDate(), BC502, BC504
vAlign, BC339–BC340
Value, document.queryCommand, 972–973
value
- button, BC132
- checkbox, 163
- checkbox, BC138–BC140
- form.elements, 158
- li, BC357
- radio, BC148
- select, 170, BC196, BC701
- setHRAttr(), BC11
- strings, 160
- text, 160–162
- text, BC161–BC164
- this, BC110
- W3C DOM, 647

values
- cookies, 923
- defined, 110
- event handlers, BC111
- JavaScript functions, 56–57
- parameters, BC111
- references, 56–57
- String, 227

valueOf(), 260–261, 474
- functions, 447
- Object, 478

var, 111, 128
- for, 384
- statement, 495–496

variable, switch, 380
variables. *See also* global variables
- as baskets, 111
- creating, 111
- declaring, 111
- event handlers, BC736
- expression evaluation, 112–114
- global, 127–128
- global scope, 56
- initializing, 111
- JavaScript, 56
- local, 127, 448–453
- local scope, 56
- names, identifiers, 495–496
- naming, 111–112
- parameters, 453–454
- scope, 127–128, 448–453
- string literals, 227
- strings, 226–227
- testValues(), 451

VBArray, 498–499
VBScript
- CreateObject(), 496
- object, BC399

vendor, navigator, BC379
vendorSub, navigator, BC379
Venkman, Mozilla, BC580
verifySong(), this, 170–173
VeriSign, BC597
version, html, BC290–BC291
verticalAlign, BC244
view, event, 1119
visibility, BC244
- Boolean, BC621
- layers, BC434–BC436

Visual Basic, safe arrays, 498
vLink, body, 983
vlinkColor, document, 912–915
void
- frame, BC327
- Java, BC526
- operator, 432
- operators, precedence, 434

vsides, frame, BC327
vspace
- applet, BC452–BC453
- iframe, 872–873
- Image, 1013
- marquee, BC20
- object, BC459

W

\W, regular expressions metacharacter, BC468
\w, regular expressions metacharacter, BC468
W3C. *See* World Wide Web Consortium
W3C DOM, 48, 79–94, 516–532
- browsers, 174
- development of, 19–20
- DHTML, 221
- diagram, top-level objects, 81
- event bubbling, 1053–1057
- events, 1044, 1052–1058, 1098
- HTML structure and, 79–81

IE5, BC610
JavaScript, 179
Mozilla, 196
OO, 523–532
table objects, BC319–BC320
text nodes, 962–963
XML, BC272
walkChildNodes(), W3C DOM, BC582
walkChildren(), document.getElementById(), 553
warnings, Firefox, BC574–BC575
watch(), 474, 492
Object, 478
Wave(), BC271
wave(), BC260
Weather Bonk, BC785
Web Accessibility Evaluation Tools, 31
Web Content Accessibility Guidelines, 31
Web Forms 2.0, BC103
browsers, BC126–BC127
DOM, BC127
HTML, BC126
web pages
document.write(), 147
thinking, 79
web sites
components, 4–5
web traffic, 4
web-authoring environment, 28–31
Mac OS X, 27, 30
reloading issues, 31
Windows, 27, 29–30
web-authoring technologies, 3–9. *See also* Cascading Style Sheets; Hypertext Markup Language; JavaScript; plug-ins; server-side programming
WebKit, Drosera, BC580
What You See Is What You Get (WYSIWYG), 27
whatToShow, TreeWalker, 991–992
wheelData, event, 1097
while, 388–389
break, 389
continue, 389
control structures, 56
debugging, BC588
oTarget.firstChild, 71
regular expressions, BC481
whitespace, 57
whiteSpace, BC238
[whitespace], 156
wholeWord, window.find(), 798
widows, BC255
width, 572–573, BC245, BC247
applet, BC452
<canvas>, 1033
colgroup, BC343
document, 940–941
embed, BC462
frame, 858
hr, BC14
iframe, 872
Image, 1012
marquee, BC20
object, BC458–BC459
pixels, 208
screen, BC401–BC402
table, BC329–BC330
td, BC351
window.open(), 807
willItFit(), 439
window, 82, 135–142, 739–880, BC360
alert(), 140
captureEvents(), 1046
document.write(), 749
DOM, 136
event, 1074
event bubbling, 1049
event handlers, 848–854, 1052
IE, 835
Java applet, BC543
load, 141–142
methods, 136–137, 789–848
onblur, 695
onload, 67, BC729, BC733
opener, 202
properties, 136–137, 750–789
releaseEvents(), 1046
self, 137, 191
syntax, 748–749
window object checks, BC594
window.alert(), 140
window.blur(), 628–629
window.close(), 138
window.confirm(), 140–141
window.dialogArguments, showModelessDialog(), 843
window.event
evt, 1065
IE, 1060
windowFeatures, 806
window.focus(), 628–629
window.getComputedStyle()
CSS, BC610
DHTML cross-browser map puzzle, BC750
window.maximize, 508
window.onerror, 397
window.open(), 137, 202, 628
eval(), BC547
IE, 812
Mozilla, 808–811
Netscape, 808–811
window.opener, 626
window.prompt(), 141
window.returnValue, handleOK(), 837–838
Windows Registry Editor, BC397
Windows web-authoring environment, 27, 29–30
window.setTimeout(), BC112, BC576
windows.open(), document.close(), 812
window.status, BC578
Winer, Dave, BC682
WinIE Objects, 496
with, statement, 392–393
Word, Microsoft, 28
wordBreak, BC238
WordPad, 28
wordSpacing, BC238
wordWrap, BC238–BC239

