

Store Objects with Ease



Java™

Data Objects

O'REILLY®

David Jordan & Craig Russell

Java™ Data Objects

Related titles from 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

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

O'REILLY®

An Initial Tour

Java is a language that defines a runtime environment in which user-defined classes execute. Instances of these user-defined classes may represent real-world data that is stored in a database, filesystem, or mainframe transaction processing system. Additionally, small-footprint environments often require a means of managing persistent data in local storage.

Because data-access techniques are different for each type of data source, accessing the data presents a challenge to application developers, who need to use a different application programming interface (API) for each type of data source. This means that you need to know at least two languages to develop business logic for these data sources: the Java programming language and the specialized data-access language required by the data source. The data-access language is likely to be different for each data source, driving up the costs to learn and use each data source.

Prior to the release of Java Data Objects (JDO), three standards existed for storing Java data: serialization, Java DataBase Connectivity (JDBC), and Enterprise JavaBeans (EJB) Container Managed Persistence (CMP). Serialization is used to write the state of an object, and the graph of objects it references, to an output stream. It preserves the relationships of Java objects such that the complete graph can be reconstructed at a later point in time. But serialization does not support transactions, queries, or the sharing of data among multiple users. It allows access only at the granularity of the original serialization and becomes cumbersome when the application needs to manage multiple serializations. Serialization is only used for persistence in the simplest of applications or in embedded environments that cannot support a database effectively.

JDBC requires you to manage the values of fields explicitly and map them into relational database tables. The developer is forced to deal with two very different data-model, language, and data-access paradigms: Java and SQL's relational data model. The development effort to implement your own mapping between the relational data model and your Java object model is so great that most developers never define an

object model for their data; they simply write procedural Java code to manipulate the tables of the underlying relational database. The end result is that they are not benefiting from the advantages of object-oriented development.

The EJB component architecture is designed to support distributed object computing. It also includes support for persistence through Container Managed Persistence (CMP). Largely due to their distributed capabilities, EJB applications are more complex and have more overhead than JDO. However, JDO has been designed so that implementations can provide persistence support in an EJB environment by integrating with EJB containers. If your application needs object persistence, but does not need distributed object capabilities, you can use JDO instead of EJB components. The most popular use of JDO in an EJB environment is to have EJB session beans directly manage JDO objects, avoiding the use of Entity Beans. EJB components must be run in a managed, application-server environment. But JDO applications can be run in either managed or nonmanaged environments, providing you with the flexibility to choose the most appropriate environment to run your application.

You can develop applications more productively if you can focus on designing Java object models and using JDO to store instances of your classes directly. You need to deal with only a single information model. JDBC requires you to understand the relational model and the SQL language. When using EJB CMP, you are also forced to learn and deal with many other aspects of its architecture. It also has modeling limitations not present in JDO.

JDO specifies the contracts between your persistent classes and the JDO runtime environment. JDO is engineered to support a wide variety of data sources, including sources that are not commonly considered databases. We therefore use the term *datastore* to refer to any underlying data source that you access with JDO.

This chapter explores some of JDO's basic capabilities, by examining a small application developed by a fictitious company called Media Mania, Inc. They rent and sell various forms of entertainment media in stores located throughout the United States. Their stores have kiosks that provide information about movies and the actors in those movies. This information is made available to the customers and store staff to help select merchandise that will be of interest to the customers.

Defining a Persistent Object Model

Figure 1-1 is a Unified Modeling Language (UML) diagram of the classes and interrelationships in the Media Mania object model. A *Movie* instance represents a particular movie. Each actor who has played a role in at least one movie is represented by an instance of *Actor*. The *Role* class represents the specific roles an actor has played in a movie and thus represents a relationship between *Movie* and *Actor* that includes an attribute (the name of the role). Each movie has one or more roles. An actor may have played a role in more than one movie or may have played multiple roles in a single movie.

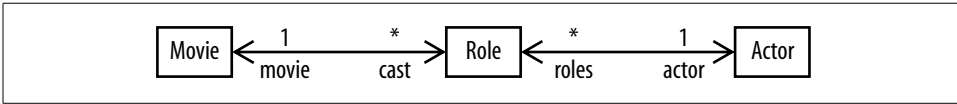


Figure 1-1. UML diagram of the Media Mania object model

We will place these persistent classes and the application programs used to manage their instances in the Java `com.mediamania.prototype` package.

The Classes to Persist

We will make the `Movie`, `Actor`, and `Role` classes persistent, so their instances can be stored in a datastore. First we will examine the complete source code for each of these classes. An import statement is included for each class, so it is clear which package contains each class used in the example.

Example 1-1 provides the source code for the `Movie` class. JDO is defined in the `javax.jdo` package. Notice that the class does not require you to import any JDO-specific classes. Java references and collections defined in the `java.util` package are used to represent the relationships between our classes, which is the standard practice used by most Java applications.

The fields of the `Movie` class use standard Java types such as `String`, `Date`, and `int`. You can declare fields to be private; it is not necessary to define a public get and set method for each field. The `Movie` class includes some methods to get and set the private fields in the class, though those methods are used by other parts of the application and are not required by JDO. You can use encapsulation, providing only the methods that support the abstraction being modeled. The class also has static fields; these are not stored in the datastore.

