Java Persistence for Relational Databases

RICHARD SPERKO

Apress™

Java Persistence for Relational Databases
Copyright ©2003 by Richard Sperko

ISBN (pbk): 1-59059-071-6

Printed and bound in the United States of America 12345678910

Trademarked names may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, we use the names only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

Distributed to the book trade in the United States by Springer-Verlag New York, Inc., 175 Fifth Avenue, New York, NY, 10010 and outside the United States by Springer-Verlag GmbH & Co. KG, Tiergartenstr. 17, 69112 Heidelberg, Germany.

In the United States, phone 1-800-SPRINGER, email orders@springer-ny.com, or visit http://www.springer-ny.com. Outside the United States, fax +49 6221 345229, email orders@springer.de, or visit http://www.springer.de.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, email info@apress.com, or visit http://www.apress.com.

The information in this book is distributed on an "as is" basis, without warranty. Although every precaution has been taken in the preparation of this work, neither the author(s) nor Apress shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in this work.

The source code for this book is available to readers at http://www.apress.com in the Downloads section.

CHAPTER 10

Persistence with Java Data Objects (JDO)

BUT THE BASIN OF THE MISSISSIPPI is the BODY OF THE NATION. All the other parts are but members, important in themselves, yet more important in their relations to this. So, too, with how Java Data Objects (JDO) relates to our applications. The goal in software development is to write a piece of software to solve some problem. The tools are primarily important in relation to that solution. JDO is not meant to be the focus, but a transparent, helping framework.

Java Data Objects is Sun's answer to write once, store anywhere. The idea is to remove the need for any type of data storage knowledge and allow Java developers to store their data in any relational database, object database, or file-based storage without having to know any of the underlying details. This is actually very similar to the goals of the Object Data Management Group (ODMG) discussed in Chapter 9.

There has been much cross-pollination of ideas between the ODMG and the JDO design team. In fact, the ODMG's Java binding was submitted as a basis for JDO. They both provide transparent persistence, object querying, and extents. Even with these similarities, there are still many differences.

You may wonder why there is a need for yet another standard with JDBC for hand-built persistence and ODMG for transparent persistence. The primary reason given by Sun is a need for simplification. There are many parts of the ODMG specification that are not necessary for developers who want to focus exclusively on writing software in the Java language.

A second reason for another specification is because the ODMG has had some trouble getting consistent and complete implementations. Many products claim ODMG compliance while only providing a part of the specification. Creating the JDO specification allows Sun to define what should be available for a product to be compliant.

Because of the similarity between ODMG and JDO, many of the Java products that are ODMG compliant were first to become JDO compliant. This applies both to commercial and open source products.

The goal of this chapter is not to be a definitive guide to JDO. This chapter is intended to introduce JDO in a way that should help people to get up and

running in a basic way very fast. For in-depth coverage, I suggest getting *Using and Understanding Java Data Objects* by David Ezzio (Apress. ISBN: 1-59059-043-0), or even reading the specification from Sun, which can be found at `http://java.sun.com/products/jdo`.

What Is a JDO Implementation?

There are three conceptual parts to any JDO implementation, much like a JDBC implementation. There is Sun's JDO API, the classes that implement that API, and the specification that spells out how the implementation should act. The API is freely available from Sun, but has very little value without an implementation behind it.

Like they do with all of their Java APIs, Sun makes available enough interfaces and classes that developers can count on how they will use the implementation. Developers can download the JDO API and write software that uses JDO without any JDO implementation behind it; they will not be able to test it, but they can write it. One way that Sun made this easier is by moving all implementation-specific configuration information into a property file that is loaded at runtime.

So unlike in JDBC where you need to write the code that pulls the driver name from a configuration file and to either have a call to `Class.forName` or explicitly reference the driver class, in JDO you specify the driver class in the properties file, which is easily loaded and passed into JDO. This wonderful feature also allows code to easily move from one implementation to another, hopefully without any recompiling.

While it is true that Sun does not freely distribute a commercially usable JDO implementation, they do make available a Reference Implementation. The Reference Implementation is a way for Sun to prove that the specification is viable. The Reference Implementation is file based: Instead of using a relational database, it uses the file system for storage and the file system context for Java Naming and Directory Interface (JNDI). The code in this chapter was written using Sun's Reference Implementation.

While there are many interfaces and classes in the JDO API, the primary parts consist of `PersistenceManager`, `Query`, and `Transaction`. Most JDO work will be done with these interfaces. In the next section I will discuss how these classes relate.

