Almacene Objetos con Facilidad

Java™ Data Objects

O'REILLY®

David Jordan & Craig Russell

JAVA DATA OBJECTS

Publicaciones relacionadas de O’Reilly

Ant: The Definitive Guide.

Building Java. Enterprise Applications

Database Programming with JDBC and Java.

Developing JavaBeans

Enterprise JavaBeans.

J2ME in a Nutshell

Java. 2D Graphics

Java. and SOAP

Java. & XML

Java. and XML Data Binding

Java. and XSLT

Java. Cookbook

Java. Cryptography

Java. Data Objects

Java. Distributed Computing

Java. Enterprise in a Nutshell

Java. Examples in a Nutshell

Java. Foundation Classes in a Nutshell

Java. I/O

Java. in a Nutshell

Java. Internationalization

Java. Message Service

Java. Network Programming

Java. NIO

Java. Performance Tuning

Java. Programming with Oracle SQLJ

Java. Security

JavaServer. Pages

Java. Servlet Programming

Java. Swing

Java. Threads

Java. Web Services

JXTA in a Nutshell

Learning Java

Mac OS X for Java. Geeks

NetBeans: The Definitive Guide

Programming Jakarta Struts

JAVA DATA OBJECTS

David Jordan and Craig Russell

O'Reilly

Beijing • Cambridge • Farnham • Köln • Paris • Sebastopol • Taipei • Tokyo

CAPÍTULO 1

UN PASEO INICIAL

Java es un lenguaje que define un entorno de ejecución en el cual se ejecutan clases definidas por el usuario. Las instancias de estas clases definidas por el usuario pueden estar representando datos del mundo real que son almacenados en una base de datos, sistema de ficheros o sistema mainframe para el proceso de transacciones. Los sistemas ligeros o de "huella pequeña" (small-footprint) requieren además frecuentemente un medio para manejar datos persistentes en sistemas de almacenamiento locales.

Al existir diferentes tecnologías de acceso a datos para cada tipo de fuente de datos, el acceso a los datos representa un desafío para los desarrolladores de aplicaciones que necesitan utilizar un interfaz de programación de aplicaciones (API – Application Programming Interface) diferente para cada una de las fuentes de datos. Eso significa que usted necesitará conocer, al menos, dos lenguajes diferentes para desarrollar lógica de negocio para estas fuentes de datos: el lenguaje de programación Java y el lenguaje de programación especializado requerido por la fuente de datos. El lenguaje para el acceso a datos será probablemente diferente para cada una de las fuentes de datos, incrementando con ello el coste de aprender a utilizar cada una.

Previamente a la publicación de Java Data Objects (JDO) existían tres estándares para almacenar datos en Java: serialización, Java DataBase Connectivity (JDBC) y Enterprise Java Beans (EJB) Container Manager Persistence (CMP). La serialización se utiliza para almacenar el estado de un objeto y el grafo de objetos que referencia a un flujo de salida. Mantiene las relaciones entre los objetos Java de manera que el grafo completo se puede reconstruir más adelante. Pero la serialización no soporta transacciones, consultas o compartir datos entre varios usuarios, solamente permite acceder con el nivel de granularidad de la serialización original, es decir, recuperar el grafo completo y se vuelve engorrosa cuando la aplicación necesita serializar múltiples grafos simultáneamente. La serialización se utiliza solamente para persistencia en las aplicaciones más sencillas o en entornos embebidos que no pueden soportar una base de datos de forma eficiente.

JDBC exige al desarrollador manejar explícitamente los campos y correlacionarlos con las tablas relacionales. El desarrollador es forzado a utilizar una doble vía a varios niveles: dos modelos de datos diferentes, dos lenguajes de programación y paradigmas de acceso a datos: Java y el modelo relacional de SQL. El esfuerzo de desarrollo para implementar su correlación (mapping) propia entre el modelo de datos relacional y el modelo es tan elevado, y su modelo de objetos Java tan grande que la mayoría de los desarrolladores nunca definen un modelo de objetos para sus datos; simplemente escriben código Java "procedural" para manipular las tablas de la base de datos relacional subyacente. El resultado es que no se benefician de las ventajas del desarrollo orientado a objetos.

La arquitectura de componentes EJB está diseñada para soportar aplicaciones informáticas distribuidas. Incluye además soporte para persistencia a través de Container Manager Persistence (CMP). En gran parte debido a sus capacidades distribuidas, las aplicaciones basadas en EJB son más complejas y sufren un mayor consumo de recursos (overhead) que JDO. De todos modos, JDO fue diseñado de tal manera que sus implementaciones pueden proveer servicios de persistencia en un entorno EJB integrándose con contenedores EJB. Si su aplicación necesita persistencia de objetos, pero no necesita funcionalidad distribuida, puede utilizar JDO en vez de componentes EJB. El uso más extendido de utilizar JDO en un entorno EJB consiste en utilizar componentes EJB del tipo Session Beans para manejar directamente objetos JDO, evitando el uso de Entity Beans. Componentes EJB tienen que ser ejecutados en un entorno administrado de un servidor de aplicaciones. Pero aplicaciones JDO pueden ser ejecutadas o bien en un entorno administrado o no administrado, aportándole la flexibilidad de elegir el entorno más apropiado para ejecutar su aplicación.

Usted puede desarrollar aplicaciones de forma más productiva si se centra en diseñar modelos de objetos Java y utiliza JDO para almacenar las instancias de sus clases directamente. Necesitará manejar solamente un único modelo de información. Sin embargo, JDBC le obliga a comprender el modelo relacional y el lenguaje SQL. Al utilizar EJB CMP, usted también será forzado a aprender y tratar con muchos otros aspectos de su arquitectura. Sufre además limitaciones de modelado que no están presentes en JDO.

JDO especifica los contratos entre sus clases persistentes y su entorno de ejecución. Ha sido diseñado para soportar un amplio abanico de fuentes de datos, incluyendo fuentes que no se suelen considerar bases de datos. Utilizamos por ese motivo el término de *almacén de datos* para referirnos a cualquier fuente de datos subyacente que es accedida mediante JDO.

Este capítulo explora algunas de las posibilidades básicas de JDO, examinando una pequeña aplicación desarrollada por una compañía ficticia llamada Media Mania, Inc. Esta compañía alquila y vende diferentes tipos de medios de entretenimiento en tiendas localizadas en todo EEUU. Sus tiendas disponen de quioscos que proveen información sobre películas y sus actores. Esta información se pone a disposición de los clientes y empleados de las tiendas para ayudar a seleccionar los artículos que son del interés de los clientes.

DEFINIENDO UN MODELO DE OBJETOS PERSISTENTE

En la Figura 1-1 se puede apreciar un diagrama UML (Unified Modelling Language) con las clases y relaciones en el modelo de Media Mania. Una instancia de una película representa una película concreta. Cada actor que ha tenido un papel en al menos una película es representado por una instancia de `Actor`. La clase `Role` (papel) representa los papeles específicos que un actor ha tenido en una película y representa por tanto una relación entre una película y un actor que incluye un atributo (el nombre de su papel). Cada película tiene uno o más papeles. Un actor puede haber tenido un papel en más de una película o haber tenido múltiples papeles en la misma película.

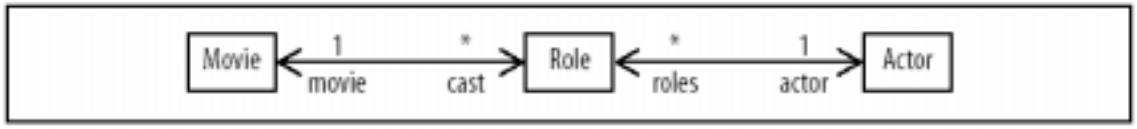

Figura 1-1. Diagrama UML del modelo de objetos de Media Mania.

Ubicaremos estas clases persistentes y los programas que manejan sus instancias en el paquete Java “com.mediamania.prototype”.

Las clases a persistir

Las clases `Movie` (película), `Actor` (actor) y `Role` (papel) las haremos persistentes, de manera que sus instancias puedan ser almacenadas en un almacén de datos. Primero examinaremos el código fuente completo para cada una de las clases. Se incluye una sentencia `import` para cada una de ellas, así queda claro qué paquete contiene a qué clase utilizada en el ejemplo.