The `genres` field is a `String` that contains the genres of the movie (action, romance, mystery, etc.). A `Set` interface is used to reference a set of `Role` instances, representing the movie's cast. The `addRole()` method adds elements to the `cast` collection, and `getCast()` returns an unmodifiable `Set` containing the elements of the `cast` collection. These methods are not a JDO requirement, but they are implemented as convenience methods for the application. The `parseReleaseDate()` and `formatReleaseDate()` methods are used to standardize the format of the movie's release date. To keep the code simple, a `null` is returned if the `parseReleaseDate()` parameter is in the wrong format.

Example 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;
  
```

Example 1-1. Movie.java (continued)

```
import java.text.SimpleDateFormat;
import java.text.ParsePosition;

public class Movie {
    private static SimpleDateFormat yearFmt = new SimpleDateFormat("yyyy");
    private 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;    // element type: Role

    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);
    }
}
```

Example 1-1. Movie.java (continued)

```
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 imposes one requirement to make a class persistent: a no-arg constructor. If you do not define any constructors in your class, the compiler generates a no-arg constructor. However, this constructor is not generated if you define any constructors with arguments; in this case, you need to provide a no-arg constructor. You can declare it to be private if you do not want your application code to use it. Some JDO implementations can generate one for you, but this is an implementation-specific, nonportable feature.

Example 1-2 provides the source for the Actor class. For our purposes, all actors have a unique name that identifies them. It can be a stage name that is distinct and different from the given name. Therefore, we represent the actor's name by a single String. Each actor has played one or more roles, and the roles member models the Actor's side of the relationship between Actor and Role. The comment on line ❶ is used merely for documentation; it does not serve any functional purpose in JDO. The addRole() and removeRole() methods in lines ❷ and ❸ are provided so that the application can maintain the relationship from an Actor instance and its associated Role instances.

Example 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;
    ❶ private Set    roles; // element type: Role

    private Actor()
    { }
    public Actor(String name) {
        this.name = name;
        roles = new HashSet();
    }
    public String getName() {
        return name;
    }
}
```

Example 1-2. Actor.java (continued)

```
❷    public void addRole(Role role) {
        roles.add(role);
    }
❸    public void removeRole(Role role) {
        roles.remove(role);
    }
    public Set getRoles() {
        return Collections.unmodifiableSet(roles);
    }
}
```

Finally, Example 1-3 provides the source for the Role class. This class models the relationship between a Movie and Actor and includes the specific name of the role played by the actor in the movie. The Role constructor initializes the references to Movie and Actor, and it also updates the other ends of its relationship by calling `addRole()`, which we defined in the Movie and Actor classes.

Example 1-3. Role.java

```
package com.mediamania.prototype;

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

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

We have now examined the complete source code for each class that will have instances in the datastore. These classes did not need to import and use any JDO-specific types. Furthermore, except for providing a no-arg constructor, no data or methods needed to be defined to make these classes persistent. The software used to

access and modify fields and define and manage relationships among instances corresponds to the standard practice used in most Java applications.

Declaring Classes to Be Persistent

It is necessary to identify which classes should be persistent and specify any persistence-related information that is not expressible in Java. JDO uses a metadata file in XML format to specify this information.

You can define metadata on a class or package basis, in one or more XML files. The name of the metadata file for a single class is the name of the class, followed by a *.jdo* suffix. So, a metadata file for the *Movie* class would be named *Movie.jdo* and placed in the same directory as the *Movie.class* file. A metadata file for a Java package is contained in a file named *package.jdo*. A metadata file for a Java package can contain metadata for multiple classes and multiple subpackages. Example 1-4 provides the metadata for the Media Mania object model. The metadata is specified for the package and contained in a file named *com/mediamania/prototype/package.jdo*.

Example 1-4. JDO metadata in the file prototype/package.jdo

```
<?xml version="1.0" encoding="UTF-8" ?>
❶ <!DOCTYPE jdo PUBLIC
    "-//Sun Microsystems, Inc.//DTD Java Data Objects Metadata 1.0//EN"
    "http://java.sun.com/dtd/jdo_1_0.dtd">
<jdo>
❷   <package name="com.mediamaia.prototype" >
❸     <class name="Movie" >
❹       <field name="cast" >
❺         <collection element-type="Role"/>
       </field>
     </class>
❻   <class name="Role" />
     <class name="Actor" >
       <field name="roles" >
         <collection element-type="Role"/>
       </field>
     </class>
  </package>
</jdo>
```

The *jdo_1_0.dtd* file specified on line ❶ provides a description of the XML elements that can be used in a JDO metadata file. This document type definition (DTD) is standardized in JDO and should be provided with a JDO implementation. It is also available for download at <http://java.sun.com/dtd>. You can also alter the DOCTYPE to refer to a local copy in your filesystem.

The metadata file can contain persistence information for one or more packages that have persistent classes. Each package is defined with a package element, which includes the name of the Java package. Line ❷ provides a package element for our

`com.mediamania.prototype` package. Within the package element are nested class elements that identify a persistent class of the package (e.g., line ❸ has the `class` element for the `Movie` class). The file can contain multiple package elements listed serially; they are not nested.

