Handling the Client Request: Form Data

Topics in This Chapter

- Reading individual request parameters
- Reading the entire set of request parameters
- Handling missing and malformed data
- Filtering special characters out of the request parameters
- Automatically filling in a data object with request parameter values
- Dealing with incomplete form submissions

Chapter 4

One of the main motivations for building Web pages dynamically is so that the result can be based upon user input. This chapter shows you how to access that input (Sections 4.1–4.4). It also shows you how to use default values when some of the expected parameters are missing (Section 4.5), how to filter < and > out of the request data to avoid messing up the HTML results (Section 4.6), how to create "form beans" that can be automatically populated from the request data (Section 4.7), and how, when required request parameters are missing, to redisplay the form with the missing values highlighted (Section 4.8).

4.1 The Role of Form Data

If you've ever used a search engine, visited an online bookstore, tracked stocks on the Web, or asked a Web-based site for quotes on plane tickets, you've probably seen funny-looking URLs like http://*host*/*path*?user=Marty+Hall&origin=bwi&dest=sfo. The part after the question mark (i.e., user=Marty+Hall&origin=bwi&dest=sfo) is known as *form data* (or *query data*) and is the most common way to get information from a Web page to a server-side program. Form data can be attached to the end of the URL after a question mark (as above) for `GET` requests; form data can also be sent to the server on a separate line for `POST` requests. If you're not familiar with

HTML forms, Chapter 19 (Creating and Processing HTML Forms) gives details on how to build forms that collect and transmit data of this sort. However, here are the basics.

1. **Use the `FORM` element to create an HTML form.** Use the `ACTION` attribute to designate the address of the servlet or JSP page that will process the results; you can use an absolute or relative URL. For example:

   ```
   <FORM ACTION="...">...</FORM>
   ```

   If `ACTION` is omitted, the data is submitted to the URL of the current page.
2. **Use input elements to collect user data.** Place the elements between the start and end tags of the `FORM` element and give each input element a `NAME`. Textfields are the most common input element; they are created with the following.

   ```
   <INPUT TYPE="TEXT" NAME="...">
   ```

3. **Place a submit button near the bottom of the form.** For example:

   ```
   <INPUT TYPE="SUBMIT">
   ```

   When the button is pressed, the URL designated by the form's `ACTION` is invoked. With `GET` requests, a question mark and name/value pairs are attached to the end of the URL, where the names come from the `NAME` attributes in the HTML input elements and the values come from the end user. With `POST` requests, the same data is sent, but on a separate request line instead of attached to the URL.

Extracting the needed information from this form data is traditionally one of the most tedious parts of server-side programming.

First of all, before servlets you generally had to read the data one way for `GET` requests (in traditional CGI, this is usually through the `QUERY_STRING` environment variable) and a different way for `POST` requests (by reading the standard input in traditional CGI).

Second, you have to chop the pairs at the ampersands, then separate the parameter names (left of the equal signs) from the parameter values (right of the equal signs).

Third, you have to *URL-decode* the values: reverse the encoding that the browser uses on certain characters. Alphanumeric characters are sent unchanged by the browser, but spaces are converted to plus signs and other characters are converted to *%XX*, where *XX* is the ASCII (or ISO Latin-1) value of the character, in hex. For example, if someone enters a value of "`~hall, ~gates, and ~mcnealy`" into a textfield with the name `users` in an HTML form, the data is sent as "`users=%7Ehall%2C+%7Egates%2C+and+%7Emcnealy`", and the server-side program has to reconstitute the original string.

Finally, the fourth reason that it is tedious to parse form data with traditional server-side technologies is that values can be omitted (e.g., "`param1=val1&`**`param2=`**`&param3=val3`") or a parameter can appear more than once (e.g., "**`param1=val1`**`&param2=val2&`**`param1=val3`**"), so your parsing code needs special cases for these situations.

Fortunately, servlets help us with much of this tedious parsing. That's the topic of the next section.

4.2 Reading Form Data from Servlets

One of the nice features of servlets is that all of this form parsing is handled automatically. You call `request.getParameter` to get the value of a form parameter. You can also call `request.getParameterValues` if the parameter appears more than once, or you can call `request.getParameterNames` if you want a complete list of all parameters in the current request. In the rare cases in which you need to read the raw request data and parse it yourself, call `getReader` or `getInputStream`.

Reading Single Values: getParameter

To read a request (form) parameter, you simply call the `getParameter` method of `HttpServletRequest`, supplying the case-sensitive parameter name as an argument. You supply the parameter name exactly as it appeared in the HTML source code, and you get the result exactly as the end user entered it; any necessary URL-decoding is done automatically. Unlike the case with many alternatives to servlet technology, you use `getParameter` exactly the same way when the data is sent by `GET` (i.e., from within the `doGet` method) as you do when it is sent by `POST` (i.e., from within `doPost`); the servlet knows which request method the client used and automatically uses the appropriate method to read the data. An empty `String` is returned if the parameter exists but has no value (i.e., the user left the corresponding textfield empty when submitting the form), and `null` is returned if there was no such parameter.

Parameter names are case sensitive so, for example, `request.getParameter("Param1")` and `request.getParameter("param1")` are *not* interchangeable.

Core Warning

The values supplied to `getParameter` and `getParameterValues` are case sensitive.

Reading Multiple Values: getParameterValues

If the same parameter name might appear in the form data more than once, you should call `getParameterValues` (which returns an array of strings) instead of `getParameter` (which returns a single string corresponding to the first occurrence of the parameter). The return value of `getParameterValues` is `null` for nonexistent parameter names and is a one-element array when the parameter has only a single value.

Now, if you are the author of the HTML form, it is usually best to ensure that each textfield, checkbox, or other user interface element has a unique name. That way, you can just stick with the simpler `getParameter` method and avoid `getParameterValues` altogether. However, you sometimes write servlets or JSP pages that handle other people's HTML forms, so you have to be able to deal with all possible cases. Besides, multiselectable list boxes (i.e., HTML `SELECT` elements with the `MULTIPLE` attribute set; see Chapter 19 for details) repeat the parameter name for each selected element in the list. So, you cannot always avoid multiple values.

Looking Up Parameter Names: getParameterNames and getParameterMap

Most servlets look for a specific set of parameter names; in most cases, if the servlet does not know the name of the parameter, it does not know what to do with it either. So, your primary tool should be `getParameter`. However, it is sometimes useful to get a full list of parameter names. The primary utility of the full list is debugging, but you occasionally use the list for applications where the parameter names are very dynamic. For example, the names themselves might tell the system what to do with the parameters (e.g., `row-1-col-3-value`), the system might build a database update assuming that the parameter names are database column names, or the servlet might look for a few specific names and then pass the rest of the names to another application.

Use `getParameterNames` to get this list in the form of an `Enumeration`, each entry of which can be cast to a `String` and used in a `getParameter` or `getParameterValues` call. If there are no parameters in the current request, `getParameterNames` returns an empty `Enumeration` (not `null`). Note that `Enumeration` is an interface that merely guarantees that the actual class will have `hasMoreElements` and `nextElement` methods: there is no guarantee that any particular underlying data structure will be used. And, since some common data structures (hash tables, in particular) scramble the order of the elements, you should not count on `getParameterNames` returning the parameters in the order in which they appeared in the HTML form.

Core Warning

Don't count on `getParameterNames` *returning the names in any particular order.*

An alternative to `getParameterNames` is `getParameterMap`. This method returns a `Map`: the parameter names (strings) are the table keys and the parameter values (string arrays as returned by `getParameterNames`) are the table values.

Reading Raw Form Data and Parsing Uploaded Files: getReader or getInputStream

Rather than reading individual form parameters, you can access the query data directly by calling `getReader` or `getInputStream` on the `HttpServletRequest` and then using that stream to parse the raw input. Note, however, that if you read the data in this manner, it is not guaranteed to be available with `getParameter`.

Reading the raw data is a bad idea for regular parameters since the input is neither parsed (separated into entries specific to each parameter) nor URL-decoded (translated so that plus signs become spaces and *%XX* is replaced by the original ASCII or ISO Latin-1 character corresponding to the hex value *XX*). However, reading the raw input is of use in two situations.

The first case in which you might read and parse the data yourself is when the data comes from a custom client rather than by an HTML form. The most common custom client is an applet; applet-servlet communication of this nature is discussed in Volume 2 of this book.

The second situation in which you might read the data yourself is when the data is from an uploaded file. HTML supports a `FORM` element (`<INPUT TYPE="FILE"...>`) that lets the client upload a file to the server. Unfortunately, the servlet API defines no mechanism to read such files. So, you need a third-party library to do so. One of the most popular ones is from the Apache Jakarta Project. See http://jakarta.apache.org/commons/fileupload/ for details.

Reading Input in Multiple Character Sets: setCharacterEncoding

By default, `request.getParameter` interprets input using the server's current character set. To change this default, use the `setCharacterEncoding` method of `ServletRequest`. But, what if input could be in more than one character set? In such a case, you cannot simply call `setCharacterEncoding` with a normal character

set name. The reason for this restriction is that `setCharacterEncoding` must be called *before* you access any request parameters, and in many cases you use a request parameter (e.g., a checkbox) to determine the character set.

So, you are left with two choices: read the parameter in one character set and convert it to another, or use an autodetect feature provided with some character sets.

For the first option, you would read the parameter of interest, use `getBytes` to extract the raw bytes, then pass those bytes to the `String` constructor along with the name of the desired character set. Here is an example that converts a parameter to Japanese:

```
String firstNameWrongEncoding = request.getParameter("firstName");
String firstName =
  new String(firstNameWrongEncoding.getBytes(), "Shift_JIS");
```

For the second option, you would use a character set that supports detection and conversion from the default set. A full list of character sets supported in Java is available at http://java.sun.com/j2se/1.4.1/docs/guide/intl/encoding.doc.html. For example, to allow input in either English or Japanese, you might use the following.

```
request.setCharacterEncoding("JISAutoDetect");
String firstName = request.getParameter("firstName");
```

4.3 Example: Reading Three Parameters

Listing 4.1 presents a simple servlet called `ThreeParams` that reads form parameters named `param1`, `param2`, and `param3` and places their values in a bulleted list. Although you are required to specify *response* settings (see Chapters 6 and 7) before beginning to generate the content, you are not required to read the *request* parameters at any particular place in your code. So, we read the parameters only when we are ready to use them. Also recall that since the `ThreeParams` class is in the `coreservlets` package, it is deployed to the coreservlets subdirectory of the WEB-INF/classes directory of your Web application (the default Web application in this case).

As we will see later, this servlet is a perfect example of a case that would be dramatically simpler with JSP. See Section 11.6 (Comparing Servlets to JSP Pages) for an equivalent JSP version.