El ejemplo 1-1 contiene el código fuente para la clase `Movie`. JDO se encuentra definido en el paquete `javax.jdo`. Observe que no se requiere importar ninguna clase específica de JDO. Para representar las relaciones entre nuestras clases, se utilizan referencias Java y colecciones definidas en el paquete `java.util` que es una práctica común en la mayoría de las aplicaciones Java.

Los campos de las clase `Movie` usan tipos Java estándar tales como `String`, `Date`, e `int`. Puede declarar campos como privados; no es necesario definir métodos `set` y `get` para cada campo. La clase `Movie` incluye algunos métodos `get` y `set` para acceder a los campos privados en de la clase, aunque estos métodos son utilizados por otras partes de la aplicación y no son requeridos por JDO. Puede utilizar encapsulación, aportando solamente los métodos que soportan la abstracción modelada. La clase contiene asimismo campos estáticos; estos no se almacenan en el almacén de datos.

El campo `genres` (género) es un `String` que contiene los géneros de la película (acción, romántica, misterio, etc.). Se utiliza un interface `Set` para referenciar a un conjunto de instancias de Role, representando el reparto de la película. El método `addRole()`[1] añade elementos a la colección contenedora del reparto, y `getCast()`[2] devuelve un Set no modificable que contiene los elementos de la colección contenedora del reparto. Estos métodos no son un requisito de JDO, pero se implementan como métodos convenientes para ella. Los métodos `parseReleaseDate()`[3] y `formatReleaseDate()`[4] se utilizan para estandarizar el formato de la fecha de estreno de la película. Para mantener el código sencillo, se devuelve un valur null si el parámetro del método `parseReleaseDate()` tiene un formato erróneo.

Ejemplo 1-1. Movie.java.

```
package com.mediamania.prototype;

import java.util.Set;
import java.util.HashSet;
import java.util.Collections;
import java.util.Date;
import java.util.Calendar;
import java.text.SimpleDateFormat;
import java.text.ParsePosition;

public class Movie {
   private static SimpleDateFormat yearFmt = new SimpleDateFormat("yyyy");
   public static final String[] MPAAratings = { "G", "PG", "PG-13", "R", "NC-17", "NR"};
   private String title;
   private Date releaseDate;
   private int runningTime;
   private String rating;
   private String webSite;
   private String genres;
   private Set cast; // tipo de elemento: Role
```

[1] Traducido: “añadir papel”

[2] Traducido: “obtener papel”

[3] Traducido: “analizar gramaticalmente fecha de estreno”

[4] Traducido: “formatear fecha de estreno”

Ejemplo 1-1. Movie.java (continuación)

```
   private Movie( )
   { }

   public Movie(String title, Date release, int duration, String rating,
                String genres) {
      this.title = title;
      releaseDate = release;
      runningTime = duration;
      this.rating = rating;
      this.genres = genres;
      cast = new HashSet( );
   }

   public String getTitle( ) {
      return title;
   }

   public Date getReleaseDate( ) {
      return releaseDate;
   }

   public String getRating( ) {
      return rating;
   }

   public int getRunningTime( ) {
      return runningTime;
   }

   public void setWebSite(String site) {
      webSite = site;
   }

   public String getWebSite( ) {
      return webSite;
   }

   public String getGenres( ) {
      return genres;
   }

   public void addRole(Role role) {
      cast.add(role);
   }

   public Set getCast( ) {
      return Collections.unmodifiableSet(cast);
   }

   public static Date parseReleaseDate(String val) {
      Date date = null;
      try {
         date = yearFmt.parse(val);
      } catch (java.text.ParseException exc) { }
         return date;
      }

   public String formatReleaseDate( ) {
      return yearFmt.format(releaseDate);
   }
}
```

JDO impone un requisito para hacer una clase persistente: que disponga de un constructor sin argumentos. Si no define ningún constructor en su clase, el compilador genera automáticamente un constructor sin argumentos. De todos los modos, este constructor no se genera si define cualquier constructor con argumentos; en este caso necesita incluir explícitamente un constructor sin argumentos. Puede declararlo como privado si no quiere que sea utilizado por el código de su aplicación. Algunas implementaciones JDO pueden generarlo por usted, pero ésta es una característica específica, no portable, de la implementación utilizada.

El Ejemplo 1-2 muestra el código fuente para la clase `Actor`. A nuestros efectos todos los actores tienen un nombre único que les identifica. Puede ser un nombre artístico que es diferente del nombre real. Por ese motivo, representamos el nombre del actor con una sola cadena. Cada actor ha actuado en uno o varios papeles, y el miembro roles modela la relación del lado del actor entre `Actor` y `Role`. El comentario en la línea **[1]** se utiliza solamente para documentación; no sirve a ningún propósito funcional en JDO. Los métodos `addRole`() y `removeRole`()[5] en las líneas **[2]** y **[3]** se incluyen para que la aplicación pueda mantener la relación entre una instancia de Actor y sus instancias Role asociadas.

Ejemplo 1-2. Actor.java

```
package com.mediamania.prototype;

import java.util.Set;
import java.util.HashSet;
import java.util.Collections;

public class Actor {
   private String name;
[1]private Set roles; // tipo de elemento: Role

   private Actor( )
   { }

   public Actor(String name) {
      this.name = name;
      roles = new HashSet( );
   }

   public String getName( ) {
      return name;
   }

[2]public void addRole(Role role) {
      roles.add(role);
   }

[3]public void removeRole(Role role) {
      roles.remove(role);
   }

   public Set getRoles( ) {
      return Collections.unmodifiableSet(roles);
   }
}
```

Finalmente, el Ejemplo 1-3 muestra el código fuente para la clase `Role`. Esta clase modela la relación entre `Movie` y `Actor` e incluye el nombre específico de papel representado por el actor en la película. El constructor de Role inicializa las referencias a `Movie` y `Actor`, y también actualiza los otros extremos de la relación llamando `addRole`(), que definimos en las clases `Movie` y `Actor`.

Ejemplo 1-3. Role.java.

```
public class Role {
   private String name;
   private Actor actor;
   private Movie movie;

   private Role( )
   { }

   public Role(String name, Actor actor, Movie movie) {
```

[5] Traducido: “eliminar papel”

Ejemplo 1-3. Role.java (continuación).

```
      this.name = name;
      this.actor = actor;
      this.movie = movie;
      actor.addRole(this);
      movie.addRole(this);
   }

   public String getName( ) {
      return name;
   }

   public Actor getActor( ) {
      return actor;
   }

   public Movie getMovie( ) {
      return movie;
   }
}
```

Con esto hemos examinado el código fuente completo para cada una de las clases que tendrán instancias en el almacén de datos. Estas clases no necesitaron importar ningún tipo específico de JDO. Es más, excepto el constructor sin argumentos, no hubo necesidad de definir métodos o datos para hacer estas clases persistentes. La forma de acceder y modificar campos y definir y gestionar relaciones entre instancias corresponde a la práctica estándar utilizada en la mayoría de las aplicaciones Java.

Declarando clases como Persistentes

Es necesario identificar qué clases deben ser persistentes y especificar cualquier información relacionada con la persistencia que no se pueda expresar en Java. JDO utiliza un fichero de meta-información en formato XML para especificar esta información.

Puede definir la meta-información según clases o paquetes, en uno o varios ficheros XML. El nombre del fichero de meta-información para una sola clase es el nombre de la clase, seguido de un sufijo *.jdo*. De este modo, un fichero de meta-información para la clase `Movie` se llamaría *Movie.jdo* y se ubicaría en el mismo directorio que el fichero *Movie.class*. Un fichero de meta-información para un paquete Java se ubica en un fichero llamado *paquete.jdo*. Un fichero de meta-información puede contener meta-información para múltiples clases y múltiples subpaquetes. El Ejemplo 1-4 aporta la meta-información para el modelo de objeto Media Mania. La meta-información se especifica para el paquete y se encuentra en un fichero llamado *com/mediamania/prototype/package.jdo*.