If information must be specified for a particular field of a class, a `field` element is nested within the `class` element, as shown on line ❹. For example, you could declare the element type for each collection in the model. This is not required, but it can result in a more efficient mapping. The `Movie` class has a collection named `cast`, and the `Actor` class has a collection named `roles`; both contain `Role` references. Line ❺ specifies the element type for `cast`. In many cases, a default value for an attribute is assumed in the metadata that provides the most commonly needed value.

All of the fields that can be persistent are made persistent by default. Static and final fields cannot be made persistent. A field declared in Java to be transient is not persistent by default, but such a field can be declared as persistent in the metadata file. Chapter 4 describes this capability.

Chapters 4, 10, 12, and 13 cover other characteristics you can specify for classes and fields. For a simple class like `Role`, which does not have any collections, you can just list the class in the metadata as shown on line ❻, if no other metadata attributes are necessary.

Project Build Environment

In this section, we examine a development environment to compile and run our JDO application. This includes the project directory structure, the jar files necessary to build applications, and the syntax for enhancing persistent classes. We describe class enhancement later in this section. The environment setup partly depends on which JDO implementation you use. Your specific project's development environment and directory structure may differ.

You can use either the Sun JDO reference implementation or another implementation of your choosing. The examples in this book use the JDO reference implementation. You can download the JDO reference implementation by visiting <http://www.jcp.org> and selecting JSR-12. Once you have installed a JDO implementation, you will need to establish a project directory structure and define a classpath that includes all the directories and jar files necessary to build and run your application.

JDO introduces a new step in your build process, called *class enhancement*. Each persistent class must be enhanced so that it can be used in a JDO runtime environment. Your persistent classes are compiled using a Java compiler that produces a class file. An enhancer program reads these class files and JDO metadata and creates new class files that have been enhanced to operate in a JDO environment. Your JDO application should load these enhanced class files. The JDO reference implementation includes an enhancer called the *reference enhancer*.

Jars Needed to Use the JDO Reference Implementation

When using the JDO reference implementation, you should include the following jar files in your classpath during development. At runtime, all of these jar files should be in your classpath.

jdo.jar

The standard interfaces and classes defined in the JDO specification.

jdori.jar

Sun's reference implementation of the JDO specification.

btree.jar

Software used by the JDO reference implementation to manage the storage of data in a file. The reference implementation uses a file for the storage of persistent instances.

jta.jar

The Java Transaction API. The Synchronization interface defined in package `javax.transaction` is used in the JDO interface and contained in this jar file. Other facilities defined in this file are likely to be useful to a JDO implementation. You can download this jar from <http://java.sun.com/products/jta/index.html>.

antlr.jar

Parsing technology used in the implementation of the JDO query language. The reference implementation uses Version 2.7.0 of Antlr. You can download it from <http://www.antlr.org>.

xerces.jar

The reference implementation uses Xerces-J Release 1.4.3 to parse XML. It can be downloaded from <http://xml.apache.org/xerces-j/>.

The first three jar files are included with the JDO reference implementation; the last three can be downloaded from the specified web sites.

The reference implementation includes an additional jar, *jdori-enhancer.jar*, that contains the reference enhancer implementation. The classes in *jdori-enhancer.jar* are also in *jdori.jar*. In most cases, you will use *jdori.jar* in both your development and runtime environment, and not need *jdori-enhancer.jar*. The *jdori-enhancer.jar* is packaged separately so that you can enhance your classes using the reference enhancer independent of a particular JDO implementation. Some implementations, besides the reference implementation, may distribute this jar for use with their implementation.

If you use a different JDO implementation, its documentation should provide you with a list of all the necessary jars. An implementation usually places all the necessary jars in a particular directory in their installation. The *jdo.jar* file containing the interfaces defined in JDO should be used with all implementations. This jar file is usually included with a vendor's implementation. JDOcentral.com (<http://www.jdocentral.com>) provides numerous JDO resources, including free trial downloads of many commercial JDO implementations.

Project Directory Structure

You should use the following directory structure for the Media Mania application development environment. The project must have a *root* directory placed somewhere in the filesystem. The following directories reside beneath the project's *root* directory:

src

This directory contains all of the application's source code. Under *src*, there is a subdirectory hierarchy of *com/mediamania/prototype* (corresponding to the Java `com.mediamania.prototype` package). This is where the *Movie.java*, *Actor.java*, and *Role.java* source files reside.

classes

When the Java source files are compiled, their class files are placed in this directory.

enhanced

This is the directory that contains the enhanced class files (produced by the enhancer).

database

This directory contains the files used by the reference implementation to store our persistent data.

Though this particular directory structure is not a requirement of JDO or the reference implementation, you need to understand it to follow our description of the Media Mania application.

When you execute your JDO application, the Java runtime must load the enhanced version of the class files, which are located in our *enhanced* directory. Therefore, the *enhanced* directory should be listed prior to the *classes* directory in your classpath. As an alternative approach, you can also enhance in-place, replacing your unenhanced class file with their enhanced form.

