Introducción al uso de JPA 2.0, Reflection, Annotations, Generics con Firebird Database en Java 6 Standard Edition

Lic. Guillermo Cherencio – UNLu – Programación III – Base de Datos

En el artículo anterior que escribí para Uds. "Introducción al uso de JPA 2.0 y Swing con Firebird Database en Java 6 Standard Edition" se mostró una interfaz gráfica hecha en swing utilizando el componente JTable, el cual trabaja bajo el patrón de diseño MVC (Model View Controller Pattern), a través de la implementación de la clase ClienteModel vimos como controlar y proveer de datos a la JTable, permitiendo hacer todo tipo de operaciones utilizando JPA 2.0 (consultas, actualizaciones y bajas). El único problema -a mi entender- es que esto sirve sólo para la entidad Cliente. Es demasiado específico, poco genérico. Un desarrollo bajo este diseño implicaría la implementación de una clase model para cada entidad.

¿De qué forma podemos hacer este código más genérico? Java 6 nos provee de tres tecnologías que pueden ayudarnos: Anotaciones (Annotations), Clases Genéricas (Generics) y Reflexión (Reflection).

Reflection

Permite -en tiempo de ejecución- que una clase Java pueda urgar en el contenido de otra clase Java. Por ejemplo, podemos averiguar qué atributos tiene, sus tipos de datos, si tienen anotaciones, sus métodos (obtener una referencia a los mismos para luego ejecutarlos!), etc. Permite acceder a toda información de metadata vinculada con nuestro código Java sin necesidad de contar con el código fuente!.

Casi todo el trabajo podemos hacerlo desde la clase `Class`, toda clase tiene un atributo estático llamado `class` que nos permite obtener una referencia a un objeto de tipo `Class`, ejemplo: `String.class`, `Integer.class`, `Empleado.class`, etc.

Otra forma de obtener una referencia de tipo `Class`, es a través del nombre completo de una clase y haciendo uso del método estático `forName()` de la clase `Class`:

```
...
String nombre = "java.lang.String";
Class strClass = null;
try {
   Class strClass = Class.forName(nombre);
} catch(ClassNotFoundException cnfe) {
   System.err.println("Error, la clase ["+nombre+"] no puede ser
cargada");
}
...
```

Como es de suponer, ¿Qué pasa si le pasamos un nombre equivocado al método `forName()`? `forName()` lanzará una excepción chequeable[1] que se llama `ClassNotFoundException`.

Una vez que tenemos una referencia de tipo Class, podemos usar métodos tales como:

Metodo de la clase Class	**Descripción**
`getAnnotations()`	Devuelve un arreglo con todas las anotaciones
`getConstructors()`	Devuelve un arreglo con todos los constructores de la clase
`getDeclaredFields()`	Devuelve un arreglo con todos los atributos de la clase
`getDeclaredMethods()`	Devuelve un arreglo con todos los métodos de la clase
`getInterfaces()`	Devuelve un arreglo con todas las interfases implementadas por la clase
`getName()`	Devuelve el nombre de la clase
`newInstance()`	Crea una nueva instancia (un objeto) de la clase

Los tipos de estos arreglos permiten -a su vez- invocar métodos propios que permiten indagar aun más sobre cada tipo de elemento de la clase que pretendemos analizar.

Annotations

Permite introducir metadata en nuestro código, a nivel de clase, de atributo, de método, etc. Son algo así como aclaraciones adicionales sobre lo que pretendemos hacer en la siguiente línea de codigo. Son como comentarios "activos", que tienen impacto en la compilación, no son ignorados por el compilador (como sucede con los comentarios). Esta información de metadata puede persistir dentro de nuestro código para luego estar disponible en runtime (para que otra clase -utilizando Reflection- pueda acceder a esta información) o sólo afectar a la compilación, sin dejar rastro dentro de nuestros archivos compilados (`.class`). Es muy utilizado en J2EE, pero también esta disponible en J2SE. Por ejemplo, una entidad -como ya vimos- es una simple clase POJO (Plain Old Java Object) que actúa como un Java Bean (clase que tiene atributos, metodos para acceder (getters) y para modificar dichos atributos (setters), constructor nulo y tambien puede agregar metodos propios (custom)). La mayoría de las clases tienen esta estructura, entonces, si esta clase debe ser administrada por un EntityManager para poder ser persistida en una base de