Overview of the JDO Architecture

I said before that JDO consists primarily of a few important interfaces. Let's talk about what those are. Figure 10-1 shows the basic players in using JDO. The `JDOHelper` class is used to gain a reference to an implementing instance of the `PersistenceManagerFactory`. The persistence manager factory is used to obtain an instance of a `PersistenceManager`. It is from the persistence manager that you will

perform most of your work. Specifically, you will use the persistence manager to obtain references to Transactions, which you need for persisting and removing objects, and Queries that you use for finding objects.
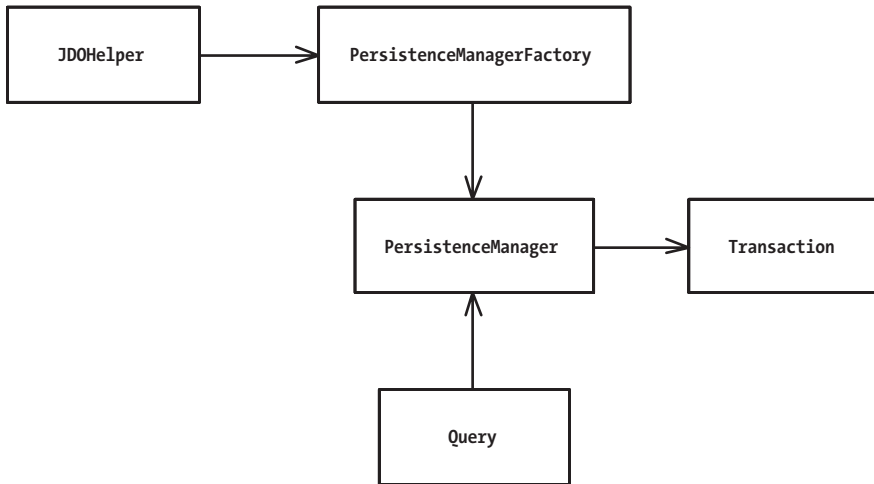

Figure 10-1. The main interfaces and classes of JDO

Of the objects shown, only the JDOHelper is an actual class. All others are interfaces. Each of those classes would be implemented by the underlying JDO implementation. It is for this reason that JDO helper is your starting point in working with JDO.

How JDO Works

In general, Sun has left the specifics of how a JDO implementation works to the JDO providers. Sun does not care how or where the implementers store their data. There are, however, commitments that any JDO implementation must fulfill.

Source and Byte-Code Enhancing

While a developer can create persistable classes for JDO by hand, I don't recommend it. There are many dependencies that would be easy to miss or introduce bugs into. While it may seem like a violation of Java to "enhance" either a source or class file, it is actually the best solution that could be arrived at.

The reason that a class needs to be enhanced is because Java does not provide enough hooks in the language to allow a persistence mechanism to know when work has been done to the persistable class. In other words, there is no way for a

JDO implementation to know that a value has been changed on an object retrieved from the data store.

In a handwritten persistence layer, it can be permissible to write data to the database that has not been modified or to load entire object graphs; these only work because the developers have complete control over the application and usage. Developers quickly notice a major performance problem introduced by loading the entire object graph. When writing a framework, you cannot control how developers will use it. For this reason you need to know exactly what is going on in the code.

In turn, the fine-grained control achieved through enhancement gives JDO the ability to provide complex functionality. It is possible in this way to provide lazy loading of data. In other words, it is possible to have a reference to an object that is not wholly loaded from the database.

So Sun decided on recommending class enhancement. There are two types of enhancement discussed in the specification: one is source code enhancement, the other is byte-code enhancement. In the first, the Java class's source code is modified before it is compiled. In the second, the post-compiled class file is modified.

The second method has a few benefits, such as the version-controlled source does not contain any non–developer-added code. Finally, a compiled then enhanced class will still contain valid debug information (provided one isn't trying to debug code the implementation added to the class). This is because the debug information is placed in the class file on the first compile and is not touched by the enhancement.

One aspect that the specification dictates is a binary compatibility; a class that is enhanced by the Reference Implementation must be usable by all compliant JDO implementations. Classes enhanced by any JDO-compliant enhancer must be usable by any other JDO-compliant implementation. This feature allows you to move enhanced JAR files and compiled classes from one implementation to another.

A class enhanced with one implementation's enhancer should work with another implementation. Since you can use any enhancer with any implementation, it is the implementation-specific features that separate the enhancers such as caching or anything else they use to differentiate themselves.

Be careful when using any implementation-specific features; this is the candy that gets organizations hooked on one specific vendor. Many times it seems prudent to use the features, but when the vendor raises its prices, changes its licensing fees, or goes out of business completely, it can be a very costly venture to refactor all your code to another implementation that does not have the same features.

JDO Transactions and What Happens After the Rollback

Transactions in JDO are similar to all the other transactions we have discussed in this book. They conform to the ACID properties. The JDO `Transaction` interface has `begin`, `rollback`, and `commit` methods. There isn't much mystery in how these work. Before modifying an object that is contained within persistent storage, the code needs to begin a transaction. When the work is done committing, the transaction will write the modified data to the data store.

Besides these expected functions, there are some additional helpful features of JDO transactions. One feature is the ability to tell the implementation to restore values to their original state after a commit or rollback. This gives developers the freedom to modify and work with objects knowing that if there is a problem, not only will the data store not be left in an instable state, but neither will the object being worked on.

One aspect of transactions in JDO that may not be immediately apparent is the fact that a persistence manager is tied to a single transaction at a time. This gives you the ability to easily access the current transaction from the persistence manager.

Querying with JDOQL

Stored objects are of little value without a way of querying, kind of like write-only memory. JDO has a very nice query mechanism. Essentially, querying in JDO relies on yet another query language, this one called JDOQL. Fortunately, it is a very simple language that is object based, similar to OQL.