Enhancing Classes for Persistence

A class must be enhanced before its instances can be managed in a JDO environment. A JDO enhancer adds data and methods to your classes that enable their instances to be managed by a JDO implementation. An enhancer reads a class file produced by the Java compiler and, using the JDO metadata, produces a new, enhanced class file that includes the necessary functionality. JDO has standardized the modifications made by enhancers so that enhanced class files are binary-compatible and can be used with any JDO implementation. These enhanced files are also independent of any specific datastore.

As mentioned previously, the enhancer provided with Sun's JDO reference implementation is called the *reference enhancer*. A JDO vendor may provide its own enhancer; the command-line syntax necessary to execute an enhancer may differ

from the syntax shown here. Each implementation should provide you with documentation explaining how to enhance your classes for use with their implementation.

Example 1-5 provides the reference enhancer command for enhancing the persistent classes in our Media Mania application. The `-d` argument specifies the *root* directory in which to place the enhanced class files; we have specified our *enhanced* directory. The enhancer is given a list of JDO metadata files and a set of class files to enhance. The directory separator and line-continuation symbols may vary, depending on your operating system and build environment.

Example 1-5. Enhancing the persistent classes

```
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
```

Though it is convenient to place the metadata files in the directory with the source code, the JDO specification recommends that the metadata files be available via resources loaded by the same class loader as the class files. The metadata is needed at both build and runtime. So, we have placed the *package.jdo* metadata file under the *classes* directory hierarchy in the directory for the prototype package.

The class files for all persistent classes in our object model are listed together in Example 1-5, but you can also enhance each class individually. When this command executes, it places new, enhanced class files in the *enhanced* directory.

Establish a Datastore Connection and Transaction

Now that our classes have been enhanced, their instances can be stored in a datastore. We now examine how an application establishes a connection with a datastore and executes operations within a transaction. We begin to write software that makes direct use of the JDO interfaces. All JDO interfaces used by an application are defined in the `javax.jdo` package.

JDO has an interface called `PersistenceManager` that has a connection with a datastore. A `PersistenceManager` has an associated instance of the JDO `Transaction` interface used to control the start and completion of a transaction. The `Transaction` instance is acquired by calling `currentTransaction()` on the `PersistenceManager` instance.

Acquiring a PersistenceManager

A `PersistenceManagerFactory` is used to configure and acquire a `PersistenceManager`. Methods in the `PersistenceManagerFactory` are used to set properties that control the

behavior of the `PersistenceManager` instances acquired from the factory. Therefore, the first step performed by a JDO application is the acquisition of a `PersistenceManagerFactory` instance. To get this instance, call the following static method of the `JDOHelper` class:

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

The `Properties` instance can be populated programmatically or by loading property values from a property file. Example 1-6 lists the contents of the property file we will use in our Media Mania application. The `PersistenceManagerFactoryClass` property on line ❶ specifies which JDO implementation we are using by providing the name of the implementation's class that implements the `PersistenceManagerFactory` interface. In this case, we specify the class defined in Sun's JDO reference implementation. Other properties listed in Example 1-6 include the connection URL used to connect to a particular datastore and a username and password, which may be necessary to establish a connection to the datastore

Example 1-6. Contents of `jdo.properties`

```
❶ 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
```

The format of the connection URL depends on the particular datastore being accessed. The JDO reference implementation has its own storage facility called File Object Store (FOStore). The `ConnectionURL` property in Example 1-6 specifies that the datastore is located in the `database` directory, which is located in our project's *root* directory. In this case, we have provided a relative path; it is also possible to provide an absolute path to the datastore. The URL specifies that the FOStore datastore files will have a name prefix of `fostoredb`.

If you are using a different implementation, you will need to provide different values for these properties. You may also need to provide values for additional properties. Check with your implementation's documentation to determine the properties that are necessary.

Creating a FOStore Datastore

To use FOStore we must first create a datastore. The program in Example 1-7 creates a datastore using the `jdo.properties` file; all applications use this property file. Line ❶ loads the properties from `jdo.properties` into a `Properties` instance. The program adds the `com.sun.jdori.option.ConnectionCreate` property on line ❷ to indicate that the datastore should be created. Setting it to `true` instructs the implementation to create the datastore. We then call `getPersistenceManagerFactory()` on line ❸ to acquire the `PersistenceManagerFactory`. Line ❹ creates a `PersistenceManager`.

To complete the creation of the datastore, we must also begin and commit a transaction. The `PersistenceManager` method `currentTransaction()` is called on line ⑤ to access the `Transaction` instance associated with the `PersistenceManager`. The `Transaction` methods `begin()` and `commit()` are called on lines ⑥ and ⑦ to start and commit a transaction. When you execute this application, a `FStore` datastore is created in the `database` directory. Two files are created: `fostore.btd` and `fostore.btx`.

Example 1-7. Creating a FStore datastore

```
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();
            jdoproperties.load(propertyStream);
            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);
        }
    }
}
```

The JDO reference implementation provides this programmatic means to create a database. Most databases provide a utility separate from JDO for creating a database. JDO does not define a standard, vendor-independent interface for creating a database. Creation of a datastore is always datastore-specific. This program illustrates how it is done using the `FStore` datastore.