Ejemplo 1-4. Meta-información JDO en el fichero prototype/package.jdo

```
   <?xml version="1.0" encoding="UTF-8" ?>
[1]<!DOCTYPE jdo PUBLIC
      "-//Sun Microsystems, Inc.//DTD Java Data Objects Metadata 1.0//EN"
      "http://java.sun.com/dtd/jdo_1_0.dtd">
   <jdo>
      package name="com.mediamania.prototype" >
[3]      <class name="Movie" >
[4]         <field name="cast" >
[5]      <collection element-type="Role"/>
            </field>
         </class>
[6]      <class name="Role" />
         <class name="Actor" >
            <field name="roles" >
               <collection element-type="Role"/>
            </field>
         </class>
      </package>
   </jdo>
```

El fichero *jdo_1_0.dtd* especificado en la línea `[1]` contiene una descripción de los elementos que se pueden utilizar en un fichero de meta-información. Esta DTD está estandarizada en JDO y debería ser incluida con una implementación JDO. También se puede descargar de *http://java.sun.com/dtd*. Puede, además, alterar el DOCTYPE para hacer referencia a una copia local en su sistema de ficheros.

El fichero de meta-información puede contener información de persistencia para uno o más paquetes con clases persistentes. Cada paquete se define con una elemento `package` que incluye el nombre del paquete Java. Línea `[2]` contiene un elemento packge para nuestro paquete `com.mediamania.prototype`. Dentro del elemento `package` se encuentra elementos anidados que identifian clases persistentes del paquete (por ejemplo, la línea `[3]` contiene el elemento class para la clase `Movie`). El fichero puede contener múltiples elementos package enumerados secuencialmente; no se anidan.

Si hay que especificar información para un campo en particular, se anida un elemento field dentro del elemento class, tal como muestra la línea `[4]`. Por ejemplo: podría declarar el tipo de elemento para cada colección en el modelo. Esto no se requiere, pero puede resultar en un mapeado más eficiente. La clase `Movie` dispone de una colección con nombre cast, y la clase Actor contiene una colección con nombre roles; ambas contienen referencias a `Role`. Línea `[5]` especifica el tipo de elemento para cast. En muchos casos se asume un valor por defecto para un atributo que corresponda al valor que se necesita habitualmente.

Todos los campos que pueden ser persistentes se hacen persistentes por defecto. Campos estáticos y finales no se pueden hacer persistentes. Un campo declarado como transient (efímero, transitorio) en Java por defecto no es persistente, pero se puede declarar como persistente en el fichero de meta-información. El Capítulo 4 esta posibilidad.

Los Capítulos 4, 10, 12 y 13 abordan las características que puede especificar para clases y campos. Para una clase simple como `Role`, que no tiene colecciones, se puede limitar a incluir la clase en la meta-información tal como muestra la línea `[6]`, si no se necesitan otros atributos de meta-información.

ENTORNO DE CREACIÓN DE PROYECTOS

En esta sección examinaremos el entorno de desarrollo que se necesita para compilar y ejecutar nuestra aplicación JDO. Esto incluye la estructura de directorios del proyecto, los ficheros .jar necesarios para crear aplicaciones, y la sintaxis para enriquecer[6] (enhance) las clases persistentes. Describiremos el enhancement más adelante en esta sección. La instalación del entorno depende en parte de la implementación JDO que utiliza. La estructura del entorno de desarrollo y directorios de su proyecto puede variar.

Puede utilizar la implementación de referencia de SUN o bien otra implementación de su libre elección. Los ejemplos en este libro utilizan la implementación de referencia. Puede descargar la implementación de referencia en http://www.jcp.org y selccionando JSR-12. Una vez instalada una implementación JDO, necesitará establecer una estructura de directorio para el proyecto y definir un classpath que incluya todos los directorios y ficheros .jar necesarios para crear y ejecutar su aplicación.

[6] Nota del traductor: Enhance en Inglés. Éste es un término importante, el "enhancement" es un concepto clase dentro de JDO. Utilizaremos por tanto a partir de ahora el término original.

JDO introduce un nuevo paso en el proceso de creación (build) de su aplicación, llamado *enriquecimiento de clases* (class enhancement). Su clases persistentes son compiladas utilizando un compilador Java que produce un fichero class. Una aplicación de enriquecimiento lee esos ficheros de clases y la meta-información JDO creando nuevos ficheros de clases que han sido enriquecidos para trabajar en un entorno JDO. Su aplicación JDO debería cargar estas clases enriquecidas. La implementación JDO de referencia incluye un enhancer llamado *reference enhancer*.

Ficheros jar que necesita la Implementación JDO de referencia

Cuando utiliza la implementación JDO de referencia, debería incluir los siguientes ficheros jar en su classpath durante el desarrollo. Todos estos ficheros .jar deberían estar en su classpath durante la ejecución.

jdo.jar

Los interfaces y clases estándar definidas en la especificación JDO.

jdori.jar

La implementación de referencia de la especificación JDO de SUN.

btree.jar

Programa utilizado por la implementación JDO de referencia para gestionar el almacenamiento de datos en un fichero. La implementación de referencia utiliza un fichero para el almacenamiento de las instancias persistentes.

Jta.jar

Java Transaction API. El interfaz de sincronización es utilizado en el interfaz JDO y se encuentra en este fichero .jar. Otras facilidades definidas en este fichero serán probablemente útiles para una implementación JDO. Puede descargar este jar de http://java.sun.com/products/jta/index.html

Antlr.jar

Tecnología para el análisis gramatical (parking) utilizada en la implementación del lenguaje de consultas de JDO. La implementación de referencia utiliza la versión 2.7.0 de Antlr. Puede descargarlo de http://www.antlr.org

Xerces.jar

La implementación de referencia utiliza Xerces-J Versión 1.4.3 para parsear XML. Puede ser descargado de http://xml.apache.org/xerces-j/.

Los primeros tres ficheros .jar se incluyen con la implementación JDO de referencia; el último se puede descargar de los sitios web especificados.

La implementación de referencia incluye un fichero .jar adicional, *jdori-enhancer.jar*, que contiene la implementación de referencia para un enhancer. Las clases en *jdori-enhancer.jar* se encuentran también en *jdori.jar*. En la mayoría de los casos necesitará *jdori.jar* tanto en su entorno de desarrollo como de ejecución y no necesitará *jdori-enhancer.jar*. El fichero *jdori-enhancer.jar* se empaqueta de forma separada de manera que puede enriquecer sus clases utilizando el enhancer de referencia de forma independiente de una implementación JDO concreta. Algunas implementaciones, aparte de la implementación de referencia, pueden distribuir este fichero .jar para utilizar ellas mismas.

Si utiliza una implementación JDO diferente, su documentación debería proporcionarle una lista de todos los ficheros .jar necesarios. Una implementación ubica normalmente todos los ficheros .jar necesarios en un directorio determinado en su instalación. El fichero *jdo.jar* que contiene los interfaces definidos en JDO debería ser utilizado con todas las implementaciones. Este fichero .jar se incluye normalmente con la implementación del fabricante. JDOCentral.com (http://www.jdocentral.com) proporciona un amplio número de recursos JDO, incluyendo versiones de prueba gratuitas de muchas implementaciones comerciales.

Estructura de Directorios del Proyecto

Debería utilizar la siguiente estructura de directorios para el entorno de desarrollo de la aplicación Media Mania. El proyecto debe tener un directorio *root* (raíz) ubicado en laguna parte del sistema de ficheros. Los siguientes directorios residen dentro del directorio *root*:

src

Este directorio contiene todo el código fuente de la aplicación. Dentro de *src* se encuentra una jerarquía de subdirectorios de *com/mediamania/prototype* (correspondiente al paquete Java com.mediamania.prototype). Aquí es dónde se encuentran los ficheros *Movie.java*, *Actor.java* y *Role.java*.

classes

Cuando se compilan los ficheros fuente Java, sus correspondientes ficheros .class se sitúan en este directorio.

enhanced

Este directorio es el que contiene los ficheros enriquecidos (generados por el enhancer)

database

Este directorio contiene los ficheros utilizados por la implementación de referencia para almacenar los datos persistentes.

Aunque esta estructura de directorio en particular no es un requerimiento de JDO de la implementación de referencia, necesita entenderla para poder seguir la descripción de la aplicación Media Mania.

Cuando ejecuta su aplicación JDO, el runtime de Java tiene que cargar la versión enriquecida de los ficheros .class que se encuentran en nuestro directorio *enhanced*. Por tanto, el directorio enhanced debería incluirse antes del directorio *classes* en su classpath. Como enfoque alternativo, también puede enriquecer en el mismo sitio reemplazando los ficheros sin enriquecer con su versión enriquecida.