Listing 4.1 ThreeParams.java

```
package coreservlets;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

/** Simple servlet that reads three parameters from the
 *  form data.
 */

public class ThreeParams extends HttpServlet {
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    String title = "Reading Three Request Parameters";
    String docType =
      "<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.0 " +
      "Transitional//EN\">\n";
    out.println(docType +
                "<HTML>\n" +
                "<HEAD><TITLE>" + title + "</TITLE></HEAD>\n" +
                "<BODY BGCOLOR=\"#FDF5E6\">\n" +
                "<H1 ALIGN=\"CENTER\">" + title + "</H1>\n" +
                "<UL>\n" +
                "  <LI><B>param1</B>: "
                + request.getParameter("param1") + "\n" +
                "  <LI><B>param2</B>: "
                + request.getParameter("param2") + "\n" +
                "  <LI><B>param3</B>: "
                + request.getParameter("param3") + "\n" +
                "</UL>\n" +
                "</BODY></HTML>");
  }
}
```

Listing 4.2 shows an HTML form that collects user input and sends it to this servlet. By using an ACTION URL beginning with a slash (/servlet/coreservlets.ThreeParams), you can install the form anywhere in the default Web application; you can move the HTML form to another directory or move both the HTML form and the servlet to another machine, all without editing the HTML form or the servlet. The general principle that form URLs beginning with slashes increases portability holds true even when you use custom Web applications, but you have to include the Web application

prefix in the URL. See Section 2.11 (Web Applications: A Preview) for details on Web applications. There are other ways to write the URLs that also simplify portability, but the most important point is to use relative URLs (no host name), not absolute ones (i.e., http://*host*/...). If you use absolute URLs, you have to edit the forms whenever you move the Web application from one machine to another. Since you almost certainly develop on one machine and deploy on another, use of absolute URLs should be strictly avoided.

Core Approach

Use form `ACTION` *URLs that are relative, not absolute.*

Listing 4.2 ThreeParamsForm.html

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML><HEAD><TITLE>Collecting Three Parameters</TITLE></HEAD>
<BODY BGCOLOR="#FDF5E6">
<H1 ALIGN="CENTER">Collecting Three Parameters</H1>

<FORM ACTION="/servlet/coreservlets.ThreeParams">
  First Parameter:  <INPUT TYPE="TEXT" NAME="param1"><BR>
  Second Parameter: <INPUT TYPE="TEXT" NAME="param2"><BR>
  Third Parameter:  <INPUT TYPE="TEXT" NAME="param3"><BR>
  <CENTER><INPUT TYPE="SUBMIT"></CENTER>
</FORM>

</BODY></HTML>
```

Recall that the location of the default Web application varies from server to server. HTML forms go in the top-level directory or in subdirectories other than WEB-INF. If we place the HTML page in the form-data subdirectory and access it from the local machine, then the full installation location on the three sample servers used in the book is as follows:

- **Tomcat Location**
  install_dir/webapps/ROOT/form-data/ThreeParamsForm.html
- **JRun Location**
  install_dir/servers/default/default-ear/default-war/form-data/ThreeParamsForm.html

- **Resin Location**
  install_dir/doc/form-data/ThreeParamsForm.html
- **Corresponding URL**
  http://localhost/form-data/ThreeParamsForm.html

Figure 4–1 shows the HTML form when the user has entered the home directory names of three famous Internet personalities. OK, OK, only two of them are famous,[1] but the point here is that the tilde (~) is a nonalphanumeric character and will be URL-encoded by the browser when the form is submitted. Figure 4–2 shows the result of the servlet; note the URL-encoded values on the address line but the original form field values in the output: `getParameter` always returns the values as the end user typed them in, regardless of how they were sent over the network.

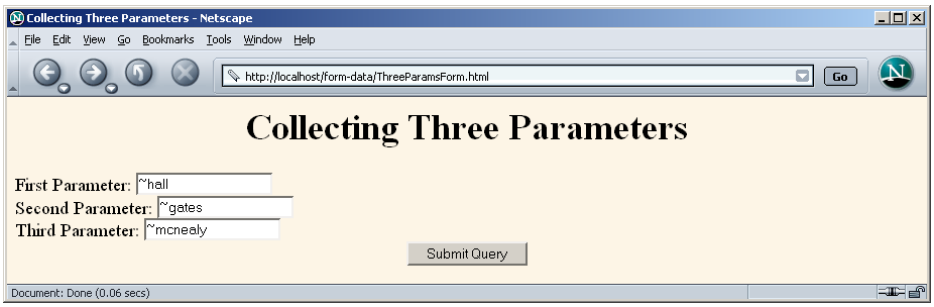


Figure 4–1 Front end to parameter-processing servlet.

Figure 4–2 Result of parameter-processing servlet: request parameters are URL-decoded automatically.

1. Gates isn't *that* famous, after all.

4.4 Example: Reading All Parameters

The previous example extracts parameter values from the form data according to prespecified parameter names. It also assumes that each parameter has exactly one value. Here's an example that looks up *all* the parameter names that are sent and puts their values in a table. It highlights parameters that have missing values as well as ones that have multiple values. Although this approach is rarely used in production servlets (if you don't know the names of the form parameters, you probably don't know what to do with them), it is quite useful for debugging.

First, the servlet looks up all the parameter names with the `getParameterNames` method of `HttpServletRequest`. This method returns an `Enumeration` that contains the parameter names in an unspecified order. Next, the servlet loops down the `Enumeration` in the standard manner, using `hasMoreElements` to determine when to stop and using `nextElement` to get each parameter name. Since `nextElement` returns an `Object`, the servlet casts the result to a `String` and passes that to `getParameterValues`, yielding an array of strings. If that array is one entry long and contains only an empty string, then the parameter had no values and the servlet generates an italicized "No Value" entry. If the array is more than one entry long, then the parameter had multiple values and the values are displayed in a bulleted list. Otherwise, the single value is placed directly into the table.

The source code for the servlet is shown in Listing 4.3; Listing 4.4 shows the HTML code for a front end you can use to try out the servlet. Figures 4–3 and 4–4 show the result of the HTML front end and the servlet, respectively.

Listing 4.3 ShowParameters.java

```
package coreservlets;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.*;

/** Shows all the parameters sent to the servlet via either
 *  GET or POST. Specially marks parameters that have
 *  no values or multiple values.
 */
```

Listing 4.3 ShowParameters.java *(continued)*

```
public class ShowParameters extends HttpServlet {
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    String docType =
    "<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.0 " +
    "Transitional//EN\">\n";
    String title = "Reading All Request Parameters";
    out.println(docType +
                "<HTML>\n" +
                "<HEAD><TITLE>" + title + "</TITLE></HEAD>\n" +
                "<BODY BGCOLOR=\"#FDF5E6\">\n" +
                "<H1 ALIGN=CENTER>" + title + "</H1>\n" +
                "<TABLE BORDER=1 ALIGN=CENTER>\n" +
                "<TR BGCOLOR=\"#FFAD00\">\n" +
                "<TH>Parameter Name<TH>Parameter Value(s)");
    Enumeration paramNames = request.getParameterNames();
    while(paramNames.hasMoreElements()) {
      String paramName = (String)paramNames.nextElement();
      out.print("<TR><TD>" + paramName + "\n<TD>");
      String[] paramValues =
        request.getParameterValues(paramName);
      if (paramValues.length == 1) {
        String paramValue = paramValues[0];
        if (paramValue.length() == 0)
          out.println("<I>No Value</I>");
        else
          out.println(paramValue);
      } else {
        out.println("<UL>");
        for(int i=0; i<paramValues.length; i++) {
          out.println("<LI>" + paramValues[i]);
        }
        out.println("</UL>");
      }
    }
    out.println("</TABLE>\n</BODY></HTML>");
  }

  public void doPost(HttpServletRequest request,
                     HttpServletResponse response)
      throws ServletException, IOException {
    doGet(request, response);
  }
}
```

Notice that the servlet uses a `doPost` method that simply calls `doGet`. That's because we want it to be able to handle *both* `GET` and `POST` requests. This approach is a good standard practice if you want HTML interfaces to have some flexibility in how they send data to the servlet. See the discussion of the `service` method in Section 3.6 (The Servlet Life Cycle) for a discussion of why having `doPost` call `doGet` (or vice versa) is preferable to overriding `service` directly. The HTML form from Listing 4.4 uses `POST`, as should *all* forms that have password fields (for details, see Chapter 19, "Creating and Processing HTML Forms"). However, the `ShowParameters` servlet is not specific to that particular front end, so the source code archive site at http://www.coreservlets.com/ includes a similar HTML form that uses `GET` for you to experiment with.

Listing 4.4 ShowParametersPostForm.html

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML><HEAD><TITLE>A Sample FORM using POST</TITLE></HEAD>
<BODY BGCOLOR="#FDF5E6">
<H1 ALIGN="CENTER">A Sample FORM using POST</H1>
<FORM ACTION="/servlet/coreservlets.ShowParameters"
      METHOD="POST">
  Item Number: <INPUT TYPE="TEXT" NAME="itemNum"><BR>
  Description: <INPUT TYPE="TEXT" NAME="description"><BR>
  Price Each: <INPUT TYPE="TEXT" NAME="price" VALUE="$"><BR>
  <HR>
  First Name: <INPUT TYPE="TEXT" NAME="firstName"><BR>
  Last Name: <INPUT TYPE="TEXT" NAME="lastName"><BR>
  Middle Initial: <INPUT TYPE="TEXT" NAME="initial"><BR>
  Shipping Address:
  <TEXTAREA NAME="address" ROWS=3 COLS=40></TEXTAREA><BR>
  Credit Card:<BR>
    <INPUT TYPE="RADIO" NAME="cardType"
                     VALUE="Visa">Visa<BR>
    <INPUT TYPE="RADIO" NAME="cardType"
                     VALUE="MasterCard">MasterCard<BR>
    <INPUT TYPE="RADIO" NAME="cardType"
                     VALUE="Amex">American Express<BR>
    <INPUT TYPE="RADIO" NAME="cardType"
                     VALUE="Discover">Discover<BR>
    <INPUT TYPE="RADIO" NAME="cardType"
                     VALUE="Java SmartCard">Java SmartCard<BR>
```

Listing 4.4 ShowParametersPostForm.html *(continued)*

```
  Credit Card Number:
  <INPUT TYPE="PASSWORD" NAME="cardNum"><BR>
  Repeat Credit Card Number:
  <INPUT TYPE="PASSWORD" NAME="cardNum"><BR><BR>
  <CENTER><INPUT TYPE="SUBMIT" VALUE="Submit Order"></CENTER>
</FORM>
</BODY></HTML>
```

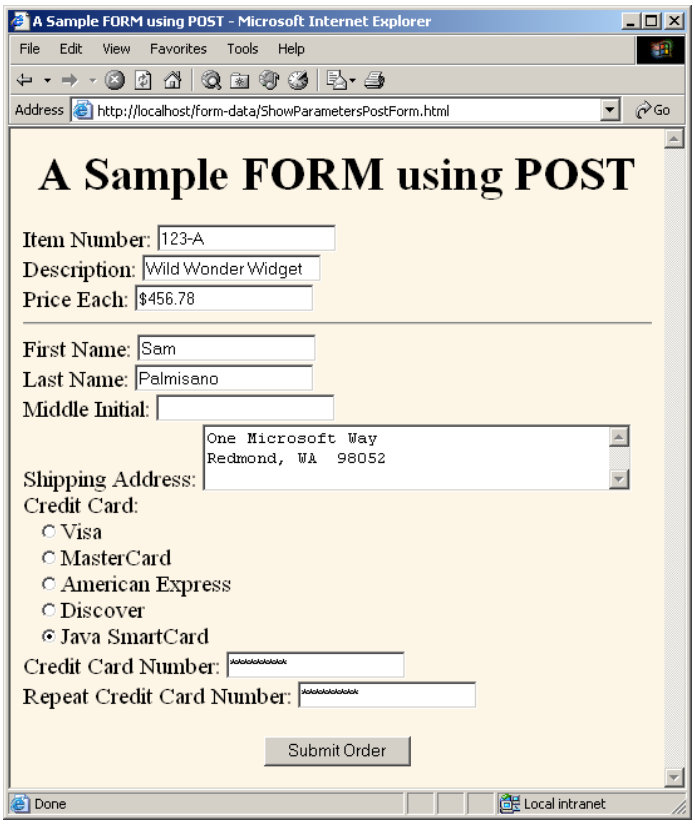


Figure 4–3 HTML form that collects data for the `ShowParameters` servlet.