[1] Todas las excepciones chequeables son aquellas que derivan de la clase Exception. El compilador chequeará que el programador aplique la regla "handle & declare": o bien se escribe un bloque try - catch manipulando la excepción (handle) o bien declarás que dicho método puede lanzar dicha excepción usando la cláusula throws.

datos utilizando JPA 2.0, ¿cómo sabe JPA 2.0 que esta clase es una entidad? muy simple: anotándola como Entidad:

```
@Entity
public class MiEntidad {
   ...
}
```

`@Entity`, en este caso, es una anotación a nivel de clase, que persiste en runtime y le permite a un proveedor de persistencia JPA 2.0 poder identificar a una clase como persistente haciendo uso de reflection en runtime. Toda clase que no este "marcada" con `@Entity` no podrá ser persistida.

El lenguaje Java ha sido plagado de anotaciones en todos sus paquetes[2] que lo componen, viene con cientos de anotaciones predefinidas que permiten interpretar el código de distinta forma y en el caso de J2EE permite que los contenedores de clases analicen el codigo en runtime de las clases contenidas para actuar en función de lo anotado y permitiendo realizar cosas sorprendentes como "inyectar dependencias"[3].

Además de las anotaciones provistas por el lenguaje, podemos crear nuestras propias anotaciones, sin ningún tipo de restricción!, de forma similar a como se declaran las interfases. Por ejemplo, podriamos declarar la anotacion Columna (para luego anotar código como `@Columna(nombre="Nombre de Columna")` de la siguiente forma:

```
...
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.FIELD)
@interface Columna {
   String nombre() default "";
}
```

Para definir anotaciones tambien se usan anotaciones!, las cuales, obviamente se llaman meta-anotaciones, tales como: `@Documented`, `@Retention`, `@Target`, etc. `@Documented` permite indicar que dicha anotación sea tenida en cuenta cuando se genera la documentación de la clase usando javadoc[4], `@Retention` indica el tipo de persistencia de la información

---

[2] Toda clase java puede agruparse en paquetes (packages) acorde con la funcionalidad de las mismas. Por ejemplo, todo el núcleo del lenguaje java se encuentra dentro del package java.lang allí podemos encontrar clases tales como String, por lo tanto, esta clase, en realidad se llama java.lang.String y cuando fue implementada, su codigo fuente se ubicaba en la carpeta /java/lang

[3] Por ejemplo, podríamos declarar una referencia de tipo EntityManager sin instanciarla y anotarla de forma tal, que el contenedor pueda localizar una referencia del objeto EntityManager apropiado y copiar dicha referencia en nuestro código en runtime!. A partir de allí, si bien en mi clase yo sólo declaré la referencia, la puedo considerar "inyectada" automáticamente por el contenedor, despreocupándome dicha tarea.

[4] Herramienta provista con el J2SDK que permite documentar nuestras clases (al mismo estilo de la propia API de Java) utilizando unos tags javadoc que se ingresan dentro de comentarios del tipo /** */ . Estos tags javadoc son muy similares en cuanto a su sintaxis con respecto a las

de metadata, en este caso, quiero que persista dentro del `.class` para que pueda ser accedida en runtime por otra clase, `@Target` permite indicar el alcance de la anotación! (lo cual afecta a los controles del compilador!, por ejemplo, si yo pretendo usar esta anotación para anotar a una clase, el compilador generará un error de compilación, puesto que esta anotación es para atributos (`ElementType.FIELD`) y no para clase).

Ejemplo de uso de la anotación `Columna`:

```
...
@Entity
public class Cliente {
   ...
   @Columna(nombre="Razón Social")
   @Column(nullable=false,length=40)
   private String nombre;
   ...
}
```

Podría usarse para indicar el título que quiero ver en la columna de la grilla `JTable` para visualizar el contenido del atributo `nombre` de la entidad `Cliente`!.