Enriqueciendo Clases para su Persistencia

Una clase tiene que ser enriquecida antes de que sus instancias pueden ser gestionadas en un entorno JDO. Un enhacer JDO añade atributos y métodos a sus clases que habilitan sus instancias para ser manejadas por una implementación JDO.

Un enhancer lee un fichero de clase generado por el compilador Java y, utilizando la meta-informacion JDO, produce un nuevo fichero .class que incluye la funcionalidad necesaria. JDO ha estandarizado las modificaciones hechas por los enhancers de manera que los ficheros enriquecidos son compatibles a nivel binario y pueden ser utilizados en cualquier implementación JDO. Estos ficheros enriquecidos son asimismo independientes de un almacén de datos específico.

Tal como mencionamos anteriormente, el enhancer incluido con la implementación JDO de referencia de SUN se llama *reference enhancer*. Un fabricante JDO puede incluir su propio enhancer; la sintaxis de la línea de comandos necesaria para ejecutar un enhancer puede variar de la sintaxis mostrada aquí. Cada implementación debe aportar documentación que explique cómo debe enriquecer sus clases para utilizarlas con su implementación.

El Ejemplo 1-5 muestra el enhancer de referencia para enriquecer las clases persistentes en nuestra aplicación de Media Mania. El argumento –d especifica el directorio *root* donde situar los ficheros .class enriquecidos; hemos especificado nuestro directorio *enhanced*. Al enhancer se le proporciona una lista de ficheros de meta-información y un conjunto de clases para enriquecer. El separador de directorio y los símbolos de continuación de línea pueden varias, dependiendo de su sistema operativo y entorno de compilación.

Aunque es conveniente situar los ficheros en el directorio con el código fuente, la especificación JDO recomienda que los ficheros de meta-información se encuentren disponibles por la vía de los recursos cargados por el mismo class loador que los ficheros .class. La meta-información de necesita tanto en tiempo de compilación como de ejecución. Por tanto, hemos situado el fichero de meta-información *package.jdo* dentro del la jerarquía de clases del directorio *classes* en el directorio del paquete prototype.

Los ficheros class para todas nuestras clases persistentes en nuestro modelo de objetos se enumeran en el Ejemplo 1-5, aunque también puede enriquecer cada clase individualmente. Cuando el comando se ejecuta, sitúa clases nuevas, enriquecidas, en el directorio *enhanced*.

Ejemplo 1-5. Enriqueciendo las clases persistentes

```
java com.sun.jdori.enhancer.Main -d enhanced \
classes/com/mediamania/prototype/package.jdo \
classes/com/mediamania/prototype/Movie.class \
classes/com/mediamania/prototype/Actor.class \
classes/com/mediamania/prototype/Role.class
```

CREANDO UNA CONEXIÓN AL ALMACÉN DE DATOS Y UNA TRANSACCIÓN

Ahora que nuestras clases han sido enriquecidas, sus instancias pueden ser almacenadas en un almacén de datos. Examinaremos ahora cómo una aplicación establece una conexión con un almacén de datos y ejecuta operaciones dentro de una transacción. Empezamos creado un programa que hace uso directo de los interfaces JDO. Todos los interfaces JDO utilizados por una aplicación son definidos en el paquete javax.jdo.

JDO dispone de un interfaz llamado `PersistenceManager` (administrador de persistencia) que tiene una conexión con un almacén de datos. Un `PersistenceManager` tiene asociada una instancia del interface JDO Transaction para controlar el comienzo y finalización de una transacción. La instancia de transacción se adquiere llamando a `currentTransaction()`[7] en la instancia de `PersistenceManager`.

[7] Traducido: “transacción actual”

Obteniendo un PersistenceManager

Se utiliza una `PersistenceManagerFactory` (factoría de PersistenceManager) para configurar y obtener un `PersistenceManager`. Los métodos en `PersistenceManagerFactory` se utilizan para definir las propiedades que controlan el comportamiento de las instancias de `PersistenceManager` adquiridas de la factoría. Por tanto, el primer paso ejecutado por una aplicación JDO es la adquisición de una instancia de `PersistenceManagerFactory`. Para obtener esta instancia, llame al siguiente método estático de la clase `JDOHelper`:

```
static PersistenceManagerFactory getPersistenceManagerFactory(Properties props);
```

La instancia de `Properties` se puede rellenar por programación o cargando valores de propiedades de un fichero de propiedades. El Ejemplo 1-6 enumera los contenidos del fichero propiedades que utilizaremos en nuestra aplicación de Media Mania. La propiedad `PersistenceManagerFactoryClass` en la línea **[1]** especifica la implementación JDO estamos utilizando mediante el nombre de la clase de la implementación que implementa el interfaz `PersistenceManagerFactory`. En este caso, especificamos la clase definida en la implementación JDO de referencia de SUN. Otras propiedades enumeradas en el Ejemplo 1-6 incluyen las URL de conexión utilizada para conectar a un almacén de datos concreto y un nombre de usuario y contraseña que pueden ser necesarios para establecer una conexión a un almacén de datos.

Ejemplo 1-6. Contenido de jdo.properties

```
[1]javax.jdo.PersistenceManagerFactoryClass=com.sun.jdori.fostore.FOStorePMF
   javax.jdo.option.ConnectionURL=fostore:database/fostoredb
   javax.jdo.option.ConnectionUserName=dave
   javax.jdo.option.ConnectionPassword=jdo4me
   javax.jdo.option.Optimistic=false
```

El formato de la URL de conexión depende de almacén de datos concreto al que se accede. La implementación JDO de referencia dispone de su propia facilidad de almacenamiento llamada `File Object Store (FOStore)`. La propiedad `ConnectionURL` en el Ejemplo 1-6 se encuentra en el directorio *database*, que está ubicado en el directorio *root* de nuestro proyecto. En este caso, hemos proporcionado un path relativo; también es posible utilizar un path absoluto al almacén de datos. La URL especifica que los ficheros del almacén FOStore tendrán el prefijo *fostoredb*.

Si está utilizando una implementación diferente, necesitará proveer valores diferentes para estas propiedades. También puede necesitar proveer valores para propiedades adicionales. Consulte la documentación de su implementación para determinar las propiedades necesarias.

Creando un almacén de datos FOStore

Para utilizar FOStore debemos crear primero un almacén de datos. El programa en el Ejemplo 1-7 crea un almacén de datos utilizando el fichero *jdo.properties*; todas las aplicaciones utilizan este fichero de propiedades. La línea **[1]** carga las propiedades de *jdo.properties* en una instancia de Properties. El programa añade la propiedad `com.sun.jdori.option.ConnectionCreate` en la línea **[2]** para indicar que el almacén de datos debe ser creado. Poner esta propiedad a `true` indica a la implementación que debe crear el almacén. A continuación llamamos a `getPersistenceManagerFactory( )` en la línea **[3]** para adquirir una instancia de `PersistenceManagerFactory`. La línea **[4]** crea un `PersistenceManager`.

Para completar la creación del almacén de datos, tenemos que iniciar y completar una transacción. El método `currentTransaction()` de `PersistenceManager` se llama en la línea **[5]** para acceder la instancia de la `Transaction` asociada al `PersistenceManager`. Los métodos begin() y commit() se invocan en las líneas **[6]** y **[7]** para iniciar y completar una transacción. Cuando ejecuta esta aplicación, un almacén FOStore se crea en el directorio *database*. Se crean dos ficheros: *fostore.btd* y *fostore.btx*.

Ejemplo 1-7. Creando un almacén de datos FOStore

```
package com.mediamania;

import java.io.FileInputStream;
import java.io.InputStream;
import java.util.Properties;
import javax.jdo.JDOHelper;
import javax.jdo.PersistenceManagerFactory;
import javax.jdo.PersistenceManager;
import javax.jdo.Transaction;

public class CreateDatabase {
   public static void main(String[] args) {
      create( );
   }
   public static void create( ) {
      try {
         InputStream propertyStream = new FileInputStream("jdo.properties");
         Properties jdoproperties = new Properties( );
[1]      jdoproperties.load(propertyStream);
[2]      jdoproperties.put("com.sun.jdori.option.ConnectionCreate", "true");
         PersistenceManagerFactory pmf =
[3]      JDOHelper.getPersistenceManagerFactory(jdoproperties);
[4]      PersistenceManager pm = pmf.getPersistenceManager( );
[5]      Transaction tx = pm.currentTransaction( );
[6]      tx.begin( );
[7]      tx.commit( );
      } catch (Exception e) {
         System.err.println("Exception creating the database");
         e.printStackTrace( );
         System.exit(-1);
      }
   }
}
```

La implementación JDO de referencia aporta los medios de programación para crear una base de datos. La mayoría de las bases de datos aportan una utilidad separada de JDO para crear la base de datos. JDO no define un interfaz estándar, independiente del fabricante, para la creación de una base de datos. La creación de un almacén de datos es siempre específica para el almacén de datos. Este programa ilustra cómo se hace utilizando el almacén de datos FOStore.