World Wide Web Consortium (W3C). *See also* W3C DOM
- event binding, 1062–1063
- events, 1071–1074
- getElementById(), BC617
- getObject, BC617
- HTML Validator, 7, 20, 31
- innerHTML, BC116, BC687
- Opera, BC615
- style, BC412
- tables, BC310
- type, 40

wrap, textarea, BC176
wraparound
- textarea, BC174
- window.find(), 798

write(), document, 974–980
writeln(), document, 974–980
writingMode, BC239
WRONG_DOCUMENT_ERR, 400
WYSIWYG (What You See Is What You Get), 27

X

x, Image, 1021–1022
XForms, BC126
XHTML, 7
- -->, 44
- <, 44
- <!--, 44
- &, 44
- case sensitivity, 64
- document.write(), 147
- Java applets, BC527–BC528
- scripts, 98–99
  - comments, 43–44
- target, 1001
- XML, 364

XML. *See* Extensible Markup Language
XML, 55
xml, BC275–BC276
- syntax, BC275
- W3C DOM, BC276

<xml>, BC272
XML Core Services (MSXML), BC277
XML Schema, 516
xmlDoc, XMLHttpRequest, BC279
XMLDocument, document, BC276
xmlEncoding, document, 954
XMLHttpRequest, BC272–BC273, BC276–BC286
- IE, BC687
- loadXMLDoc(), BC769
- methods, BC284–BC286
- properties, BC281–BC284
- syntax, BC277
- try/catch, BC687

XMLHttpRequest()
- iframe, 196
- JSON, 361

XMLList, 55, 366–370
xmlStandalone, document, 954
xmlVersion, document, 954
XOR, currState, BC692
XOR, bitwise operator, 425
xpconnect, NPAPI, 750
xRay(), BC260, BC271

Y

y, Image, 1021–1022
yourAge, 112

Z

zIndex, BC247, BC621
- layers, BC436–BC439

z-lock, window.open(), 807
zoom, BC245

Wiley Publishing, Inc.
End-User License Agreement

READ THIS. You should carefully read these terms and conditions before opening the software packet(s) included with this book "Book". This is a license agreement "Agreement" between you and Wiley Publishing, Inc. "WPI". By opening the accompanying software packet(s), you acknowledge that you have read and accept the following terms and conditions. If you do not agree and do not want to be bound by such terms and conditions, promptly return the Book and the unopened software packet(s) to the place you obtained them for a full refund.