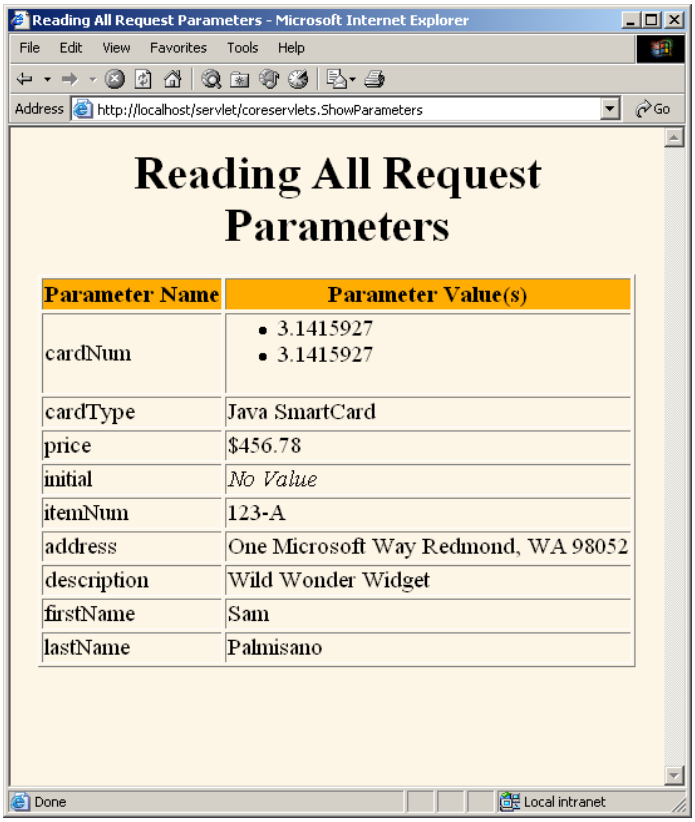
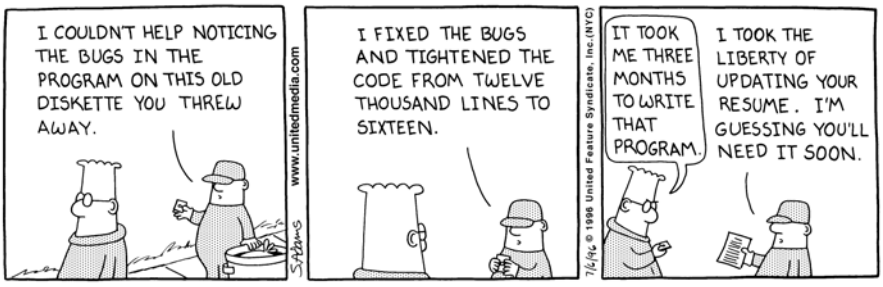


Figure 4–4 Result of the `ShowParameters` servlet.

4.5 Using Default Values When Parameters Are Missing or Malformed

Online job services have become increasingly popular of late. A reputable site provides a useful service to job seekers by giving their skills wide exposure and provides a useful service to employers by giving them access to a large pool of prospective employees. This section presents a servlet that handles part of such a site: the creation of online résumés from user-submitted data. Now, the question is: what should the servlet do when the user fails to supply the necessary information? This question has two answers: use default values or redisplay the form (prompting the user for missing values). This section illustrates the use of default values; Section 4.8 illustrates redisplay of the form.



DILBERT reprinted by permission of United Feature Syndicate, Inc.

When examining request parameters, you need to check for three conditions:

1. **The value is `null`.** A call to `request.getParameter` returns `null` if the form contains no textfield or other element of the expected name so that the parameter name does not appear in the request at all. This can happen when the end user uses an incorrect HTML form or when a bookmarked URL containing `GET` data is used but the parameter names have changed since the URL was bookmarked. To avoid a `NullPointerException`, you have to check for `null` *before* you try to call any methods on the string that results from `getParameter`.
2. **The value is an empty string.** A call to `request.getParameter` returns an empty string (i.e., `""`) if the associated textfield is empty when the form is submitted. To check for an empty string, compare the string to `""` by using `equals` or compare the length of the string to 0. *Do not use the == operator*; in the Java programming language, `==` always tests whether the two arguments are the same object (at the same memory location), not whether the two objects look similar. Just to be safe, it is also a good idea to call `trim` to remove any white space that the user may have entered, since in most scenarios you want to treat pure white space as missing data. So, for example, a test for missing values might look like the following.

   ```
   String param = request.getParameter("someName");
   if ((param == null) || (param.trim().equals(""))) {
     doSomethingForMissingValues(...);
   } else {
     doSomethingWithParameter(param);
   }
   ```

3. **The value is a nonempty string of the wrong format.** What defines the wrong format is application specific: you might expect certain textfields to contain only numeric values, others to have exactly seven characters, and others to only contain single letters.

Note that the use of JavaScript for client-side validation does not remove the need for also doing this type of checking on the server. After all, you are responsible for the server-side application, and often another developer or group is responsible for the forms. You do not want *your* application to crash if *they* fail to detect every type of illegal input. Besides, clients can use their own HTML forms, can manually edit URLs that contain `GET` data, and can disable JavaScript.

Core Approach

Design your servlets to gracefully handle parameters that are missing (`null` or empty string) or improperly formatted. Test your servlets with missing and malformed data as well as with data in the expected format.

Listing 4.5 and Figure 4–5 show the HTML form that acts as the front end to the résumé-processing servlet. If you are not familiar with HTML forms, see Chapter 19. The form uses `POST` to submit the data and it gathers values for various parameter names. The important thing to understand here is what the servlet does with missing and malformed data. This process is summarized in the following list. Listing 4.6 shows the complete servlet code.

- **`name`**, **`title`**, **`email`**, **`languages`**, **`skills`**
  These parameters specify various parts of the résumé. Missing values should be replaced by default values specific to the parameter. The servlet uses a method called `replaceIfMissing` to accomplish this task.
- **`fgColor`**, **`bgColor`**
  These parameters give the colors of the foreground and background of the page. Missing values should result in black for the foreground and white for the background. The servlet again uses `replaceIfMissing` to accomplish this task.
- **`headingFont`**, **`bodyFont`**
  These parameters designate the font to use for headings and the main text, respectively. A missing value or a value of "default" should result in a sans-serif font such as Arial or Helvetica. The servlet uses a method called `replaceIfMissingOrDefault` to accomplish this task.
- **`headingSize`**, **`bodySize`**
  These parameters specify the point size for main headings and body text, respectively. Subheadings will be displayed in a slightly smaller size than the main headings. Missing values or nonnumeric values should result in a default size (32 for headings, 18 for body). The servlet uses a call to `Integer.parseInt` and a `try/catch` block for `NumberFormatException` to handle this case.

Listing 4.5 SubmitResume.html

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML><HEAD><TITLE>Free Resume Posting</TITLE>
  <LINK REL=STYLESHEET
        HREF="jobs-site-styles.css"
        TYPE="text/css">
</HEAD>
<BODY>
<H1>hot-computer-jobs.com</H1>
<P CLASS="LARGER">
To use our <I>free</I> resume-posting service, simply fill
out the brief summary of your skills below. Use "Preview"
to check the results, then press "Submit" once it is
ready. Your mini-resume will appear online within 24 hours.</P>
<HR>
<FORM ACTION="/servlet/coreservlets.SubmitResume"
      METHOD="POST">
<DL>
<DT><B>First, give some general information about the look of
your resume:</B>
<DD>Heading font:
    <INPUT TYPE="TEXT" NAME="headingFont" VALUE="default">
<DD>Heading text size:
    <INPUT TYPE="TEXT" NAME="headingSize" VALUE=32>
<DD>Body font:
    <INPUT TYPE="TEXT" NAME="bodyFont" VALUE="default">
<DD>Body text size:
    <INPUT TYPE="TEXT" NAME="bodySize" VALUE=18>
<DD>Foreground color:
    <INPUT TYPE="TEXT" NAME="fgColor" VALUE="BLACK">
<DD>Background color:
    <INPUT TYPE="TEXT" NAME="bgColor" VALUE="WHITE">

<DT><B>Next, give some general information about yourself:</B>
<DD>Name: <INPUT TYPE="TEXT" NAME="name">
<DD>Current or most recent title:
    <INPUT TYPE="TEXT" NAME="title">
<DD>Email address: <INPUT TYPE="TEXT" NAME="email">
<DD>Programming Languages:
    <INPUT TYPE="TEXT" NAME="languages">

<DT><B>Finally, enter a brief summary of your skills and
    experience:</B> (use &lt;P&gt; to separate paragraphs.
    Other HTML markup is also permitted.)
<DD><TEXTAREA NAME="skills"
              ROWS=10 COLS=60 WRAP="SOFT"></TEXTAREA>
```

Listing 4.5 SubmitResume.html *(continued)*

```
</DL>
  <CENTER>
    <INPUT TYPE="SUBMIT" NAME="previewButton" Value="Preview">
    <INPUT TYPE="SUBMIT" NAME="submitButton" Value="Submit">
  </CENTER>
</FORM>
<HR>
<P CLASS="TINY">See our privacy policy
<A HREF="we-will-spam-you.html">here</A>.</P>
</BODY></HTML>
```

Figure 4–5 Front end to résumé-previewing servlet.

Listing 4.6 SubmitResume.java

```
package coreservlets;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.*;