Además, cuando utiliza JDO con una base de datos relacional, existe frecuentemente un paso más de crear o mapear el modelo de objetos contra una estructura relacional existente. El procedimiento a seguir para establecer un esquema que corresponda con su modelo de objetos JDO depende de la implementación. Debería consultar la documentación de la implementación que está utilizando para determinar los pasos necesarios.

OPERACIONES CON INSTANCIAS

Ahora disponemos de un almacén de datos que puede almacenar instancias de nuestras clases. Cada aplicación necesita adquirir un `PersistenceManager` para acceder y actualizar el almacén de datos. El Ejemplo 1-8 aporta el código fuente para la clase `MediaManiaApp`, que sirve como la clase base para cada aplicación en este libro. Cada aplicación es una subclase concreta de `MediaManiaApp` que implementa su lógica de aplicación en el método ejecutar().

`MediaManiaApp` contiene un constructor que carga las propiedades del fichero *jdo.properties* (línea **[1]**). Después de cargar las propiedades del fichero, llama a `getPropertyOverrides`() e introduce las propiedades devueltas en `jdoproperties`. Una subclase de la aplicación puede redefinir `getPropertyOverrides`() para aportar propiedades adicionales o cambiar propiedades que han sido especificadas en el fichero *jdo.properties*. El constructor adquiere una `PersistenceManagerFactory` (línea **[2]**) y luego un `PersistenceManager` (línea **[3]**). También aportamos el método `getPersistenceManager`() para acceder al `PersistenceManager` desde fuera de la clase `MediaManiaApp`. La transacción asociada al `PersistenceManager` se adquiere en la línea **[4]**.

Las subclases de la aplicación hacen una llamada a `executeTransaction`(), definida en la clase `MediaManiaApp`. Este método inicia una transacción en la línea **[5]**. Luego llama a `execute()` en la línea **[6]** que ejecutará la funcionalidades específica de la subclase.

Escogimos este diseño particular para las clases de la aplicación con el objetivo de simplificar y reducir la cantidad de código redundante necesaria en los ejemplos para establecer un entorno de ejecución. Esto no es un requerimiento en JDO; puede escoger simplemente el enfoque que mejor se adapte a su entorno de aplicación.

Después de finalizar la llamada al método execute() se hace un intento de completar la transacción (línea **[7]**). Si se lanza alguna excepción, se realiza un rollback de la transacción y se imprime la excepción por el flujo de errores.

Ejemplo 1-8. La clase base MediaManiaApp.

```
package com.mediamania;

import java.io.FileInputStream;
import java.io.InputStream;
import java.util.Properties;
import java.util.Map;
import java.util.HashMap;
import javax.jdo.JDOHelper;
import javax.jdo.PersistenceManagerFactory;
import javax.jdo.PersistenceManager;
import javax.jdo.Transaction;

public abstract class MediaManiaApp {
   protected PersistenceManagerFactory pmf;
   protected PersistenceManager pm;
   protected Transaction tx;

   public abstract void execute( ); // definido en subclases concretas de la aplicación

   protected static Map getPropertyOverrides( ) {
      return new HashMap( );
   }

   public MediaManiaApp( ) {
      try {
         InputStream propertyStream = new FileInputStream("jdo.properties");
         Properties jdoproperties = new Properties( );
[1]      jdoproperties.load(propertyStream);
         jdoproperties.putAll(getPropertyOverrides( ));
[2]      pmf = JDOHelper.getPersistenceManagerFactory(jdoproperties);
[3]      pm = pmf.getPersistenceManager( );
```

Ejemplo 1-8. La clase base MediaManiaApp (continuación).

```
[4]       tx = pm.currentTransaction( );
       } catch (Exception e) {
          e.printStackTrace(System.err);
          System.exit(-1);
       }
   }

   public PersistenceManager getPersistenceManager( ) {
      return pm;
   }

   public void executeTransaction( ) {
      try {
          tx.begin( );
[6]       execute( );
[7]       tx.commit( );
       } catch (Throwable exception) {
          exception.printStackTrace(System.err);
          if (tx.isActive()) tx.rollback( );
       }
   }
}
```

Haciendo las Instancias Persistentes

Examinemos una simple aplicación, llamada `CreateMovie`, que hace persistente una instancia concreta, tal como se muestra en el Ejemplo 1-9. La funcionalidad de la aplicación se encuentra en `execute`(). Después de construir una instancia de `CreateMovie`, llamamos a `executeTransaction`(), que se define en la clase base `MediaManiaApp`. Ésta hace una llamada a `execute`(), que es el método definido en esta clase. El método `execute`() crea una sola instancia de `Movie` en la línea **[5]**. La llamada al método `makePersistent`() de `PersistenceManager` en la línea **[6]** hace la instancia de `Movie` persistente. Si la transacción se completa de forma exitosa en `executeTransaction`(), la instancia de `Movie` será almacenada en el almacén de datos.

Ejemplo 1-9. Creando una instancia de Movie y haciéndola persistente

```
package com.mediamania.prototype;

import java.util.Calendar;
import java.util.Date;
import com.mediamania.MediaManiaApp;

public class CreateMovie extends MediaManiaApp {

   public static void main(String[] args) {
      CreateMovie createMovie = new CreateMovie( );
      createMovie.executeTransaction( );
   }

   public void execute( ) {
      Calendar cal = Calendar.getInstance( );
      cal.clear( );
      cal.set(Calendar.YEAR, 1997);
      Date date = cal.getTime( );
[5]   Movie movie = new Movie("Titanic", date, 194, "PG-13", "historical, drama");
[6]   pm.makePersistent(movie);
   }
}
```

Examinemos ahora una aplicación de mayor tamaño. `LoadMovies`, que se muestra en el Ejemplo 1-10, lee un fichero que contiene información de películas y crea múltiples instancias de `Movie`. El nombre del fichero se pasa a la aplicación como argumento, y el constructor de `LoadMovies` inicializa un `BufferedReader` para leer los datos.

El método `execute`() lee una línea del fichero cada vez y llama a `parseMovieData`(), que parsea la línea de información de entrada, crea una instancia de `Movie` en la línea **[1]**, y la hace persistente en la línea **[2]**. Cuando la transacción se completa en `executeTransaction`(), todas las instancias de `Movie` creadas se almacenarán en el almacén de datos.

Ejemplo 1-10. LoadMovies.

```
package com.mediamania.prototype;

import java.io.FileReader;
import java.io.BufferedReader;
import java.util.Calendar;
import java.util.Date;
import java.util.StringTokenizer;
import javax.jdo.PersistenceManager;
import com.mediamania.MediaManiaApp;

public class LoadMovies extends MediaManiaApp {
   private BufferedReader reader;

   public static void main(String[] args) {
      LoadMovies loadMovies = new LoadMovies(args[0]);
      loadMovies.executeTransaction( );
   }

   public LoadMovies(String filename) {
      try {
         FileReader fr = new FileReader(filename);
         reader = new BufferedReader(fr);
      } catch (Exception e) {
         System.err.print("Unable to open input file ");
         System.err.println(filename);
         e.printStackTrace( );
         System.exit(-1);
      }
   }

   public void execute( ) {
      try {
         while ( reader.ready( ) ) {
            String line = reader.readLine( );
            parseMovieData(line);
         }
      } catch (java.io.IOException e) {
         System.err.println("Exception reading input file");
         e.printStackTrace(System.err);
      }
   }

   public void parseMovieData(String line) {
      StringTokenizer tokenizer = new StringTokenizer(line, ";");
      String title = tokenizer.nextToken( );
      String dateStr = tokenizer.nextToken( );
      Date releaseDate = Movie.parseReleaseDate(dateStr);
      int runningTime = 0;
      try {
         runningTime = Integer.parseInt(tokenizer.nextToken( ));
      } catch (java.lang.NumberFormatException e) {
         System.err.print("Exception parsing running time for ");
         System.err.println(title);
      }
      String rating = tokenizer.nextToken( );
      String genres = tokenizer.nextToken( );
[1]   Movie movie = new Movie(title, releaseDate, runningTime, rating, genres);
[2]   pm.makePersistent(movie);
   }
}
```

La información de las películas se encuentra en un fichero con el siguiente formato:

```
movie title;release date;running time;movie rating;genre1,genre2,genre3
```

El formato a utilizar para fechas de estreno se mantiene en la clase `Movie`, de manera que `parseReleaseDate`() se llama para crear una instancia de Date a partir de la información de entrada. Una película es descrita por uno o más géneros que se listan al final de la línea de información.