In addition, when you are using JDO with a relational database, there is often an additional step of creating or mapping to an existing relational schema. The procedure to

follow for establishing a schema that corresponds with your JDO object model is implementation-specific. You should examine the documentation of the implementation you are using to determine the necessary steps.

Operations on Instances

Now we have a datastore in which we can store instances of our classes. Each application needs to acquire a `PersistenceManager` to access and update the datastore. Example 1-8 provides the source for the `MediaManiaApp` class, which serves as the base class for each application in this book. Each application is a concrete subclass of `MediaManiaApp` that implements its application logic in the `execute()` method.

`MediaManiaApp` has a constructor that loads the properties from `jdo.properties` (line ❶). After loading properties from the file, it calls `getPropertyOverrides()` and merges the returned properties into `jdoproperties`. An application subclass can redefine `getPropertyOverrides()` to provide any additional properties or change properties that are set in the `jdo.properties` file. The constructor gets a `PersistenceManagerFactory` (line ❷) and then acquires a `PersistenceManager` (line ❸). We also provide the `getPersistenceManager()` method to access the `PersistenceManager` from outside the `MediaManiaApp` class. The `Transaction` associated with the `PersistenceManager` is acquired on line ❹.

The application subclasses make a call to `executeTransaction()`, defined in the `MediaManiaApp` class. This method begins a transaction on line ❺. It then calls `execute()` on line ❻, which will execute the subclass-specific functionality.

We chose this particular design for application classes to simplify and reduce the amount of redundant code in the examples for establishing an environment to run. This is not required in JDO; you can choose an approach that is best suited for your application environment.

After the return from the `execute()` method (implemented by a subclass), an attempt is made to commit the transaction (line ❼). If any exceptions are thrown, the transaction is rolled back and the exception is printed to the error stream.

Example 1-8. MediaManiaApp base class

```
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;
```

Example 1-8. *MediaManiaApp* base class (continued)

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

    public abstract void execute(); // defined in concrete application subclasses

    protected static Map getPropertyOverrides() {
        return new HashMap();
    }
    public MediaManiaApp() {
        try {
            ❶ InputStream propertyStream = new FileInputStream("jdo.properties");
            Properties jdoproperties = new Properties();
            ❷ jdoproperties.load(propertyStream);
            ❸ jdoproperties.putAll(getPropertyOverrides());
            ❹ pmf = JDOHelper.getPersistenceManagerFactory(jdoproperties);
            pm = pmf.getPersistenceManager();
            tx = pm.currentTransaction();
        } catch (Exception e) {
            e.printStackTrace(System.err);
            System.exit(-1);
        }
    }
    public PersistenceManager getPersistenceManager() {
        return pm;
    }
    public void executeTransaction() {
        try {
            ❺ tx.begin();
            ❻ execute();
            ❼ tx.commit();
        } catch (Throwable exception) {
            exception.printStackTrace(System.err);
            if (tx.isActive()) tx.rollback();
        }
    }
}
```

Making Instances Persistent

Let's examine a simple application, called *CreateMovie*, that makes a single *Movie* instance persistent, as shown in Example 1-9. The functionality of the application is placed in `execute()`. After constructing an instance of *CreateMovie*, we call `executeTransaction()`, which is defined in the *MediaManiaApp* base class. It makes a call to `execute()`, which will be the method defined in this class. The `execute()` method instantiates a single *Movie* instance on line ❺. Calling the *PersistenceManager* method `makePersistent()` on line ❻ makes the *Movie* instance persistent. If the transaction commits successfully in `executeTransaction()`, the *Movie* instance will be stored in the datastore.

Example 1-9. Creating a Movie instance and making it persistent

```
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);
    }
}
```

Now let's examine a larger application. `LoadMovies`, shown in Example 1-10, reads a file containing movie data and creates multiple instances of `Movie`. The name of the file is passed to the application as an argument, and the `LoadMovies` constructor initializes a `BufferedReader` to read the data. The `execute()` method reads one line at a time from the file and calls `parseMovieData()`, which parses the line of input data, creates a `Movie` instance on line ❶, and makes it persistent on line ❷. When the transaction commits in `executeTransaction()`, all of the newly created `Movie` instances will be stored in the datastore.

Example 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.persistence.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();
    }
}
```

Example 1-10. LoadMovies (continued)

```
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();
    Movie movie = new Movie(title, releaseDate, runningTime, rating, genres);
    pm.makePersistent(movie);
}
}
```

The movie data is in a file with the following format:

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

The format to use for release dates is maintained in the `Movie` class, so `parseReleaseDate()` is called to create a `Date` instance from the input data. A movie is described by one or more genres, which are listed at the end of the line of data.

Accessing Instances

Now let's access the `Movie` instances in the datastore to verify that they were stored successfully. There are several ways to access instances in JDO:

- Iterate an extent
- Navigate the object model
- Execute a query

An *extent* is a facility used to access all the instances of a particular class or the class and all its subclasses. If the application wants to access only a subset of the instances, a query can be executed with a filter that constrains the instances returned to those that satisfy a Boolean predicate. Once the application has accessed an instance from the datastore, it can navigate to related instances in the datastore by traversing through references and iterating collections in the object model. Instances that are not yet in memory are read from the datastore on demand. These facilities for accessing instances are often used in combination, and JDO ensures that each persistent instance is represented in the application memory only once per `PersistenceManager`. Each `PersistenceManager` manages a single transaction context.