Realmente esta tecnología me ha sorprendido y la considero un enorme avance en la programación.

Generics

Las clases genéricas son el equivalente Java de los templates de C++. Esta tecnología está relacionada con los conceptos de upcasting, downcasting y polimorfismo. Supongamos que tenemos una clase `Persona` y una clase `Empleado` que deriva de `Persona`. Podemos decir que una operación de upcasting "es segura", puesto que todo `Empleado` es una `Persona`:

```
...
Persona p;
Empleado e = new Empleado();
p = e;
p.metodoDePersona()
...
```

No sucede lo mismo con "downcasting" puesto que no toda `Persona` es un `Empleado`:

```
...
Persona p = new Empleado();
Empleado e = (Empleado) p;
e.metodoDeEmpleado();
...
```

---

anotaciones (ejemplo: @author, @since, @version, etc.), pero son sólo comentarios, no persistentes e ignorados por el compilador. No confundir con anotaciones.

En este caso funciona, puesto que p en realidad apunta a una instancia de tipo `Empleado`, pero ello no siempre es asi:

```
...
Persona p = new Persona();
Empleado e = (Empleado) p;  // error en runtime!
e.metodoDeEmpleado();
...
```

se genera un error en runtime, ya que el compilador no puede detectar ninguna violación. Lo ideal es tratar de detectar estos errores en tiempo de compilación[5].

Otro ejemplo de situaciones que pueden llevarnos a errores de downcasting es el uso de colecciones. El framework `Collections` fue programado para almacenar colecciones de objetos de cualquier tipo, por lo tanto, tuvieron que definirlo en términos de la clase `Object` (puesto que toda clase es de tipo `Object`) para permitir el almacenamiento de cualquier tipo de objeto. El problema se nos presenta al intentar recuperar los objetos que allí guardamos, no nos queda otra que hacer downcasting y el compilador no puede controlar que efectivamente estemos recuperando el tipo de objeto que pusimos originalmente dentro de la colección!. Ejemplo:

```
Lista lista = new ArrayList();
Integer i1 = new Integer(12);
Integer i2 = new Integer(1234);
lista.add(i1);
lista.add(i2);
...
// recupero el primer elemento de la lista asumiendo que se trata de un
// Integer
Integer i = (Integer) lista.get(0);
...
```

El método `get()` devuelve algo de tipo `Object`, pero no todo `Object` es un `Integer` !! y encima el compilador no puede ayudarnos!!. Solución: uso de Generics.

Una clase genérica implica dos cosas: implementarla genéricamente y luego usarla genéricamente. Volviendo al ejemplo anterior, un uso genérico y por lo tanto seguro de nuestra operación de downcasting sería el siguiente:

```
Lista<Integer> lista = new ArrayList<Integer>();
Integer i1 = new Integer(12);
Integer i2 = new Integer(1234);
// ahora el metodo add() recibe algo de tipo Integer en vez de Object
lista.add(i1);
lista.add(i2);
...
// ahora el metodo get() devuelve algo de tipo Integer en vez de Object
Integer i = lista.get(0);
```

[5] Sin perder las ventajas polimórficas del código.

```
...
```

Ahora sí el compilador puede controlar que hagamos un uso correcto de la colección:

```
Lista<Integer> lista = new ArrayList<Integer>();
Integer i1 = new Integer(12);
Integer i2 = new Integer(1234);
// ahora el metodo add() recibe algo de tipo Integer en vez de Object
lista.add(i1);
lista.add(i2);
...
// ahora el metodo get() devuelve algo de tipo Integer en vez de Object
Empleado e = lista.get(0); // error de compilación!!
...
```

esto puede lograrse gracias a que el framewrok `Collections` fue reescrito en forma genérica, entonces, ahora el metodo `add()` de la interfase `Collection` esta declarado genéricamente:

```
public interface Collection<E> {
   ...
   boolean add(E e);
   ...
}
```