Accediendo a Instancias

Accedamos ahora a las instancias de `Movie` en el almacén de datos para comprobar que fueron almacenadas con éxito. Existen diferentes maneras de acceder instancias en JDO:

- Iterar un extent
- Navegar por el modelo de objetos
- Ejecutar una consulta

Un *extent* es una facilidad utilizada para acceder a todas las instancias de una clase particular o la clase y todas sus subclases. Si la aplicación quiere acceder solamente a un subconjunto de las instancias, se puede ejecutar una consulta con un filtro que limite las instancias devueltas a aquellas que satisfacen un predicado booleano. Una vez que la aplicación ha accedido a una instancia desde del almacén de datos, puede navegar a las instancias relacionadas en el almacén a través de referencias e iterando colecciones en el modelo de objetos. Las instancias aún no se encuentran en memoria y se leen bajo demanda. Estas facilidades para acceder a instancias se utilizan frecuentemente de forma combinada, y JDO asegura que cada una de las instancias persistentes existe solamente una única vez en memoria por `PersistenceManager`. Cada uno de los `PersistenceManager` maneja un único contexto de transacción.

Iterando un extent

JDO aporta el interfaz `Extent` para acceder el extent de una clase. El extent permite acceder a todas las instancias de una clase, pero utilizar un extent no implica que todas las instancias se encuentran en memoria. La aplicación `PrintMovies` del Ejemplo 1-11 utiliza el extent de `Movie`.

Ejemplo 1-11. Iterando el extent de Movie.

```
package com.mediamania.prototype;

import java.util.Iterator;
import java.util.Set;
import javax.jdo.PersistenceManager;
import javax.jdo.Extent;
import com.mediamania.MediaManiaApp;

public class PrintMovies extends MediaManiaApp {

   public static void main(String[] args) {
      PrintMovies movies = new PrintMovies( );
      movies.executeTransaction( );
   }

   public void execute( ) {
      Extent extent = pm.getExtent(Movie.class, true);
[2]   Iterator iter = extent.iterator( );
      while (iter.hasNext( )) {
[3]      Movie movie = (Movie) iter.next( );
         System.out.print(movie.getTitle( )); System.out.print(";");
         System.out.print(movie.getRating( )); System.out.print(";");
         System.out.print(movie.formatReleaseDate( ) ); System.out.print(";");
```

Ejemplo 1-11. Iterando el extent de Movie (continuación).

```
        System.out.print(movie.getRunningTime( )); System.out.print(";");
[4]     System.out.println(movie.getGenres( ));

[5]     Set cast = movie.getCast( );
        Iterator castIterator = cast.iterator( );
        while (castIterator.hasNext( )) {
[6]        Role role = (Role) castIterator.next( );
```

Ejemplo 1-11. Iterando el extent de Movie (continuación).

```
           System.out.print("\t");
           System.out.print(role.getName( ));
           System.out.print(", ");
[7]        System.out.println(role.getActor().getName( ));
        }
     }
[8]  extent.close(iter);
  }
}
```

En la línea **[1]** adquirimos un extent para la clase `Movie` del `PersistenceManager`. El segundo parámetro indica si se deben incluir instancias de subclases de `Movie`. Un valor `false` hace que solamente se devuelvan instancias de `Movie`, incluso si existen instancias de subclases. Aunque no disponemos actualmente de clases que extiendan la clase `Movie`, especificar un valor `true` hará que se devuelvan instancias de este tipo de clases que podamos definir en el futuro. El interfaz `Extent` dispone del método `iterator`(), que llamamos en la línea **[2]** para obtener un `Iterator` que accederá a cada elemento del extent. La línea **[3]** utiliza el `Iterator` para acceder a las instancias de `Movie`. La aplicación puede ejecutar entonces operaciones con la instancia de `Movie` para obtener información sobre la película a imprimir. Por ejemplo: en la línea **[4]** llamamos `getGenres`() para obtener los géneros asociados a la película. En la línea **[5]** obtenemos un conjunto de papeles. Obtenemos una referencia a una instancia de `Role` en la línea **[6]** y luego imprimimos el nombre del papel. En la línea **[7]** navegamos al actor de este papel llamando a `getActor`() que hemos definido en la clase `Role`. A continuación imprimimos el nombre del actor.

Una vez que la aplicación ha completado la iteración a lo largo del extent, la línea **[8]** cierra el iterador para liberar posibles recursos que se hayan utilizado para iterar el extent. Se pueden utilizar múltiples instancias de `Iterator` simultáneamente sobre un `Extent`. Este método cierra un `Iterator` específico; `closeAll`() cierra todas las instancias de `Iterator` asociadas a un Extent.

Navegando por el modelo de objetos

El Ejemplo 1-11 demuestra la iteración de un extent de Movie. Pero en la línea **[6]** también navegamos por un conjunto de instancias de Role relacionadas iterando una colección en nuestro modelo de objetos. En la línea **[7]** utilizamos la instancia de Role para navegar por medio de una referencia a la instancia de Actor relacionada. Las líneas **[5]** y **[7]** respectivamente demuestran la navegación de relaciones del tipo *a muchos* y *a uno*. Una relación desde una clase a otra tiene una cardinalidad que indica si existen una o múltiples instancias asociadas. Se utiliza una referencia para una cardinalidad de uno, y una colección cuando puede haber más de una instancia.

La sintaxis necesaria para acceder a estas instancias relacionadas corresponde a la práctica estándar de navegar por instancias en memoria. La aplicación no necesita hacer llamadas directas a interfaces JDO entre las líneas **[3]** y **[7]**. Simplemente atraviesa objetos en memoria. Las instancias relacionadas no se leen directamente desde el almacén de datos y se instancian en memoria hasta que son accedidas por la aplicación. El acceso al almacén de datos es trasparente; las instancias se cargan en memoria bajo demanda. Algunas implementación aportan facilidades separadas del interfaz Java que permiten alterar los algoritmos de acceso y caché de la implementación. Su aplicación Java se encuentra aislada de estas optimizaciones, pero se puede beneficiar de ellas en cuanto a su rendimiento global.

El acceso a instancias persistentes relacionadas en un entorno JDO es idéntico al acceso de instancias efímeras (transient) en un entorno no-JDO, de esta forma puede escribir su software de manera independiente de su uso en un entorno JDO. El software existente, escrito sin conocimiento de JDO u otras problemáticas relacionadas con la persistencia es capaz de navegar por los objetos de la base de datos a través de JDO. Esta capacidad conlleva un incremento dramático en la productividad del desarrollo y permite a software existente ser incorporado en un entorno JDO de forma rápida y sencilla.

Ejecutando una consulta

También es posible ejecutar una consulta sobre un `Extent`. El interfaz `Query` de JDO se utiliza para seleccionar un subconjunto de las instancias que cumplen ciertos criterios. Los ejemplos restantes en este capítulo necesitan acceder a un objeto `Actor` o `Movie` específico, basado en un nombre único. Estos métodos, mostrados en el Ejemplo 1-12, son prácticamente idénticos; `getActor`() ejecuta una consulta para obtener un objeto `Actor` basado en un nombre, y `getMovie`() ejecuta una consulta para obtener una película (clase `Movie`) basada en un nombre.