/** Servlet that handles previewing and storing resumes
 *  submitted by job applicants.
 */

public class SubmitResume extends HttpServlet {
  public void doPost(HttpServletRequest request,
                     HttpServletResponse response)
      throws ServletException, IOException {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    if (request.getParameter("previewButton") != null) {
      showPreview(request, out);
    } else {
      storeResume(request);
      showConfirmation(request, out);
    }
  }

  /** Shows a preview of the submitted resume. Takes
   *  the font information and builds an HTML
   *  style sheet out of it, then takes the real
   *  resume information and presents it formatted with
   *  that style sheet.
   */

  private void showPreview(HttpServletRequest request,
                           PrintWriter out) {
    String headingFont = request.getParameter("headingFont");
    headingFont = replaceIfMissingOrDefault(headingFont, "");
    int headingSize =
      getSize(request.getParameter("headingSize"), 32);
    String bodyFont = request.getParameter("bodyFont");
    bodyFont = replaceIfMissingOrDefault(bodyFont, "");
    int bodySize =
      getSize(request.getParameter("bodySize"), 18);
    String fgColor = request.getParameter("fgColor");
    fgColor = replaceIfMissing(fgColor, "BLACK");
    String bgColor = request.getParameter("bgColor");
```

Listing 4.6 SubmitResume.java *(continued)*

```
  bgColor = replaceIfMissing(bgColor, "WHITE");
  String name = request.getParameter("name");
  name = replaceIfMissing(name, "Lou Zer");
  String title = request.getParameter("title");
  title = replaceIfMissing(title, "Loser");
  String email = request.getParameter("email");
  email =
    replaceIfMissing(email, "contact@hot-computer-jobs.com");
  String languages = request.getParameter("languages");
  languages = replaceIfMissing(languages, "<I>None</I>");
  String languageList = makeList(languages);
  String skills = request.getParameter("skills");
  skills = replaceIfMissing(skills, "Not many, obviously.");
  out.println
    (ServletUtilities.DOCTYPE + "\n" +
     "<HTML><HEAD><TITLE>Resume for " + name + "</TITLE>\n" +
     makeStyleSheet(headingFont, headingSize,
                    bodyFont, bodySize,
                    fgColor, bgColor) + "\n" +
     "</HEAD>\n" +
     "<BODY>\n" +
     "<CENTER>\n"+
     "<SPAN CLASS=\"HEADING1\">" + name + "</SPAN><BR>\n" +
     "<SPAN CLASS=\"HEADING2\">" + title + "<BR>\n" +
     "<A HREF=\"mailto:" + email + "\">" + email +
         "</A></SPAN>\n" +
     "</CENTER><BR><BR>\n" +
     "<SPAN CLASS=\"HEADING3\">Programming Languages" +
     "</SPAN>\n" +
     makeList(languages) + "<BR><BR>\n" +
     "<SPAN CLASS=\"HEADING3\">Skills and Experience" +
     "</SPAN><BR><BR>\n" +
     skills + "\n" +
     "</BODY></HTML>");
}

/** Builds a cascading style sheet with information
 *  on three levels of headings and overall
 *  foreground and background cover. Also tells
 *  Internet Explorer to change color of mailto link
 *  when mouse moves over it.
 */
```

Listing 4.6 SubmitResume.java *(continued)*

```
private String makeStyleSheet(String headingFont,
                              int heading1Size,
                              String bodyFont,
                              int bodySize,
                              String fgColor,
                              String bgColor) {
  int heading2Size = heading1Size*7/10;
  int heading3Size = heading1Size*6/10;
  String styleSheet =
    "<STYLE TYPE=\"text/css\">\n" +
    "<!--\n" +
    ".HEADING1 { font-size: " + heading1Size + "px;\n" +
    "            font-weight: bold;\n" +
    "            font-family: " + headingFont +
                  "Arial, Helvetica, sans-serif;\n" +
    "}\n" +
    ".HEADING2 { font-size: " + heading2Size + "px;\n" +
    "            font-weight: bold;\n" +
    "            font-family: " + headingFont +
                  "Arial, Helvetica, sans-serif;\n" +
    "}\n" +
    ".HEADING3 { font-size: " + heading3Size + "px;\n" +
    "            font-weight: bold;\n" +
    "            font-family: " + headingFont +
                  "Arial, Helvetica, sans-serif;\n" +
    "}\n" +
    "BODY { color: " + fgColor + ";\n" +
    "       background-color: " + bgColor + ";\n" +
    "       font-size: " + bodySize + "px;\n" +
    "       font-family: " + bodyFont +
                  "Times New Roman, Times, serif;\n" +
    "}\n" +
    "A:hover { color: red; }\n" +
    "-->\n" +
    "</STYLE>";
  return(styleSheet);
}

/** Replaces null strings (no such parameter name) or
 *  empty strings (e.g., if textfield was blank) with
 *  the replacement. Returns the original string otherwise.
 */
```

Listing 4.6 SubmitResume.java *(continued)*

```
private String replaceIfMissing(String orig,
                                 String replacement) {
  if ((orig == null) || (orig.trim().equals(""))) {
    return(replacement);
  } else {
    return(orig);
  }
}

// Replaces null strings, empty strings, or the string
// "default" with the replacement.
// Returns the original string otherwise.

private String replaceIfMissingOrDefault(String orig,
                                         String replacement) {
  if ((orig == null) ||
      (orig.trim().equals("")) ||
      (orig.equals("default"))) {
    return(replacement);
  } else {
    return(orig + ", ");
  }
}

// Takes a string representing an integer and returns it
// as an int. Returns a default if the string is null
// or in an illegal format.

private int getSize(String sizeString, int defaultSize) {
  try {
    return(Integer.parseInt(sizeString));
  } catch(NumberFormatException nfe) {
    return(defaultSize);
  }
}

// Given "Java,C++,Lisp", "Java C++ Lisp" or
// "Java, C++, Lisp", returns
// "<UL>
//   <LI>Java
//   <LI>C++
//   <LI>Lisp
//  </UL>"
```

Listing 4.6 SubmitResume.java *(continued)*

```
  private String makeList(String listItems) {
    StringTokenizer tokenizer =
      new StringTokenizer(listItems, ", ");
    String list = "<UL>\n";
    while(tokenizer.hasMoreTokens()) {
      list = list + "  <LI>" + tokenizer.nextToken() + "\n";
    }
    list = list + "</UL>";
    return(list);
  }

  /** Shows a confirmation page when the user clicks the
   *  "Submit" button.
   */

  private void showConfirmation(HttpServletRequest request,
                                PrintWriter out) {
    String title = "Submission Confirmed.";
    out.println(ServletUtilities.headWithTitle(title) +
                "<BODY>\n" +
                "<H1>" + title + "</H1>\n" +
                "Your resume should appear online within\n" +
                "24 hours. If it doesn't, try submitting\n" +
                "again with a different email address.\n" +
                "</BODY></HTML>");
  }

  /** Why it is bad to give your email address to
   *  untrusted sites.
   */

  private void storeResume(HttpServletRequest request) {
    String email = request.getParameter("email");
    putInSpamList(email);
  }

  private void putInSpamList(String emailAddress) {
    // Code removed to protect the guilty.
  }
}
```

Once the servlet has meaningful values for each of the font and color parameters, it builds a cascading style sheet out of them. Style sheets are a standard way of specifying the font faces, font sizes, colors, indentation, and other formatting information in an HTML 4.0 Web page. Style sheets are usually placed in a separate file so that several

Web pages at a site can share the same style sheet, but in this case it is more convenient to embed the style information directly in the page by use of the `STYLE` element. For more information on style sheets, see http://www.w3.org/TR/REC-CSS1.

After creating the style sheet, the servlet places the job applicant's name, job title, and email address centered under each other at the top of the page. The heading font is used for these lines, and the email address is placed inside a `mailto:` hypertext link so that prospective employers can contact the applicant directly by clicking on the address. The programming languages specified in the `languages` parameter are parsed by `StringTokenizer` (assuming spaces or commas are used to separate the language names) and placed in a bulleted list beneath a "Programming Languages" heading. Finally, the text from the `skills` parameter is placed at the bottom of the page beneath a "Skills and Experience" heading.

Figures 4–6 and 4–7 show results when the required data is supplied and omitted, respectively. Figure 4–8 shows the result of clicking Submit instead of Preview.

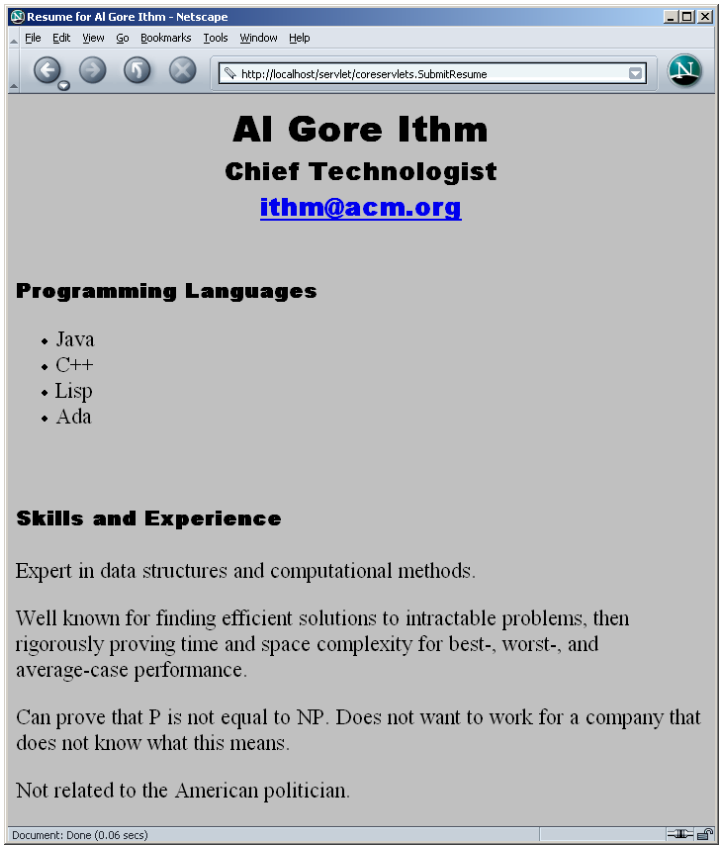


Figure 4–6 Preview of a résumé submission that contained the required form data.

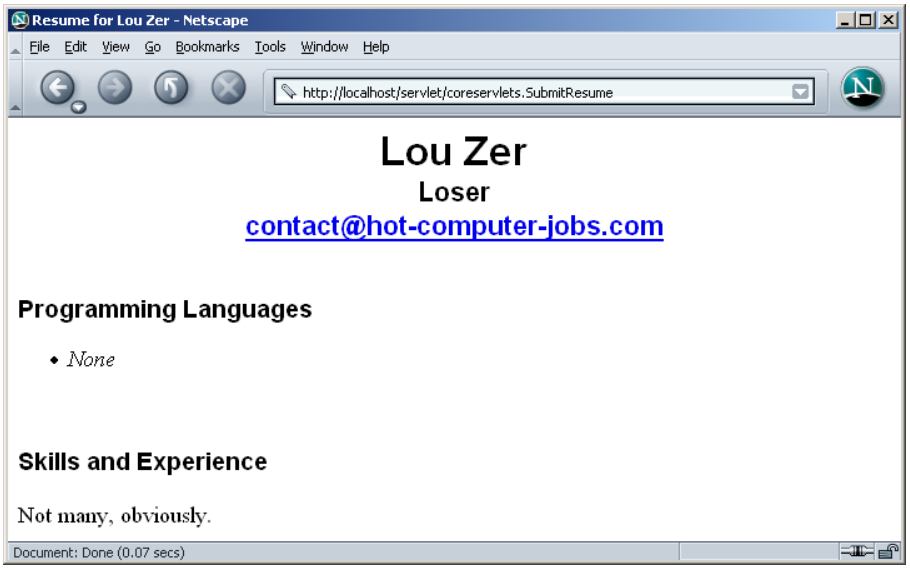


Figure 4–7 Preview of a submission that was missing much of the required data: default values replace the omitted values.

Figure 4–8 Result of submitting the résumé to the database.

4.6 Filtering Strings for HTML-Specific Characters

Normally, when a servlet wants to generate HTML that will contain characters like < or >, it simply uses `<` or `>`, the standard HTML character entities. Similarly, if a servlet wants a double quote or an ampersand to appear inside an HTML attribute value, it uses `&quot;` or `&`. Failing to make these substitutions results in malformed HTML code, since < or > will often be interpreted as part of an HTML markup tag, a double quote in an attribute value may be interpreted as the end of the value, and ampersands are just plain illegal in attribute values. In most cases, it is easy to note the special characters and use the standard HTML replacements. However, there are two cases in which it is not so easy to make this substitution manually.

The first case in which manual conversion is difficult occurs when the string is derived from a program excerpt or another source in which it is already in some standard format. Going through manually and changing all the special characters can be tedious in such a case, but forgetting to convert even one special character can result in your Web page having missing or improperly formatted sections.

The second case in which manual conversion fails is when the string is derived from HTML form data. Here, the conversion absolutely must be performed at runtime, since of course the query data is not known at compile time. If the user accidentally or deliberately enters HTML tags, the generated Web page will contain spurious HTML tags and can have completely unpredictable results (the HTML specification tells browsers what to do with legal HTML; it says nothing about what they should do with HTML containing illegal syntax).

Core Approach

If you read request parameters and display their values in the resultant page, you should filter out the special HTML characters. Failing to do so can result in output that has missing or oddly formatted sections.

Failing to do this filtering for externally accessible Web pages also lets your page become a vehicle for the *cross-site scripting attack*. Here, a malicious programmer embeds `GET` parameters in a URL that refers to one of your servlets (or any other server-side program). These `GET` parameters expand to HTML tags (usually `<SCRIPT>` elements) that exploit known browser bugs. So, an attacker could embed the code in a URL that refers to your site and distribute only the URL, not the malicious Web page itself. That way, the attacker can remain undiscovered more easily and can also exploit

trusted relationships to make users think the scripts are coming from a trusted source (your organization). For more details on this issue, see http://www.cert.org/advisories/CA-2000-02.html and http://www.microsoft.com/technet/security/topics/ExSumCS.asp.

Code for Filtering

Replacing <, >, ", and & in strings is a simple matter, and a number of different approaches can accomplish the task. However, it is important to remember that Java strings are immutable (i.e., can't be modified), so repeated string concatenation involves copying and then discarding many string segments. For example, consider the following two lines:

```
String s1 = "Hello";
String s2 = s1 + " World";
```

Since `s1` cannot be modified, the second line makes a copy of `s1` and appends `"World"` to the copy, then the copy is discarded. To avoid the expense of generating and copying these temporary objects, whenever you perform repeated concatenation within a loop, you should use a mutable data structure; and `StringBuffer` is the natural choice.

Core Approach

If you do string concatenation from within a loop, use `StringBuffer`, not `String`.

Listing 4.7 shows a static `filter` method that uses a `StringBuffer` to efficiently copy characters from an input string to a filtered version, replacing the four special characters along the way.

Listing 4.7 ServletUtilities.java (Excerpt)

```
package coreservlets;
import javax.servlet.*;
import javax.servlet.http.*;

public class ServletUtilities {
  ...

  /** Replaces characters that have special HTML meanings
   *  with their corresponding HTML character entities.
   */
```

Listing 4.7 ServletUtilities.java (Excerpt) *(continued)*

```
  // Note that Javadoc is not used for the more detailed
  // documentation due to the difficulty of making the
  // special chars readable in both plain text and HTML.
  //
  //  Given a string, this method replaces all occurrences of
  //  '<' with '&lt;', all occurrences of '>' with
  //  '&gt;', and (to handle cases that occur inside attribute
  //  values), all occurrences of double quotes with
  //  '&quot;' and all occurrences of '&' with '&amp;'.
  //  Without such filtering, an arbitrary string
  //  could not safely be inserted in a Web page.

  public static String filter(String input) {
    if (!hasSpecialChars(input)) {
      return(input);
    }
    StringBuffer filtered = new StringBuffer(input.length());
    char c;
    for(int i=0; i<input.length(); i++) {
      c = input.charAt(i);
      switch(c) {
        case '<': filtered.append("&lt;"); break;
        case '>': filtered.append("&gt;"); break;
        case '"': filtered.append("&quot;"); break;
        case '&': filtered.append("&amp;"); break;
        default: filtered.append(c);
      }
    }
    return(filtered.toString());
  }

  private static boolean hasSpecialChars(String input) {
    boolean flag = false;
    if ((input != null) && (input.length() > 0)) {
      char c;
      for(int i=0; i<input.length(); i++) {
        c = input.charAt(i);
        switch(c) {
          case '<': flag = true; break;
          case '>': flag = true; break;
          case '"': flag = true; break;
          case '&': flag = true; break;
        }
      }
    }
    return(flag);
  }
}
```

Example: A Servlet That Displays Code Snippets

As an example, consider the HTML form of Listing 4.8 that gathers a snippet of the Java programming language and sends it to the servlet of Listing 4.9 for display.

Now, when the user enters normal input, the result is fine, as illustrated by Figure 4–9. However, as shown in Figure 4–10, the result can be unpredictable when the input contains special characters like < and >. Different browsers can give different results since the HTML specification doesn't say what to do in this case, but most browsers think that the "<b" in "`if (a<b) {`" starts an HTML tag. But, since the characters after the `b` are unrecognized, browsers ignore them until the next >, which is at the end of the `</PRE>` tag. Thus, not only does most of the code snippet disappear, but the browser does not interpret the `</PRE>` tag, so the text after the code snippet is improperly formatted, with a fixed-width font and line wrapping disabled.

Listing 4.10 shows a servlet that works exactly like the previous one except that it filters the special characters from the request parameter value before displaying it. Listing 4.11 shows an HTML form that sends data to it (except for the `ACTION` URL, this form is identical to that shown in Listing 4.8). Figure 4–11 shows the result of the input that failed for the previous servlet: no problem here.

Listing 4.8 CodeForm1.html

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML><HEAD><TITLE>Submit Code Sample</TITLE></HEAD>
<BODY BGCOLOR="#FDF5E6">
<CENTER>
<H1 ALIGN="CENTER">Submit Code Sample</H1>
<FORM ACTION="/servlet/coreservlets.BadCodeServlet">
  Code:<BR>
  <TEXTAREA ROWS="6" COLS="40" NAME="code"></TEXTAREA><P>
  <INPUT TYPE="SUBMIT" VALUE="Submit Code">
</FORM>
</CENTER></BODY></HTML>
```

Listing 4.9 BadCodeServlet.java

```
package coreservlets;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

/** Servlet that reads a code snippet from the request
 *  and displays it inside a PRE tag. Fails to filter
 *  the special HTML characters.
 */

public class BadCodeServlet extends HttpServlet {
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    String title = "Code Sample";
    String docType =
      "<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.0 " +
      "Transitional//EN\">\n";
    out.println(docType +
                "<HTML>\n" +
                "<HEAD><TITLE>" + title + "</TITLE></HEAD>\n" +
                "<BODY BGCOLOR=\"#FDF5E6\">\n" +
                "<H1 ALIGN=\"CENTER\">" + title + "</H1>\n" +
                "<PRE>\n" +
                getCode(request) +
                "</PRE>\n" +
                "Now, wasn't that an interesting sample\n" +
                "of code?\n" +
                "</BODY></HTML>");
  }

  protected String getCode(HttpServletRequest request) {
    return(request.getParameter("code"));
  }
}
```

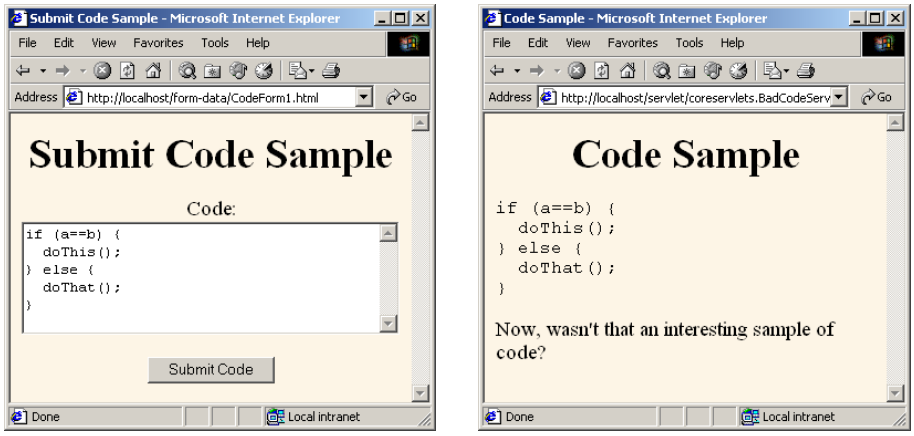


Figure 4–9 `BadCodeServlet`: result is fine when request parameters contain no special characters.

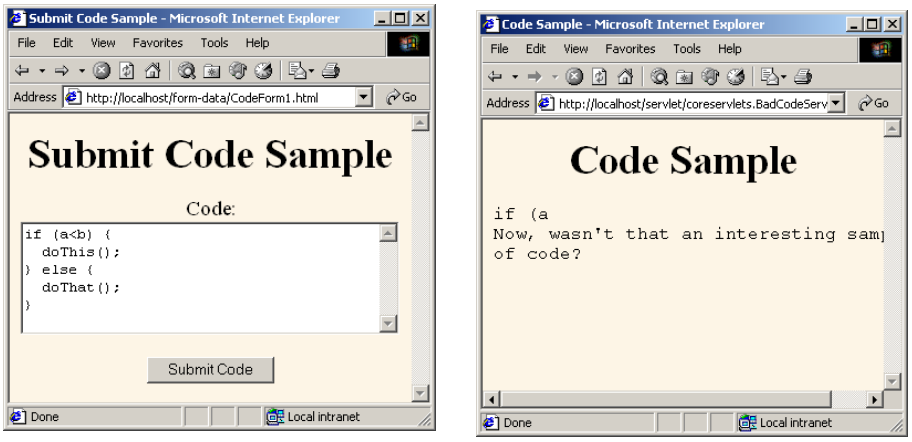


Figure 4–10 `BadCodeServlet`: result has missing and incorrectly formatted sections when request parameters contain special characters.