¿Que es <E>? <E> puede ser sustituida -en tiempo de compilación- por cualquier clase, pero eso sí, el método `add()` deberá recibir una instancia de ese mismo tipo que acabo de sustituir. Observe la declaración del método `get()` de la interfase `List` (que deriva de `Collection`):

```
public interface List<E> extends Collection {
   ...
   E get(int index);
   ...
}
```

Ahora el método `get()` devolverá algo de un tipo genérico, el mismo que envié como argumento del método `add()`, por lo tanto, si invoco al método `add()` con un `String`, el método `get()` devoverá algo de tipo `String`, si no fuese así, el compilador dará un error, advirtiéndonos de nuestro error antes de ejecutarlo! facilitando el código y haciendo ahorrar mucho tiempo. Maravilloso.

Hay mucho más sobre generics, también se pueden usar expresiones del tipo <T `extends` W> en donde también hacemos referencias genericas de relaciones entre generics, hasta incluso usar <? `extends` W> o simplemente <?>.

Juntando todo (Putting all together)

Combinando estas tres tecnologías podemos resolver nuestro problema: podemos implementar una versión genérica de nuestra específica clase model `ClienteModel`, la llamaremos `Model<T>`, que tendrá este aspecto:

```
public class Model<T> extends AbstractTableModel implements KeyListener
{
   private List<T> lista;
   private List<T> listaAlta = new ArrayList<T>();
   private String columnas[];
   private Class clase_columnas[];
   private Method getters[];
   private Method setters[];
   private T miobj;
....
   public Model(final String nomGenerico,List<T> c, ... ) { ... }
...
}
```

Ahora <T> podría sustituirse por `<Cliente>` cuando la usemos desde `Main2`:

```
public class Main2 {
   ...
   public static void pantalla() {
      ...
      List<Cliente> clientes = qry.getResultList();
      Model<Cliente> cm = new
         Model<Cliente>("Cliente",clientes,Main2.em,tabla);
      tabla.setModel(cm);
      tabla.addKeyListener(cm);
      ...
   }
   ...
}
```

Al constructor de la clase `Model<T>` se le pasa el nombre de la clase para permitir crear una instancia de la misma (`miobj`) (puesto que debe contemplarse la posiblidad de que la lista `clientes` estuviera vacía), indagarla utilizando reflection, crear (con la cantidad de elementos apropiada) los arreglos (cuyas referencias estan previamente declaradas) `columnas`, `clase_columnas`, `setters`, `getters`, etc.

La indagación hecha en el constructor incluye la búsqueda de la anotación `Columna` para determinar el nombre (título) de cada columna[6] de la grilla:

```
public class Model<T> extends AbstractTableModel implements KeyListener
{
...
    public Model(final String nomGenerico,List<T> c,EntityManager
e,JTable t) {
      ...
        // para cada uno de los atributos de la clase hacer:
```

[6] Previamente, el título de cada columna se había asignado con el nombre de cada atributo.

```
        for(Field f : listaClass.getDeclaredFields()) {
         ...
            // determino si la columna es clave primaria y/o esta
            // anotada con la anotación Columna
            pk_columnas[i]=false;
            for(Annotation a:f.getAnnotations()) {
                if ( a.toString().startsWith("@javax.persistence.Id(") )
{ pk_columnas[i]=true; }
                if ( a.toString().startsWith("@Columna(") ) {
                   Columna aa = (Columna ) a;
                   columnas[i]=aa.nombre();
                }
            }
         ...
        }
    }
...
}
```

Obsérvese que se indaga acerca de las anotaciones que tiene cada atributo de la clase entidad <T>, la representación en forma de `String` de cada anotación incluye su nombre y sus atributos actuales, una vez que nos aseguramos que el atributo contiene la anotación `Columna`, obtenemos una referencia de tipo anotación `Columna` e invocamos su método `nombre()` el cual nos devuelve el título de la columna!.