Ejemplo 1-12. Métodos de consulta en la clase PrototypeQueries.

```
package com.mediamania.prototype;

import java.util.Collection;
import java.util.Iterator;
import javax.jdo.PersistenceManager;
import javax.jdo.Extent;
import javax.jdo.Query;

public class PrototypeQueries {
   public static Actor getActor(PersistenceManager pm, String actorName)
   {
[1]   Extent actorExtent = pm.getExtent(Actor.class, true);
[2]   Query query = pm.newQuery(actorExtent, "name == actorName");
[3]   query.declareParameters("String actorName");
[4]   Collection result = (Collection) query.execute(actorName);
      Iterator iter = result.iterator( );
      Actor actor = null;
[5]   if (iter.hasNext()) actor = (Actor)iter.next( );
[6]   query.close(result);
      return actor;
   }

   public static Movie getMovie(PersistenceManager pm, String movieTitle)
   {
      Extent movieExtent = pm.getExtent(Movie.class, true);
      Query query = pm.newQuery(movieExtent, "title == movieTitle");
      query.declareParameters("String movieTitle");
      Collection result = (Collection) query.execute(movieTitle);
      Iterator iter = result.iterator( );
      Movie movie = null;
      if (iter.hasNext()) movie = (Movie)iter.next( );
      query.close(result);
      return movie;
   }
}
```

Examinemos `getActor`(). En la línea `[1]` obtenemos una referencia al extent de `Actor`. La línea `[2]` crea una instancia de `Query` utilizando el método `newQuery`() definido en el interfaz `PersistenceManager`. La consulta se inicializa con el extent y un filtro de consulta para aplicar el extent.

El identificador `name` en el filtro es el nombre de la clase Actor. El espacio de nombres utilizado para determinar cómo se interpreta el identificador está basado en la clase del Extent utilizada para inicializar la instancia de `Query`. La expresión de filtro requiere que un campo de nombre de un Actor sea igual a `actorName`. En el filtro podemos utilizar el parámetro == directamente para comparar dos Strings, en vez de utilizar la sintaxis Java (`name.equals(actorName)`).

El identificador `actorName` representa un parámetro de consulta, que se declara en la línea `[3]`. Un parámetro de consulta permite utilizar un valor para la ejecución de la consulta. Hemos decidido utilizar el mismo nombre, `actorName`, como parámetro de método y parámetro de consulta. Esta práctica no se requiere, y no hay una asociación directa entre los nombres de nuestros parámetros de los métodos Java y nuestros parámetros de consulta. La consulta se ejecuta en la línea `[4]` , pasando el parámetros de `getActor`() como el valor a utilizar para el parámetro `actorName` de la consulta.

El tipo del resultado de `Query.execute`() se declara como `Object`. En JDO 1.0.1, la instancia devuelta es siempre una colección, de manera que podemos convertir el resultado a una `Collection`. Se declara en JDO 1.0.1 con un valor de retorno tipo `Object` para permitir la futura extensión de devolver otros valores distintos a `Collection`. Nuestro método luego adquiere un `Iterator` y, en la línea `[5]` , intenta acceder a un elemento. Asumimos aquí que sólo puede haber una única instancia de Actor para un nombre determinado. Antes de devolver el resultado, la línea `[6]` , cierra el resultado de la consulta para liberar posibles recursos. Si el método encuentra una instancia de Actor con el nombre indicado, la instancia se devuelve. De lo contrario, si el resultado de la consulta no contiene elementos, se devuelve null.

Modificando una instancia

Examinemos ahora dos aplicaciones que modifican una instancia en un almacén de datos. Una vez que una aplicación haya accedido a una instancia del almacén de datos en una transacción, puede modificar uno o más campos de esa instancia. Cuando se realiza el commit de la transacción, todas las modificaciones que se han hecho a las instancias se propagan al almacén de datos de forma automática.

La aplicación `UpdateWebsite` en el Ejemplo 1-13 se utiliza para asignar el sitio web correspondiente a una película. Toma dos argumentos: el primero es el título de la película, y el segundo es la URL del sitio web de la película. Después de inicializar la instancia de la aplicación, se llama a `executeTransaction`(), que llama al método `execute`() definido en esta clase.

La línea `[1]` llama a `getMovie`() [definida en el Ejemplo 1-12] para recuperar una película con un nombre determinado. Si `getMovie`() devuelve null, la aplicación indica que no pudo encontrar una película con el título indicado y retorna. En el caso contrario, en la línea `[2]` llamamos a `setWebSite`() [definida para la clase `Movie` en el Ejemplo 1-1], que asigna al campo `webSite` de `Movie` el valor del parámetro. Cuando `executeTransaction`() realiza en commit de la transacción, la modificación a la instancia de `Movie` se propaga al almacén de datos automáticamente.

Ejemplo 1-13. Modificar un atributo.

```
package com.mediamania.prototype;

import com.mediamania.MediaManiaApp;

public class UpdateWebSite extends MediaManiaApp {
   private String movieTitle;
   private String newWebSite;

   public static void main (String[] args) {
      String title = args[0];
      String website = args[1];
      UpdateWebSite update = new UpdateWebSite(title, website);
      update.executeTransaction( );
   }

   public UpdateWebSite(String title, String site) {
      movieTitle = title;
      newWebSite = site;
   }

   public void execute( ) {
[1]   Movie movie = PrototypeQueries.getMovie(pm, movieTitle);
      if (movie == null) {
      System.err.print("Could not access movie with title of ");
      System.err.println(movieTitle);
      return;
      }
[2]   movie.setWebSite(newWebSite);
   }
}
```

Como se puede apreciar en el Ejemplo 1-13, la aplicación no necesita hacer llamadas directas a interfaces JDO para modificar el campo `Movie`. Esta aplicación accede a una instancia y llama un método para modificar el campo del sistio web. El método modifica el campo utilizan una sintaxis Java estándar. No se requiere ninguna programación adicional antes del commit para propagar los datos al almacén de datos. El entorno JDO propaga las modificaciones automáticamente. Esta aplicación ejecuta una operación en instancias persistentes, pero no importa ni usa directamente ningún interfaz JDO.

Veamos ahora una aplicación más grande, llamada `LoadRoles`, que expone varias de las capacidades de JDO. `LoadRoles`, mostrada en el Ejemplo 1-14, es responsable de cargar información sobre los papeles de las películas y los actores que actúan en ellas. `LoadRoles` recibe un único argumento que especifica el nombre del fichero a leer, y un constructor que inicializa un `BufferedReader` para leer el fichero. Lee el fichero de texto, que contiene un papel por línea, con el siguiente formato: `título película;nombre actor;nombre papel`

Normalmente, todos los papeles asociados con un película en particular se agrupan en este fichero; `LoadRoles` realiza una pequeña optimización para determinar si la información sobre el papel es para la misma película que la del papel anterior.

Ejemplo 1-14. Modificación de una instancia y persistencia por alcance.

```
package com.mediamania.prototype;

import java.io.FileReader;
import java.io.BufferedReader;
import java.util.StringTokenizer;
import com.mediamania.MediaManiaApp;

public class LoadRoles extends MediaManiaApp {
   private BufferedReader reader;

   public static void main(String[] args) {
      LoadRoles loadRoles = new LoadRoles(args[0]);
      loadRoles.executeTransaction( );
```

Ejemplo 1-14. Modificación de una instancia y persistencia por alcance (continuación).

```
   }

   public LoadRoles(String filename) {
      try {
         FileReader fr = new FileReader(filename);
         reader = new BufferedReader(fr);
      } catch(java.io.IOException e){
         System.err.print("Unable to open input file ");
         System.err.println(filename);
         System.exit(-1);
      }
   }

   public void execute( ) {
      String lastTitle = "";
      Movie movie = null;
      try {
         while (reader.ready( )) {
            String line = reader.readLine( );
            StringTokenizer tokenizer = new StringTokenizer(line, ";");
            String title = tokenizer.nextToken( );
            String actorName = tokenizer.nextToken( );
            String roleName = tokenizer.nextToken( );
            if (!title.equals(lastTitle)) {
               movie = PrototypeQueries.getMovie(pm, title);
               if (movie == null) {
                  System.err.print("Movie title not found: ");
                  System.err.println(title);
                  continue;
               }
               lastTitle = title;
            }
[2]         Actor actor = PrototypeQueries.getActor(pm, actorName);
            if (actor == null) {
[3]            actor = new Actor(actorName);
[4]            pm.makePersistent(actor);
            }
[5]      Role role = new Role(roleName, actor, movie);
         }
      } catch (java.io.IOException e) {
         System.err.println("Exception reading input file");
         System.err.println(e);
         return;
      }
   }
}
```

El método `execute()` lee cada entrada en el fichero. Primero, chequea para ver si el título de la película de la nueva entrada es el mismo que en la entrada anterior. Si no es así, la línea **[1]** llama a `getMovie()` para acceder a la película con el nuevo título. Si no existe una película con ese título en el almacén de datos, la aplicación imprime un mensaje de error y se salta la entrada actual. En la línea **[2]** intentamos acceder a una instancia de `Actor` con el nombre especificado. Si no hay ningún `Actor` en el almacén de datos con este nombre, se crea uno nuevo y se le asigna este nombre en la línea **[3]**, y se hace persistente en la línea **[4]**.