Iterating an extent

JDO provides the `Extent` interface for accessing the extent of a class. The extent allows access to all of the instances of a class, but using an extent does not imply that all the instances are in memory. The `PrintMovies` application, provided in Example 1-11, uses the `Movie` extent.

Example 1-11. Iterating the `Movie` extent

```
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() {
        ❶ Extent extent = pm.getExtent(Movie.class, true);
        ❷ Iterator iter = extent.iterator();
        while (iter.hasNext()) {
            ❸ Movie movie = (Movie) iter.next();
                System.out.print(movie.getTitle());
                System.out.print(movie.getRating());
                System.out.print(";");
                System.out.print(";");
        }
    }
}
```

Example 1-11. Iterating the Movie extent (continued)

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

On line ❶ we acquire an Extent for the `Movie` class from the `PersistenceManager`. The second parameter indicates whether to include instances of `Movie` subclasses. A value of `false` causes only `Movie` instances to be returned, even if there are instances of subclasses. Though we don't currently have any classes that extend the `Movie` class, providing a value of `true` will return instances of any such classes that we may define in the future. The Extent interface has the `iterator()` method, which we call on line ❷ to acquire an `Iterator` that will access each element of the extent. Line ❸ uses the `Iterator` to access `Movie` instances. The application can then perform operations on the `Movie` instance to acquire data about the movie to print. For example, on line ❹ we call `getGenres()` to get the genres associated with the movie. On line ❺ we acquire the set of `Roles`. We acquire a reference to a `Role` on line ❻ and then print the role's name. On line ❼ we navigate to the `Actor` for that role by calling `getActor()`, which we defined in the `Role` class. We then print the actor's name.

Once the application has completed iteration through the extent, line ❽ closes the `Iterator` to relinquish any resources required to perform the extent iteration. Multiple `Iterator` instances can be used concurrently on an Extent. This method closes a specific `Iterator`; `closeAll()` closes all the `Iterator` instances associated with an Extent.

Navigating the object model

Example 1-11 demonstrates iteration of the `Movie` extent. But on line ❻ we also navigate to a set of related `Role` instances by iterating a collection in our object model. On line ❼ we use the `Role` instance to navigate through a reference to the related `Actor` instance. Line ❺ and ❼ demonstrate, respectively, traversal of *to-many* and *to-one* relationships. A relationship from one class to another has a cardinality that indicates whether there are one or multiple associated instances. A *reference* is used for a cardinality of one, and a *collection* is used when there can be more than one instance.

The syntax needed to access these related instances corresponds to the standard practice of navigating instances in memory. The application does not need to make any direct calls to JDO interfaces between lines ❸ and ❷. It simply traverses among objects in memory. The related instances are not read from the datastore and instantiated in memory until they are accessed directly by the application. Access to the datastore is transparent; instances are brought into memory on demand. Some implementations provide facilities separate from the Java interface that allow you to influence the implementation's access and caching algorithms. Your Java application is insulated from these optimizations, but it can take advantage of them to affect its overall performance.

The access of related persistent instances in a JDO environment is identical to the access of transient instances in a non-JDO environment, so you can write your software in a manner that is independent of its use in a JDO environment. Existing software written without any knowledge of JDO or any other persistence concerns is able to navigate objects in the datastore through JDO. This capability yields dramatic increases in development productivity and allows existing software to be incorporated into a JDO environment quickly and easily.

Executing a query

It is also possible to perform a query on an Extent. The JDO Query interface is used to select a subset of the instances that meet certain criteria. The remaining examples in this chapter need to access a specific Actor or Movie based on a unique name. These methods, shown in Example 1-12, are virtually identical; `getActor()` performs a query to get an Actor based on a name, and `getMovie()` performs a query to get a Movie based on a name.

Example 1-12. Query methods in the `PrototypeQueries` class

```
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)
    {
        ❶ Extent actorExtent = pm.getExtent(Actor.class, true);
        ❷ Query query = pm.newQuery(actorExtent, "name == actorName");
        ❸ query.declareParameters("String actorName");
        ❹ Collection result = (Collection) query.execute(actorName);
        Iterator iter = result.iterator();
        Actor actor = null;
        ❺ if (iter.hasNext()) actor = (Actor)iter.next();
        ❻ query.close(result);
    }
}
```

Example 1-12. Query methods in the `PrototypeQueries` class (continued)

```
        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;
    }
}
```

Let's examine `getActor()`. On line ❶ we get a reference to the `Actor` extent. Line ❷ creates an instance of `Query` using the `newQuery()` method defined in the `PersistenceManager` interface. The query is initialized with the extent and a query filter to apply to the extent.

The name identifier in the filter is the name field in the `Actor` class. The namespace used to determine how to interpret the identifier is based on the class of the `Extent` used to initialize the `Query` instance. The filter expression requires that an `Actor`'s name field is equal to `actorName`. In the filter we can use the `==` operator directly to compare two `Strings`, instead of using the Java syntax (`name.equals(actorName)`).