Listing 4.10 GoodCodeServlet.java

```
package coreservlets;

import javax.servlet.http.*;

/** Servlet that reads a code snippet from the request and displays
 *  it inside a PRE tag. Filters the special HTML characters.
 */

public class GoodCodeServlet extends BadCodeServlet {
  protected String getCode(HttpServletRequest request) {
    return(ServletUtilities.filter(super.getCode(request)));
  }
}
```

Listing 4.11 CodeForm2.html

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML><HEAD><TITLE>Submit Code Sample</TITLE></HEAD>
<BODY BGCOLOR="#FDF5E6">
<CENTER>
<H1 ALIGN="CENTER">Submit Code Sample</H1>
<FORM ACTION="/servlet/coreservlets.GoodCodeServlet">
  Code:<BR>
  <TEXTAREA ROWS="6" COLS="40" NAME="code"></TEXTAREA><P>
  <INPUT TYPE="SUBMIT" VALUE="Submit Code">
</FORM>
</CENTER></BODY></HTML>
```

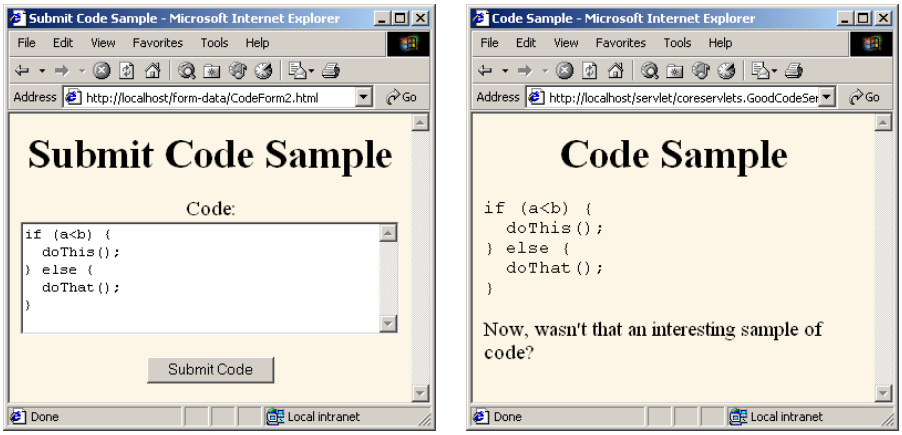


Figure 4–11 `GoodCodeServlet`: result is fine even when request parameters contain special characters.

4.7 Automatically Populating Java Objects from Request Parameters: Form Beans

The `getParameter` method makes it easy to read incoming request parameters: you use the same method from `doGet` as from `doPost`, and the value returned is automatically URL-decoded (i.e., in the format that the end user typed it in, not necessarily in the format in which it was sent over the network). Since the return value of `getParameter` is `String`, however, you have to parse the value yourself (checking for missing or malformed data, of course) if you want other types of values. For example, if you expect `int` or `double` values, you have to pass the result of `getParameter` to `Integer.parseInt` or `Double.parseDouble` and enclose the code inside a `try/catch` block that looks for `NumberFormatException`. If you have many request parameters, this procedure can be quite tedious.

For example, suppose that you have a data object with three `String` fields, two `int` fields, two `double` fields, and a `boolean`. Filling in the object based on a form submission would require eight separate calls to `getParameter`, two calls each to `Integer.parseInt` and `Double.parseDouble`, and some special-purpose code to set the `boolean` flag. It would be nice to do this work automatically.

Now, in JSP, you can use the JavaBeans component architecture to greatly simplify the process of reading request parameters, parsing the values, and storing the results in Java objects. This process is discussed in detail in Chapter 14 (Using JavaBeans Components in JSP Documents). If you are unfamiliar with the idea of beans, refer to that chapter for details. The gist, though, is that an ordinary Java object is considered to be a *bean* if the class uses private fields and has methods that follow the get/set naming convention. The names of the methods (minus the word "get" or "set" and with the first character in lower case) are called *properties*. For example, an arbitrary Java class with a `getName` and `setName` method is said to define a bean that has a property called `name`.

As discussed in Chapter 14, there is special JSP syntax (`property="*"` in a `jsp:setProperty` call) that you can use to populate a bean in one fell swoop. Specifically, this setting indicates that the system should examine all incoming request parameters and pass them to bean properties that match the request parameter name. In particular, if the request parameter is named `param1`, the parameter is passed to the `setParam1` method of the object. Furthermore, simple type conversions are performed automatically. For instance, if there is a request parameter called `numOrdered` and the object has a method called `setNumOrdered` that expects an `int` (i.e., the bean has a `numOrdered` property of type `int`), the `numOrdered` request parameter is automatically converted to an `int` and the resulting value is automatically passed to the `setNumOrdered` method.

Now, if you can do this in JSP, you would think you could do it in servlets as well. After all, as discussed in Chapter 10, JSP pages are really servlets in disguise: each JSP page gets translated into a servlet, and it is the servlet that runs at request time. Furthermore, as we see in Chapter 15 (Integrating Servlets and JSP: The Model View Controller (MVC) Architecture), in complicated scenarios it is often best to combine servlets and JSP pages in such a way that the servlets do the programming work and the JSP pages do the presentation work. So, it is really more important for servlets to be able to read request parameters easily than it is for JSP pages to do so. Surprisingly, however, the servlet specification provides no such capability: the code behind the `property="*"` JSP process is not exposed through a standard API.

Fortunately, the widely used Jakarta Commons package (see http://jakarta.apache.org/commons/) from The Apache Software Foundation contains classes that make it easy to build a utility to automatically associate request parameters with bean properties (i.e., with `setXxx` methods). The next subsection provides information on obtaining the Commons packages, but the important point here is that a static `populateBean` method takes a bean (i.e., a Java object with at least some methods that follow the get/set naming convention) and a `Map` as input and passes all `Map` values to the bean property that matches the associated `Map` key name. This utility also does type conversion automatically, using default values (e.g., 0 for numeric values) instead of throwing exceptions when the corresponding request parameter is malformed. If the bean has no property matching the name, the `Map` entry is ignored; again, no exception is thrown.

Listing 4.12 presents a utility that uses the Jakarta Commons utility to automatically populate a bean according to incoming request parameters. To use it, simply pass the bean and the request object to `BeanUtilities.populateBean`. That's it! You want to put two request parameters into a data object? No problem: one method call is all that's needed. Fifteen request parameters plus type conversion? Same one method call.

Listing 4.12 BeanUtilities.java

```
package coreservlets.beans;

import java.util.*;
import javax.servlet.http.*;
import org.apache.commons.beanutils.BeanUtils;

/** Some utilities to populate beans, usually based on
 *  incoming request parameters. Requires three packages
 *  from the Apache Commons library: beanutils, collections,
 *  and logging. To obtain these packages, see
 *  http://jakarta.apache.org/commons/. Also, the book's
 *  source code archive (see http://www.coreservlets.com/)
```

Listing 4.12 BeanUtilities.java *(continued)*

```
 *  contains links to all URLs mentioned in the book, including
 *  to the specific sections of the Jakarta Commons package.
 *  <P>
 *  Note that this class is in the coreservlets.beans package,
 *  so must be installed in .../coreservlets/beans/.
 */

public class BeanUtilities {
  /** Examines all of the request parameters to see if
   *  any match a bean property (i.e., a setXxx method)
   *  in the object. If so, the request parameter value
   *  is passed to that method. If the method expects
   *  an int, Integer, double, Double, or any of the other
   *  primitive or wrapper types, parsing and conversion
   *  is done automatically. If the request parameter value
   *  is malformed (cannot be converted into the expected
   *  type), numeric properties are assigned zero and boolean
   *  properties are assigned false: no exception is thrown.
   */

  public static void populateBean(Object formBean,
                                  HttpServletRequest request) {
    populateBean(formBean, request.getParameterMap());
  }

  /** Populates a bean based on a Map: Map keys are the
   *  bean property names; Map values are the bean property
   *  values. Type conversion is performed automatically as
   *  described above.
   */

  public static void populateBean(Object bean,
                                  Map propertyMap) {
    try {
      BeanUtils.populate(bean, propertyMap);
    } catch(Exception e) {
      // Empty catch. The two possible exceptions are
      // java.lang.IllegalAccessException and
      // java.lang.reflect.InvocationTargetException.
      // In both cases, just skip the bean operation.
    }
  }
}
```

Putting BeanUtilities to Work

Listing 4.13 shows a servlet that gathers insurance information about an employee, presumably to use it to determine available insurance plans and associated costs. To perform this task, the servlet needs to fill in an insurance information data object (InsuranceInfo.java, Listing 4.14) with information on the employee's name and ID (both of type `String`), number of children (`int`), and whether or not the employee is married (`boolean`). Since this object is represented as a bean, `BeanUtilities.populateBean` can be used to fill in the required information with a single method call. Listing 4.15 shows the HTML form that gathers the data; Figures 4–12 and 4–13 show typical results.

Listing 4.13 SubmitInsuranceInfo.java

```
package coreservlets;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import coreservlets.beans.*;

/** Example of simplified form processing. Illustrates the
 *  use of BeanUtilities.populateBean to automatically fill
 *  in a bean (Java object with methods that follow the
 *  get/set naming convention) from request parameters.
 */

public class SubmitInsuranceInfo extends HttpServlet {
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {
    InsuranceInfo info = new InsuranceInfo();
    BeanUtilities.populateBean(info, request);
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    String docType =
      "<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.0 " +
      "Transitional//EN\">\n";
    String title = "Insurance Info for " + info.getName();
    out.println(docType +
                "<HTML>\n" +
                "<HEAD><TITLE>" + title + "</TITLE></HEAD>\n" +
                "<BODY BGCOLOR=\"#FDF5E6\">\n" +
                "<CENTER>\n" +
                "<H1>" + title + "</H1>\n" +
```

Listing 4.13 SubmitInsuranceInfo.java *(continued)*

```
                "<UL>\n" +
                "  <LI>Employee ID: " +
                   info.getEmployeeID() + "\n" +
                "  <LI>Number of children: " +
                   info.getNumChildren() + "\n" +
                "  <LI>Married?: " +
                   info.isMarried() + "\n" +
                "</UL></CENTER></BODY></HTML>");
  }
}
```

Listing 4.14 InsuranceInfo.java

```
package coreservlets.beans;

import coreservlets.*;