1. **License Grant.** WPI grants to you (either an individual or entity) a nonexclusive license to use one copy of the enclosed software program(s) (collectively, the "Software," solely for your own personal or business purposes on a single computer (whether a standard computer or a workstation component of a multi-user network). The Software is in use on a computer when it is loaded into temporary memory (RAM) or installed into permanent memory (hard disk, CD-ROM, or other storage device). WPI reserves all rights not expressly granted herein.
2. **Ownership.** WPI is the owner of all right, title, and interest, including copyright, in and to the compilation of the Software recorded on the physical packet included with this Book "Software Media". Copyright to the individual programs recorded on the Software Media is owned by the author or other authorized copyright owner of each program. Ownership of the Software and all proprietary rights relating thereto remain with WPI and its licensers.
3. **Restrictions On Use and Transfer.**
   - **(a)** You may only (i) make one copy of the Software for backup or archival purposes, or (ii) transfer the Software to a single hard disk, provided that you keep the original for backup or archival purposes. You may not (i) rent or lease the Software, (ii) copy or reproduce the Software through a LAN or other network system or through any computer subscriber system or bulletin-board system, or (iii) modify, adapt, or create derivative works based on the Software.
   - **(b)** You may not reverse engineer, decompile, or disassemble the Software. You may transfer the Software and user documentation on a permanent basis, provided that the transferee agrees to accept the terms and conditions of this Agreement and you retain no copies. If the Software is an update or has been updated, any transfer must include the most recent update and all prior versions.
4. **Restrictions on Use of Individual Programs.** You must follow the individual requirements and restrictions detailed for each individual program in the About the CD-ROM appendix of this Book or on the Software Media. These limitations are also contained in the individual license agreements recorded on the Software Media. These limitations may include a requirement that after using the program for a specified period of time, the user must pay a registration fee or discontinue use. By opening the Software packet(s), you will be agreeing to abide by the licenses and restrictions for these individual programs that are detailed in the About the CD-ROM appendix and/or on the Software Media. None of the material on this Software Media or listed in this Book may ever be redistributed, in original or modified form, for commercial purposes.
5. **Limited Warranty.**
   - **(a)** WPI warrants that the Software and Software Media are free from defects in materials and workmanship under normal use for a period of sixty (60) days from the date of purchase

of this Book. If WPI receives notification within the warranty period of defects in materials or workmanship, WPI will replace the defective Software Media.

(b) WPI AND THE AUTHOR(S) OF THE BOOK DISCLAIM ALL OTHER WARRANTIES, EXPRESS OR IMPLIED, INCLUDING WITHOUT LIMITATION IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, WITH RESPECT TO THE SOFTWARE, THE PROGRAMS, THE SOURCE CODE CONTAINED THEREIN, AND/OR THE TECHNIQUES DESCRIBED IN THIS BOOK. WPI DOES NOT WARRANT THAT THE FUNCTIONS CONTAINED IN THE SOFTWARE WILL MEET YOUR REQUIREMENTS OR THAT THE OPERATION OF THE SOFTWARE WILL BE ERROR FREE.

(c) This limited warranty gives you specific legal rights, and you may have other rights that vary from jurisdiction to jurisdiction.

6. **Remedies.**

(a) WPI's entire liability and your exclusive remedy for defects in materials and workmanship shall be limited to replacement of the Software Media, which may be returned to WPI with a copy of your receipt at the following address: Software Media Fulfillment Department, Attn.: *JavaScript Bible, 7e*, Wiley Publishing, Inc., 10475 Crosspoint Blvd., Indianapolis, IN 46256, or call 1-800-762-2974. Please allow four to six weeks for delivery. This Limited Warranty is void if failure of the Software Media has resulted from accident, abuse, or misapplication. Any replacement Software Media will be warranted for the remainder of the original warranty period or thirty (30) days, whichever is longer.

(b) In no event shall WPI or the author be liable for any damages whatsoever (including without limitation damages for loss of business profits, business interruption, loss of business information, or any other pecuniary loss) arising from the use of or inability to use the Book or the Software, even if WPI has been advised of the possibility of such damages.

(c) Because some jurisdictions do not allow the exclusion or limitation of liability for consequential or incidental damages, the above limitation or exclusion may not apply to you.

7. **U.S. Government Restricted Rights.** Use, duplication, or disclosure of the Software for or on behalf of the United States of America, its agencies and/or instrumentalities "U.S. Government" is subject to restrictions as stated in paragraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause of DFARS 252.227-7013, or subparagraphs (c) (1) and (2) of the Commercial Computer Software - Restricted Rights clause at FAR 52.227-19, and in similar clauses in the NASA FAR supplement, as applicable.

8. **General.** This Agreement constitutes the entire understanding of the parties and revokes and supersedes all prior agreements, oral or written, between them and may not be modified or amended except in a writing signed by both parties hereto that specifically refers to this Agreement. This Agreement shall take precedence over any other documents that may be in conflict herewith. If any one or more provisions contained in this Agreement are held by any court or tribunal to be invalid, illegal, or otherwise unenforceable, each and every other provision shall remain in full force and effect.