El resto del código de la clase `Model<T>` sigue la misma lógica que `ClienteModel`, el código es algo más extenso en cuanto a la indagación a través de reflection (que se realiza en el constructor de `Model<T>`), pero el resto del código -hasta incluso- podría ser más simple que `ClienteModel`, por ejemplo, en el método `setValueAt()` y `getValueAt()` antes debíamos hacer un `switch(col) { }` para determinar cuál era el método getter o setter apropiado invocar, ahora, la versión genérica es más compacta:

```
public class Model<T> extends AbstractTableModel implements KeyListener
{
...
    public Object getValueAt(int row, int col) {
        if ( !modoAlta && row > lista.size() ) return null;
        if ( modoAlta && row > 0 ) return null;
        List<T> l = (!modoAlta) ? lista : listaAlta;
        try {
            if ( getters[col] != null ) return
getters[col].invoke(l.get(row));
        } catch(Exception e) {
            System.err.println("Model:Exception ejecutando getter de
columna"+col+":\n"+e.getMessage());
        }
        return null;
    }
    public void setValueAt(Object value, int row, int col) {
        if ( row > lista.size() ) return;
        if ( !modoAlta && col == 0 ) return;
        List<T> l = (!modoAlta) ? lista : listaAlta;
        if ( !modoAlta ) emt.begin();
        try {
            if ( setters[col] != null )
setters[col].invoke(l.get(row),value);
```

```
        } catch(Exception e) {
            System.err.println("Model:Exception ejecutando setter de
columna"+col+":\n"+e.getMessage());
        }
        if ( !modoAlta ) emt.commit();
        fireTableCellUpdated(row, col);
    }
...
}
```

Obsérvese que a través de una referencia al método setter o getter podemos realizar su invocación usando el método `invoke()`, cuyo primer argumento es el objeto sobre el cual se va a invocar al método[7] y del segundo argumento en adelante, son los argumentos del método setter o getter.

El proyecto Bluej `jpaqry` incluye el código explicado en el artículo anterior así como también el código explicado en este artículo. Para ejecutar la versión genérica de la grilla, debemos generar el `.jar` de este proyecto, elegir los `.jar` que forman parte de nuestra aplicación (jaybird jdbc driver, jpa 2.0, eclipseLink (proveedor de jpa 2.0), nuestro código), generarlo en un directorio `X`, posicionarse en el directorio `X` y desde la consola ejecutar la aplicación: `java -jar jpaqrytest.jar` (asumiendo que nuestro código se empaquetó bajo el nombre `jpaqrytest.jar` y que se indicó a `Main2` como clase ejecutable de nuestro proyecto). Esta operación ya fue explicada en el primer artículo. El resultado de la ejecución de `Main2` debería ser idéntico a la versión no genérica.

Este proyecto es una muestra elemental de la potencia de combinar estas tecnologías las cuales nos permiten desacoplar cada vez más nuestro modelo de objetos del modelo relacional, permitiendo que la misma aplicación pueda interactuar libremente con distintas implementaciones de la base de datos. Se podrían continuar agregando más anotaciones y/o propiedades a la actual anotación `Columna`[8] para "customizar" aún más nuestra grilla sin necesidad de escribir nuestras clases Model y Controller para realizar altas, bajas y modificaciones sobre nuestras entidades. Con esto finalizo esta serie de 3 artículos dedicados a JPA 2.0 usado desde J2SE 6, espero que le haya sido útil para su proyecto y futuros desarrollos.

Atte. Lic. Guillermo Cherencio

---

[7] Esto implica que teniendo una única referencia de tipo Method puedo realizar todas las invocaciones que desee, sobre distintos objetos.

[8] Por ejemplo Ud. podría agregar la propiedad "ancho" o "width" para indicar la cantidad de pixels de ancho que debería tener inicialmente esta columna en la grilla. También podría modificar Model<T> para inyectar dependencias o tomar conocimiento de los métodos que tratarán eventos de persistencia tales como: @PrePersist, @PreUpdate (ver método validar() de la clase Cliente), @PostLoad, @PostPersist, @PostUpdate (ver método calcular() de la clase Cliente).