/** Simple bean that represents information needed to
 *  calculate an employee's insurance costs. Has String,
 *  int, and boolean properties. Used to demonstrate
 *  automatically filling in bean properties from request
 *  parameters.
 */

public class InsuranceInfo {
  private String name = "No name specified";
  private String employeeID = "No ID specified";
  private int numChildren = 0;
  private boolean isMarried = false;

  public String getName() {
    return(name);
  }

  /** Just in case user enters special HTML characters,
   *  filter them out before storing the name.
   */

  public void setName(String name) {
    this.name = ServletUtilities.filter(name);
  }

  public String getEmployeeID() {
    return(employeeID);
  }
```

Listing 4.14 InsuranceInfo.java *(continued)*

```
  /** Just in case user enters special HTML characters,
   *  filter them out before storing the name.
   */

  public void setEmployeeID(String employeeID) {
    this.employeeID = ServletUtilities.filter(employeeID);
  }

  public int getNumChildren() {
    return(numChildren);
  }

  public void setNumChildren(int numChildren) {
    this.numChildren = numChildren;
  }

  /** Bean convention: name getter method "isXxx" instead
   *  of "getXxx" for boolean methods.
   */

  public boolean isMarried() {
    return(isMarried);
  }

  public void setMarried(boolean isMarried) {
    this.isMarried = isMarried;
  }
}
```

Listing 4.15 InsuranceForm.html

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML><HEAD><TITLE>Employee Insurance Signup</TITLE></HEAD>
<BODY BGCOLOR="#FDF5E6">
<CENTER>
<H1>Employee Insurance Signup</H1>

<FORM ACTION="/servlet/coreservlets.SubmitInsuranceInfo">
  Name:  <INPUT TYPE="TEXT" NAME="name"><BR>
  Employee ID: <INPUT TYPE="TEXT" NAME="employeeID"><BR>
  Number of Children:  <INPUT TYPE="TEXT" NAME="numChildren"><BR>
  <INPUT TYPE="CHECKBOX" NAME="married" VALUE="true">Married?<BR>
  <CENTER><INPUT TYPE="SUBMIT"></CENTER>
</FORM>

</CENTER></BODY></HTML>
```

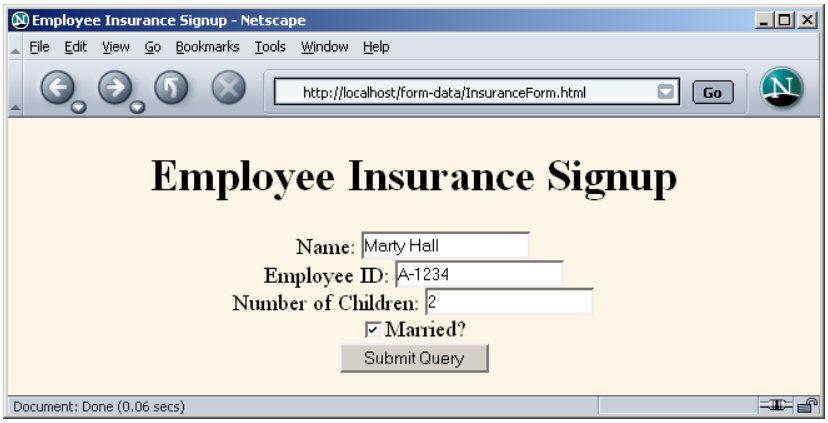


Figure 4–12 Front end to insurance-processing servlet.

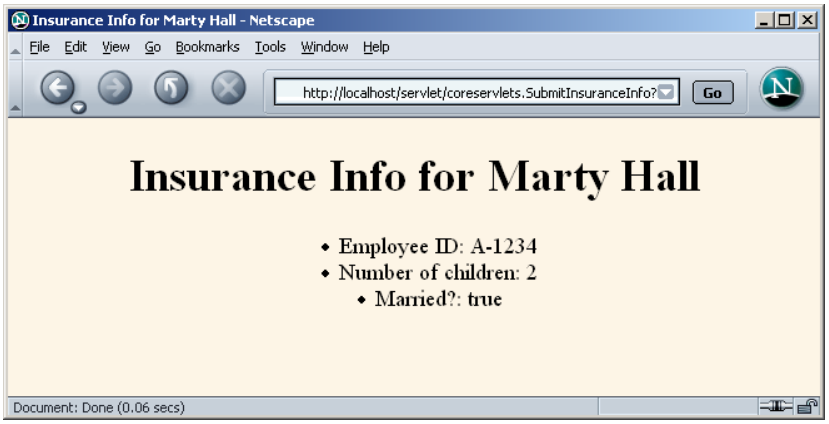


Figure 4–13 Insurance-processing servlet: the gathering of request data is greatly simplified by use of `BeanUtilities.populateBean`.

Obtaining and Installing the Jakarta Commons Packages

Most of the work of our `BeanUtilities` class is done by the Jakarta Commons `BeanUtils` component. This component performs the reflection (determination of what writable bean properties—`set`*Xxx* methods—the object has) and the type conversion (parsing a String as an `int`, `double`, `boolean`, or other primitive or wrapper type). So, `BeanUtilities` will not work unless you install the Jakarta

Commons `BeanUtils`. However, since `BeanUtils` depends on two other Jakarta Commons components—`Collections` and `Logging`—you have to download and install all three.

To download these components, start at http://jakarta.apache.org/commons/, look for the "Components Repository" heading in the left column, and, for each of the three components, download the JAR file for the latest version. (Our code is based on version 1.5 of the `BeanUtils`, but it is likely that any recent version will work identically.) Perhaps the easiest way to download the components is to go to http://www.coreservlets.com/, go to Chapter 4 of the source code archive, and look for the direct links to the three JAR files.

The most portable way to install the components is to follow the standard approach:

- For development, list the three JAR files in your `CLASSPATH`.
- For deployment, put the three JAR files in the WEB-INF/lib directory of your Web application.

When dealing with JAR files like these—used in multiple Web applications—many developers use server-specific features that support sharing of JAR files across Web applications. For example, Tomcat permits common JAR files to be placed in *tomcat_install_dir*/common/lib. Another shortcut that many people use on their development machines is to drop the three JAR files into *sdk_install_dir*/jre/lib/ext. Doing so makes the JAR files automatically accessible both to the development environment and to the locally installed server. These are both useful tricks as long as you remember that *your-web-app*/WEB-INF/lib is the only standardized location on the deployment server.

4.8 Redisplaying the Input Form When Parameters Are Missing or Malformed

It sometimes makes sense to use default values when the user fails to fill in certain form fields. Other times, however, there are no reasonable default values to use, and the form should be redisplayed to the user. Two desirable capabilities make the use of a normal HTML form impossible in this scenario:

- Users should not have to reenter values that they already supplied.
- Missing form fields should be marked prominently.

Redisplay Options

So, what can be done to implement these capabilities? Well, a full description of the possible approaches is a bit complicated and requires knowledge of several techniques (e.g., Struts, JSTL) that are not covered in Volume 1 of this book. So, you should refer to Volume 2 for details, but here is a quick preview.

- **Have the same servlet present the form, process the data, and present the results.** The servlet first looks for incoming request data: if it finds none, it presents a blank form. If the servlet finds partial request data, it extracts the partial data, puts it back into the form, and marks the other fields as missing. If the servlet finds the full complement of required data, it processes the request and displays the results. The form omits the `ACTION` attribute so that form submissions automatically go to the same URL as the form itself. This is the only approach for which we have already covered all of the necessary techniques, so this is the approach illustrated in this section.
- **Have one servlet present the form; have a second servlet process the data and present the results.** This option is better than the first since it divides up the labor and keeps each servlet smaller and more manageable. However, using this approach requires two techniques we have not yet covered: how to transfer control from one servlet to another and how to access user-specific data in one servlet that was created in another. Transferring from one servlet to another can be done with `response.sendRedirect` (see Chapter 6, "Generating the Server Response: HTTP Status Codes") or the `forward` method of `RequestDispatcher` (see Chapter 15, "Integrating Servlets and JSP: The Model View Controller (MVC) Architecture"). The easiest way to pass the data from the processing servlet back to the form-display servlet is to store it in the `HttpSession` object (see Chapter 9, "Session Tracking").
- **Have a JSP page "manually" present the form; have a servlet or JSP page process the data and present the results.** This is an excellent option, and it is widely used. However, it requires knowledge of JSP in addition to knowledge of the two techniques mentioned in the previous bullet (how to transfer the user from the data-processing servlet to the form page and how to use session tracking to store user-specific data). In particular, you need to know how to use JSP expressions (see Chapter 11, "Invoking Java Code with JSP Scripting Elements") or `jsp:getProperty` (see Chapter 14, "Using JavaBeans Components in JSP Documents") to extract the partial data from the data object and put it into the HTML form.

- **Have a JSP page present the form, automatically filling in the fields with values obtained from a data object. Have a servlet or JSP page process the data and present the results.** This is perhaps the best option of all. But, in addition to the techniques described in the previous bullet, it requires custom JSP tags that mimic HTML form elements but use designated values automatically. You can write these tags yourself or you can use ready-made versions such as those that come with JSTL or Apache Struts (see Volume 2 for coverage of custom tags, JSTL, and Struts).

A Servlet That Processes Auction Bids

To illustrate the first of the form-redisplay options, consider a servlet that processes bids at an auction site. Figures 4–14 through 4–16 show the desired outcome: the servlet initially displays a blank form, redisplays the form with missing data marked when partial data is submitted, and processes the request when complete data is submitted.

To accomplish this behavior, the servlet (Listing 4.16) performs the following steps.

1. Fills in a `BidInfo` object (Listing 4.17) from the request data, using `BeanUtilities.populateBean` (see Section 4.7, "Automatically Populating Java Objects from Request Parameters: Form Beans") to automatically match up request parameter names with bean properties and to perform simple type conversion.
2. Checks whether that `BidInfo` object is completely empty (no fields changed from the default). If so, it calls `showEntryForm` to display the initial input form.
3. Checks whether the `BidInfo` object is partially empty (some, but not all, fields changed from the default). If so, it calls `showEntryForm` to display the input form with a warning message and with the missing fields highlighted. Fields in which the user already entered data keep their previous values.
4. Checks whether the `BidInfo` object is completely filled in. If so, it calls `showBid` to process the data and present the result.