The `actorName` identifier is a *query parameter*, which is declared on line ❸. A query parameter lets you provide a value to be used when the query is executed. We have chosen to use the same name, `actorName`, for the method parameter and query parameter. This practice is not required, and there is no direct association between the names of our Java method parameters and our query parameters. The query is executed on line ❹, passing `getActor()`'s `actorName` parameter as the value to use for the `actorName` query parameter.

The result type of `Query.execute()` is declared as `Object`. In JDO 1.0.1, the returned instance is always a `Collection`, so we cast the query result to a `Collection`. It is declared in JDO 1.0.1 to return `Object`, to allow for a future extension of returning a value other than a `Collection`. Our method then acquires an `Iterator` and, on line ❺, attempts to access an element. We assume here that there can only be a single `Actor` instance with a given name. Before returning the result, line ❻ closes the query result to relinquish any associated resources. If the method finds an `Actor` instance with the given name, the instance is returned. Otherwise, if the query result has no elements, a `null` is returned.

Modifying an Instance

Now let's examine two applications that modify instances in the datastore. Once an application has accessed an instance from the datastore in a transaction, it can modify one or more fields of the instance. When the transaction commits, all modifications that have been made to instances are propagated to the datastore automatically.

The `UpdateWebSite` application provided in Example 1-13 is used to set the web site associated with a movie. It takes two arguments: the first is the movie's title, and the second is the movie's web site URL. After initializing the application instance, `executeTransaction()` is called, which calls the `execute()` method defined in this class.

Line ❶ calls `getMovie()` (defined in Example 1-12) to retrieve the `Movie` with the given title. If `getMovie()` returns `null`, the application reports that it could not find a `Movie` with the given title and returns. Otherwise, on line ❷ we call `setWebSite()` (defined for the `Movie` class in Example 1-1), which sets the `webSite` field of `Movie` to the parameter value. When `executeTransaction()` commits the transaction, the modification to the `Movie` instance is propagated to the datastore automatically.

Example 1-13. Modifying an attribute

```
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() {
❶      Movie movie = PrototypeQueries.getMovie(pm, movieTitle);
        if (movie == null) {
            System.err.print("Could not access movie with title of ");
            System.err.println(movieTitle);
            return;
        }
❷      movie.setWebSite(newWebSite);
    }
}
```

As you can see in Example 1-13, the application does not need to make any direct JDO interface calls to modify the `Movie` field. This application accesses an instance and calls a method to modify the `web site` field. The method modifies the field using standard Java syntax. No additional programming is necessary prior to commit in order to propagate the data to the datastore. The JDO environment propagates the modifications automatically. This application performs an operation on persistent instances, yet it does not directly import or use any JDO interfaces.

Now let's examine a larger application, called `LoadRoles`, that exhibits several JDO capabilities. `LoadRoles`, shown in Example 1-14, is responsible for loading information about the movie roles and the actors who play them. `LoadRoles` is passed a single argument that specifies the name of a file to read, and the constructor initializes a `BufferedReader` to read the file. It reads the text file, which contains one role per line, in the following format:

```
movie title;actor's name;role name
```

Usually, all the roles associated with a particular movie are grouped together in this file; `LoadRoles` performs a small optimization to determine whether the role information being processed is for the same movie as the previous role entry in the file.

Example 1-14. Instance modification and persistence-by-reachability

```
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();
    }

    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 {
```

Example 1-14. Instance modification and persistence-by-reachability (continued)

```
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)) {
❶      movie = PrototypeQueries.getMovie(pm, title);
        if (movie == null) {
            System.err.print("Movie title not found: ");
            System.err.println(title);
            continue;
        }
        lastTitle = title;
    }
❷      Actor actor = PrototypeQueries.getActor(pm, actorName);
    if (actor == null) {
❸          actor = new Actor(actorName);
❹          pm.makePersistent(actor);
    }
❺      Role role = new Role(roleName, actor, movie);
}
} catch (java.io.IOException e) {
    System.err.println("Exception reading input file");
    System.err.println(e);
    return;
}
}
```

The `execute()` method reads each entry in the file. First, it checks to see whether the new entry's movie title is the same as the previous entry. If it is not, line ❶ calls `getMovie()` to access the `Movie` with the new title. If a `Movie` with that title does not exist in the datastore, the application prints an error message and skips over the entry. On line ❷ we attempt to access an `Actor` instance with the specified name. If no `Actor` in the datastore has this name, a new `Actor` is created and given this name on line ❸, and made persistent on line ❹.

Up to this point in the application, we have just been reading the input file and looking up instances in the datastore that have been referenced by a name in the file. We perform the real task of the application on line ❺, where we create a new `Role` instance. The `Role` constructor was defined in Example 1-3; it is repeated here so that we can examine it in more detail:

```
public Role(String name, Actor actor, Movie movie) {
❶     this.name = name;
❷     this.actor = actor;
❸     this.movie = movie;
❹     actor.addRole(this);
❺     movie.addRole(this);
}
```

Line ❶ initializes the name of the Role. Line ❷ establishes a reference to the associated Actor, and line ❸ establishes a reference to the associated Movie instance. The relationships between Actor and Role and between Movie and Role are bidirectional, so it is also necessary to update the other side of each relationship. On line ❹ we call `addRole()` on actor, which adds this Role to the `roles` collection in the Actor class. Similarly, line ❺ calls `addRole()` on movie to add this Role to the `cast` collection field in the Movie class. Adding the Role as an element in `Actor.roles` and `Movie.cast` causes a modification to the instances referenced by actor and movie.