En este punto de la aplicación, hemos estando leyendo el fichero de entrada y buscando instancias en el almacén de datos que han sido referenciadas por un nombre en el fichero. Ejecutamos la tarea real de la aplicación en la línea **[5]**, dónde creamos una nueva instancias de `Role`. El constructor de `Role` fue definido en el Ejemplo 1-3; se repite aquí de manera que lo podemos examinar con mayor detalle:

```
public Role(String name, Actor actor, Movie movie) {
[1]   this.name = name;
[2]   this.actor = actor;
[3]   this.movie = movie;
[4]   actor.addRole(this);
[5]   movie.addRole(this);
}
```

La línea **[1]** inicializa el nombre del papel. La línea **[2]** crea una referencia a la instancia de `Actor` asociada. Las relaciones entre `Actor` y Role y entre `Movie` y `Role` son bidireccionales, por tanto tambien es necesario actualizar cada uno de los extremos de la relación. En la línea **[3]** llamamos a `addRole`() de actor que añade este papel a la colección de roles en la clase `Actor`. De forma similar, la línea **[4]** llama a `addRole`() en la colección película de papeles en la clase `Actor`. De forma similar, la línea **[5]** llama a `addRole`() en `Movie` para añadir este papel al campo de colección `cast` en la clase `Movie`. Añadir este papel como un elemento en `Actor.roles` y `Movie.cast` causa una modificación a las instancias referenciadas por actor y movie.

El constructor de Role demuestra que puede establecer una relación a una instancia simplemente inicializando una referencia a ella, y que puede establecer una referencia con más de una instancia añadiendo referencias a una colección. Esta es la manera de la que se representan relaciones en Java y se soportan directamente por JDO. Cuando se realiza el commit de la transacción, las relaciones establecidas en memoria se preservan en el almacén de datos.

Después de retornar del constructor de `Role`, `load`() procesa la siguiente entrada en el fichero. El bucle while termina una vez leído todo el contenido del fichero.

Posiblemente haya notado que nunca llamamos `makePersistent`() en las instancias de Role que creamos. Aún así, las instancias de Role se almacenan en el almacén de datos cuando se realiza el commit porque JDO soporta *persistencia por alcance*. La persistencia por alcance hace que cualquier instancia efímera (transient), es decir, no persistente de una clase persistente, se haga persistente en el commit si puede ser alcanzada (directa o indirectamente) por una instancia persistente. Instancia son alcanzables o bien por una referencia o una colección de referencias. El conjunto de todas las instancias alcanzables desde una instancia determinada es un grafo de objetos llamado el *cierre completo* (complete closure) de instancias relacionadas. El algoritmo de cálculo de alcance se aplica transitivamente a todas las instancias por todas sus referencias a instancias en memoria, haciendo al cierre completo persistente.

Eliminar todas las referencias a instancias persistentes no elimina automáticamente a la instancia. Necesita eliminar las instancias de forma explícita, lo cual lo veremos en la siguiente sección. Si crear una referencia desde una instancia persistente a una instancia efímera (transient) durante una transacción, pero cambia esa referencia y no quedan instancias persistentes que hagan referencia a ésta en el commit se mantiene efímera.

La persistencia por alcance le permite escribir su software sin tener llamadas explícitas a interfaces JDO para almacenar instancias. Su software puede centrarse en establecer relaciones entre las instancias en memoria, y la implementación JDO se encarga de almacenar cualquier nueva instancia o relación que establezca entre instancias en memoria. Sus aplicaciones pueden construir grafos realmente complejos en memoria y hacerlos persistentes creando simplemente una referencia al grafo desde una instancia persistente.

Eliminando instancias

Examinemos ahora una aplicación que elimina algunas instancias del almacén de datos. En el Ejemplo 1-15 la aplicación `DeleteMovie` se utiliza para eliminar una instancia de `Movie`. El título de la película a eliminar se pasa como argumento al programa. La línea intenta acceder a la instancia de `Movie`. Si no existe ninguna película con ese título, la aplicación informa de un error y retorna. En la línea **[1]** llamamos a `deletePersistent`() para eliminar la instancia de `Movie` en sí.

Ejemplo 1-15. Eliminación de una película del almacén de datos.

```
package com.mediamania.prototype;

import java.util.Collection;
import java.util.Set;
import java.util.Iterator;
import javax.jdo.PersistenceManager;
import com.mediamania.MediaManiaApp;

public class DeleteMovie extends MediaManiaApp {
   private String movieTitle;

   public static void main(String[] args) {
      String title = args[0];
      DeleteMovie deleteMovie = new DeleteMovie(title);
      deleteMovie.executeTransaction( );
   }

   public DeleteMovie(String title) {
      movieTitle = title;
   }

   public void execute( ) {
[1]   Movie movie = PrototypeQueries.getMovie(pm, movieTitle);
      if (movie == null) {
         System.err.print("Could not access movie with title of ");
         System.err.println(movieTitle);
         return;
      }
[2]   Set cast = movie.getCast( );
      Iterator iter = cast.iterator( );
      while (iter.hasNext( )) {
         Role role = (Role) iter.next( );
[3]      Actor actor = role.getActor( );
[4]      actor.removeRole(role);
      }
[5]   pm.deletePersistentAll(cast);
[6]   pm.deletePersistent(movie);
   }
}
```

Pero también es necesario eliminar las instancias de `Role` asociadas a la película. Adicionalmente, ya que un `Actor` incluye una referencia a una instancia de `Role`, es necesario eliminar esta referencia. En la línea **[2]** accedemos al conjunto de instancias de `Role` asociadas a `Movie`. A continuación iteramos por cada instancia `Role` y accedemos a la instancia `Actor` asociada en la línea **[3]**. Puesto que vamos a eliminar la instancia de `Role`, en la línea eliminamos la referencia de actor a la instancia de `Role`. En la línea hacemos una llamada a `deletePersistentAll`() para eliminar todas las instancias de `Role` en el reparto de la película. Al realizar el commit de la transacción, la instancia de `Movie` y sus instancias de `Role` asociadas se eliminan del almacén de datos, y las instancias asociadas a la película se actualizan de manera que dejan de referenciar a las instancias `Role` eliminadas.

Tiene que llamar a estos métodos `deletePersistent()` de forma explícita para eliminar instancias del almacén de datos. No son la versión inversa de `makePersistent()`, que utiliza el algoritmo de persistencia por alcance. Es más, no existe ningún almacén de datos JDO equivalente a la recolección de basura de Java, que elimina instancias automáticamente una vez que ya no son referenciadas por otras instancias. Implementar el equivalente de un recolector de basura es una tarea muy compleja, y este tipo de sistemas padecen muchas veces de prestaciones pobres.

RESUMEN

Como puede ver, una gran parte de una aplicación puede ser escrita de forma completamente independiente de JDO utilizando técnicas de modelado, sintaxis y programación convencionales de Java. Puede definir el modelo de información persistente de su aplicación limitándose a términos de un modelo de objetos Java. Una vez que accede a instancias desde un almacén de datos vía un extent o una consulta, su software no tendrá un aspecto diferente del software que accede a instancias en memoria. No necesita aprender ningún otro modelo de datos o lenguaje de acceso como SQL. No necesita pensar cómo crear una correlación para sus datos entre su representación en la base de datos y en objetos en memoria. Puede explotar plenamente las capacidades orientadas a objetos de Java sin ninguna limitación. Esto incluye el uso de herencia y polimorfismo, que no son posibles utilizando tecnologías como JDBC o la arquitectura Enterprise JavaBeans (EJB). Además, puede desarrollar una aplicación utilizando un modelo de objetos con mucho menos software que utilizando arquitecturas complejas. Puede almacenar objetos Java simples, normales en un almacén de datos y acceder a ellos de forma transparente. JDO aporta un entorno productivo y muy sencillo de aprender para construir aplicaciones Java que manejan información persistente.