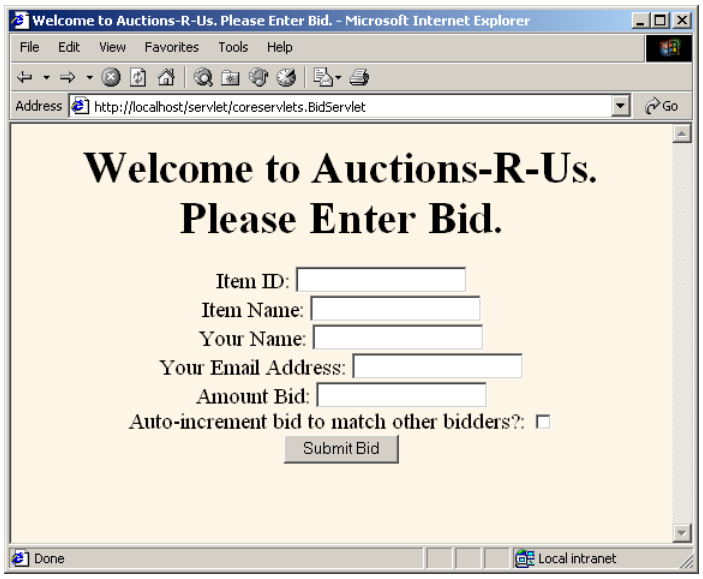


Figure 4–14 Original form of servlet: it presents a form to collect data about a bid at an auction.

Figure 4–15 Bid servlet with incomplete data. If the user submits a form that is not fully filled in, the bid servlet redisplays the form. The user's previous partial data is maintained, and missing fields are highlighted.

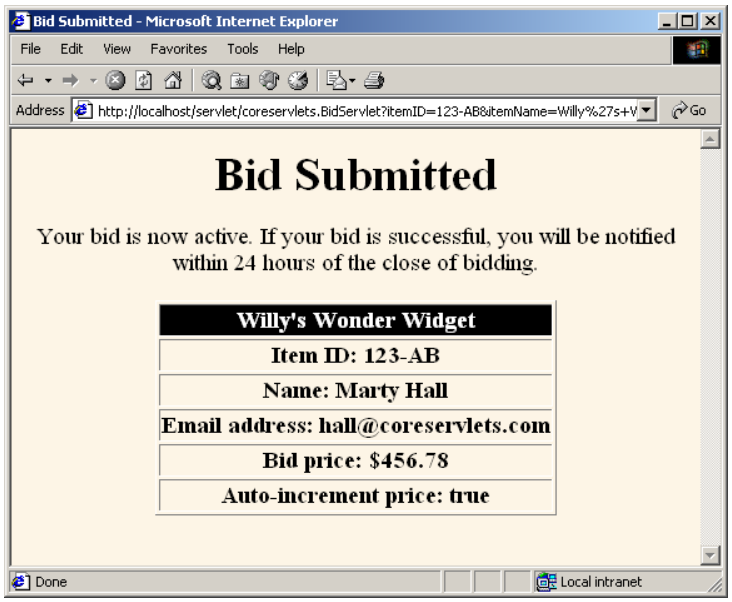


Figure 4–16 Bid servlet with complete data: it presents the results.

Listing 4.16 BidServlet.java

```
package coreservlets;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import coreservlets.beans.*;

/** Example of simplified form processing. Shows two features:
 *  <OL>
 *    <LI>Automatically filling in a bean based on the
 *        incoming request parameters.
 *    <LI>Using the same servlet both to generate the input
 *        form and to process the results. That way, when
 *        fields are omitted, the servlet can redisplay the
 *        form without making the user reenter previously
 *        entered values.
 *  </UL>
 */
```

Listing 4.16 BidServlet.java *(continued)*

```
public class BidServlet extends HttpServlet {

  /** Try to populate a bean that represents information
   *  in the form data sent by the user. If this data is
   *  complete, show the results. If the form data is
   *  missing or incomplete, display the HTML form
   *  that gathers the data.
   */

  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {
    BidInfo bid = new BidInfo();
    BeanUtilities.populateBean(bid, request);
    if (bid.isComplete()) {
      // All required form data was supplied: show result.
      showBid(request, response, bid);
    } else {
      // Form data was missing or incomplete: redisplay form.
      showEntryForm(request, response, bid);
    }
  }

  /** All required data is present: show the results page. */

  private void showBid(HttpServletRequest request,
                       HttpServletResponse response,
                       BidInfo bid)
      throws ServletException, IOException {
    submitBid(bid);
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    String title = "Bid Submitted";
    out.println
      (DOCTYPE +
       "<HTML>\n" +
       "<HEAD><TITLE>" + title + "</TITLE></HEAD>\n" +
       "<BODY BGCOLOR=\"#FDF5E6\"><CENTER>\n" +
       "<H1>" + title + "</H1>\n" +
       "Your bid is now active. If your bid is successful,\n" +
       "you will be notified within 24 hours of the close\n" +
       "of bidding.\n" +
       "<P>\n" +
       "<TABLE BORDER=1>\n" +
       "  <TR><TH BGCOLOR=\"BLACK\"><FONT COLOR=\"WHITE\">" +
       bid.getItemName() + "</FONT>\n" +
       "  <TR><TH>Item ID: " +
```

Listing 4.16 BidServlet.java *(continued)*

```
      bid.getItemID() + "\n" +
      "  <TR><TH>Name: " +
      bid.getBidderName() + "\n" +
      "  <TR><TH>Email address: " +
      bid.getEmailAddress() + "\n" +
      "  <TR><TH>Bid price: $" +
      bid.getBidPrice() + "\n" +
      "  <TR><TH>Auto-increment price: " +
      bid.isAutoIncrement() + "\n" +
      "</TABLE></CENTER></BODY></HTML>");
}

/** If the required data is totally missing, show a blank
 *  form. If the required data is partially missing,
 *  warn the user, fill in form fields that already have
 *  values, and prompt user for missing fields.
 */

private void showEntryForm(HttpServletRequest request,
                           HttpServletResponse response,
                           BidInfo bid)
    throws ServletException, IOException {
  boolean isPartlyComplete = bid.isPartlyComplete();
  response.setContentType("text/html");
  PrintWriter out = response.getWriter();
  String title =
    "Welcome to Auctions-R-Us. Please Enter Bid.";
  out.println
    (DOCTYPE +
     "<HTML>\n" +
     "<HEAD><TITLE>" + title + "</TITLE></HEAD>\n" +
     "<BODY BGCOLOR=\"#FDF5E6\"><CENTER>\n" +
     "<H1>" + title + "</H1>\n" +
     warning(isPartlyComplete) +
     "<FORM>\n" +
     inputElement("Item ID", "itemID",
                  bid.getItemID(), isPartlyComplete) +
     inputElement("Item Name", "itemName",
                  bid.getItemName(), isPartlyComplete) +
     inputElement("Your Name", "bidderName",
                  bid.getBidderName(), isPartlyComplete) +
     inputElement("Your Email Address", "emailAddress",
                  bid.getEmailAddress(), isPartlyComplete) +
     inputElement("Amount Bid", "bidPrice",
                  bid.getBidPrice(), isPartlyComplete) +
```

Listing 4.16 BidServlet.java *(continued)*

```
      checkbox("Auto-increment bid to match other bidders?",
               "autoIncrement", bid.isAutoIncrement()) +
      "<INPUT TYPE=\"SUBMIT\" VALUE=\"Submit Bid\">\n" +
      "</CENTER></BODY></HTML>");
}

private void submitBid(BidInfo bid) {
  // Some application-specific code to record the bid.
  // The point is that you pass in a real object with
  // properties populated, not a bunch of strings.
}

private String warning(boolean isFormPartlyComplete) {
  if(isFormPartlyComplete) {
    return("<H2>Required Data Missing! " +
           "Enter and Resubmit.</H2>\n");
  } else {
    return("");
  }
}

/** Create a textfield for input, prefaced by a prompt.
 *  If this particular textfield is missing a value but
 *  other fields have values (i.e., a partially filled form
 *  was submitted), then add a warning telling the user that
 *  this textfield is required.
 */

private String inputElement(String prompt,
                            String name,
                            String value,
                            boolean shouldPrompt) {
  String message = "";
  if (shouldPrompt && ((value == null) || value.equals(""))) {
    message = "<B>Required field!</B> ";
  }
  return(message + prompt + ": " +
         "<INPUT TYPE=\"TEXT\" NAME=\"" + name + "\"" +
         " VALUE=\"" + value + "\"><BR>\n");
}

private String inputElement(String prompt,
                            String name,
                            double value,
                            boolean shouldPrompt) {
```

Listing 4.16 BidServlet.java *(continued)*

```
    String num;
    if (value == 0.0) {
      num = "";
    } else {
      num = String.valueOf(value);
    }
    return(inputElement(prompt, name, num, shouldPrompt));
  }

  private String checkbox(String prompt,
                          String name,
                          boolean isChecked) {
    String result =
      prompt + ": " +
      "<INPUT TYPE=\"CHECKBOX\" NAME=\"" + name + "\"";
    if (isChecked) {
      result = result + " CHECKED";
    }
    result = result + "><BR>\n";
    return(result);
  }

  private final String DOCTYPE =
    "<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.0 " +
    "Transitional//EN\">\n";
}
```

Listing 4.17 BidInfo.java

```
package coreservlets.beans;

import coreservlets.*;

/** Bean that represents information about a bid at
 *  an auction site. Used to demonstrate redisplay of forms
 *  that have incomplete data.
 */

public class BidInfo {
  private String itemID = "";
  private String itemName = "";
  private String bidderName = "";
  private String emailAddress = "";
  private double bidPrice = 0;
  private boolean autoIncrement = false;
```

Listing 4.17 BidInfo.java *(continued)*

```
public String getItemName() {
  return(itemName);
}

public void setItemName(String itemName) {
  this.itemName = ServletUtilities.filter(itemName);
}

public String getItemID() {
  return(itemID);
}

public void setItemID(String itemID) {
  this.itemID = ServletUtilities.filter(itemID);
}

public String getBidderName() {
  return(bidderName);
}

public void setBidderName(String bidderName) {
  this.bidderName =  ServletUtilities.filter(bidderName);
}

public String getEmailAddress() {
  return(emailAddress);
}

public void setEmailAddress(String emailAddress) {
  this.emailAddress = ServletUtilities.filter(emailAddress);
}

public double getBidPrice() {
  return(bidPrice);
}

public void setBidPrice(double bidPrice) {
  this.bidPrice = bidPrice;
}

public boolean isAutoIncrement() {
  return(autoIncrement);
}
```

Listing 4.17 BidInfo.java *(continued)*

```
  public void setAutoIncrement(boolean autoIncrement) {
    this.autoIncrement = autoIncrement;
  }

  /** Has all the required data been entered? Everything except
      autoIncrement must be specified explicitly (autoIncrement
      defaults to false).
  */

  public boolean isComplete() {
    return(hasValue(getItemID()) &&
           hasValue(getItemName()) &&
           hasValue(getBidderName()) &&
           hasValue(getEmailAddress()) &&
           (getBidPrice() > 0));
  }

  /** Has any of the data been entered? */

  public boolean isPartlyComplete() {
    boolean flag =
           (hasValue(getItemID()) ||
           hasValue(getItemName()) ||
           hasValue(getBidderName()) ||
           hasValue(getEmailAddress()) ||
           (getBidPrice() > 0) ||
           isAutoIncrement());
    return(flag);
  }

  private boolean hasValue(String val) {
    return((val != null) && (!val.equals("")));
  }
}
```