The Role constructor demonstrates that you can establish a relationship to an instance simply by initializing a reference to it, and you can establish a relationship with more than one instance by adding references to a collection. This process is how relationships are represented in Java and is supported directly by JDO. When the transaction commits, the relationships established in memory are preserved in the datastore.

Upon return from the Role constructor, `load()` processes the next entry in the file. The `while` loop terminates once we have exhausted the contents of the file.

You may have noticed that we never called `makePersistent()` on the Role instances we created. Still, at commit, the Role instances are stored in the datastore because JDO supports *persistence-by-reachability*. Persistence-by-reachability causes any transient (nonpersistent) instance of a persistent class to become persistent at commit if it is reachable (directly or indirectly) by a persistent instance. Instances are reachable through either a reference or collection of references. The set of all instances reachable from a given instance is an object graph that is called the instance's *complete closure* of related instances. The reachability algorithm is applied to all persistent instances transitively through all their references to instances in memory, causing the complete closure to become persistent.

Removing all references to a persistent instance does not automatically delete the instance. You need to delete instances explicitly, which we cover in the next section. If you establish a reference from a persistent instance to a transient instance during a transaction, but you change this reference and no persistent instances reference the transient instance at commit, it remains transient.

Persistence-by-reachability lets you write a lot of your software without having any explicit calls to JDO interfaces to store instances. Much of your software can focus on establishing relationships among the instances in memory, and the JDO implementation takes care of storing any new instances and relationships you establish among the instances in memory. Your applications can construct fairly complex object graphs in memory and make them persistent simply by establishing a reference to the graph from a persistent instance.

Deleting Instances

Now let's examine an application that deletes some instances from the datastore. In Example 1-15, the `DeleteMovie` application is used to delete a `Movie` instance. The title of the movie to delete is provided as the argument to the program. Line ❶ attempts to access the `Movie` instance. If no movie with the title exists, the application reports an error and returns. On line ❷ we call `deletePersistent()` to delete the `Movie` instance itself.

Example 1-15. Deleting a Movie from the datastore

```
package com.mediamania.prototype;

import java.util.Collection;
import java.util.Set;
import java.util.Iterator;
import javax.persistence.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() {
❶      Movie movie = PrototypeQueries.getMovie(pm, movieTitle);
        if (movie == null) {
            System.err.print("Could not access movie with title of ");
            System.err.println(movieTitle);
            return;
        }
❷      Set cast = movie.getCast();
        Iterator iter = cast.iterator();
        while (iter.hasNext()) {
            Role role = (Role) iter.next();
❸          Actor actor = role.getActor();
❹          actor.removeRole(role);
        }
❺      pm.deletePersistentAll(cast);
❻      pm.deletePersistent(movie);
    }
}
```

But it is also necessary to delete the `Role` instances associated with the `Movie`. In addition, since an `Actor` includes a reference to the `Role` instance, it is necessary to remove this reference. On line ❷ we access the set of `Role` instances associated with the

Movie. We then iterate through each Role and access the associated Actor on line ③. Since we will be deleting the Role instance, on line ④ we remove the actor's reference to the Role. On line ⑤ we make a call to `deletePersistentAll()` to delete all the Role instances in the movie's cast. When we commit the transaction, the Movie instance and associated Role instances are deleted from the datastore, and the Actor instances associated with the Movie are updated so that they no longer reference the deleted Role instances.

You must call these `deletePersistent()` methods explicitly to delete instances from the datastore. They are not the inverse of `makePersistent()`, which uses the persistence-by-reachability algorithm. Furthermore, there is no JDO datastore equivalent to Java's garbage collection, which deletes instances automatically once they are no longer referenced by any instances in the datastore. Implementing the equivalent of a persistent garbage collector is a very complex undertaking, and such systems often have poor performance.

Summary

As you can see, a large portion of an application can be written in a completely JDO-independent manner using conventional Java modeling, syntax, and programming techniques. You can define your application's persistent information model solely in terms of a Java object model. Once you access instances from the datastore via an extent or query, your software looks no different from any other Java software that accesses instances in memory. You do not need to learn any other data model or access language like SQL. You do not need to figure out how to provide a mapping of your data between a database representation and an in-memory object representation. You can fully exploit the object-oriented capabilities of Java without any limitation. This includes use of inheritance and polymorphism, which are not possible using technologies like JDBC and the Enterprise JavaBeans (EJB) architecture. In addition, you can develop an application using an object model with much less software than when using competitive architectures. Plain, ordinary Java objects can be stored in a datastore and accessed in a transparent manner. JDO provides a very easy-to-learn and productive environment to build Java applications that manage persistent data.