Unlike OQL and SQL, JDOQL is a combination of objects, method calls, and filter strings. The entire query process is composed of creating a specific `Query` object, setting attributes of the query, and then executing the query and retrieving the results.

Queries are created from the persistence manager by making calls to the `newQuery` method. The idea of this method is to specify what class the query applies to. Optionally, in creating the query you can specify a list of objects that can be part of the results and/or indicate a filter that each part of the result must match.

Complex queries can contain parameters, variables, and filtering all specified through methods. Parameters can be specified to the query through calls to `declareParameters`. Variables are specified through `declareVariables`. Result ordering is specified through `setOrdering`.

The filtering language is very simple object and boolean based. This means that filtering is based on matching object attributes, either an object's attribute matches the filter or it does not. So to filter on employee information you create

the query with an `Employee` class. Then filter based on employees' attributes. The code for this can be as simple as that in Listing 10-1.

Listing 10-1. Employee Query Example

```
Query query = persistenceManager.newQuery(Employee.class, "firstName==\"John\"");
```

This query will return all employees whose first name is "John". The query itself will not be run until the `execute` method is called. At that time an unmodifiable collection will be returned containing all employees with the first name of John. By making the collection unmodifiable, Sun prevents confusion between the collection and the data store. Otherwise developers might believe they are adding objects to the data store if they could add the object to the collection.

Notice that the filter uses JavaBean property names very much like those used in OQL. You don't need to specify `getFirstName`, you simply use the attribute name "firstName". JDOQL also supports nested properties. You could use something like `address.zipCode.firstFive == "22333"` to find all employees in the 22333 zip code.

Getting All Instances in Extents

An extent is the way to obtain a reference to all instances of classes within the data store. The extent facility is different from a query in the fact that there is no ability to filter within the extent. The actual interface returned from the extent is that of a `java.util.Iterator`, which makes it very easy to run through the contents.

Often it is desirable to retrieve all instances of a class and its subclasses. An extent can optionally contain all instances of subclasses of the specified class. Whether to include subclasses is specified by the second argument, which is a boolean.

An extent can be extremely large. Once again, depending on the object graph, it could be prohibitively large to load all instances of a class. For this reason, many implementations support loading *hollow objects*. This refers to code that can have a reference to an object, but none of its data has been retrieved from the database. When data for the hollow object is accessed for the first time, it is loaded from the database.

JDO Class Restrictions and Requirements

There are very few restrictions placed on classes that can be persisted by JDO. Some of the restrictions are helped by JDO enhancers. Other restrictions you simply need to be aware of such as inheritance and attribute types.

For inheritance, persistence classes can inherit from nonpersistent classes. Nonpersistent classes can inherit from persistent classes. Persistent classes can inherit from persistent classes. However, persistent classes cannot inherit from natively implemented classes, e.g., `java.net.Socket` and `java.lang.Thread`.

That was a mouthful. Let's walk through that one more time. If you have a non-persistent class named `Person`, you can create a persistent class named `Employee` that extends it. You can create a second persistent class named `Manager` that extends `Employee`. You cannot, however, create a subclass of `java.lang.Thread` and persist it.

Any fields in a nonpersistent superclass cannot be persisted, which means that any attributes of nonpersistent class `Person` cannot be saved when you save an `Employee` object.

All classes in the hierarchy must use the same JDO identity type. So your `Employee` and `Manager` class must both use the same identity type. This is important for supporting extents.

Arrays are a concern when it comes to JDO. Arrays cannot be subclassed, so no specialized tracking class can be implemented. It is impossible to create a specialized array wrapper that would notice and track changes to an array. Therefore, a developer must use `JDOHelper`'s `makeDirty` method to indicate that an array's values have changed.

Different Types of JDO Identity

If two objects represent the same data in the JDO repository, they are said to share the same JDO identity. This is different from Java identity exposed through the "==" operator or the `equals` method. While JDO uniqueness is not dependent on these Java methods of comparison, if the same persistence manager is used, these tests can be true.

It is very important to maintain object uniqueness with JDO—this is how you can maintain data integrity. If two objects could have the same identity, they could end up having different state or data. The question would then arise as to which one would be considered the definitive source of data for the data store.

There are three types of identity in JDO: application, data store, and nondurable. *Application identity* is analogous to a primary key. It is maintained by the application and often required by the data store. A *data store identity* is maintained between JDO and the data store; the application does not know or care about it. *Nondurable identity* is intended for an implementation to track objects within the JVM.

JDO XML Metadata

One of the requirements of the JDO specification is that a persistable object must have a metadata file associated with it. JDO's metadata files are XML files with a .jdo extension. The purpose of the file is to tell the enhancer and implementation what objects to persist and how.

If only one class is defined in the metadata file, then the filename should be the same as the class but with the .jdo extension. If multiple classes are defined in the file, then the metadata file should have the name of the shared package. A metadata file for `com.apress.javapersist.bo.Employee` could be named Employee.jdo, bo.jdo, javapersist.jdo, apress.jdo, or com.jdo. The enhancer will look for the metadata file in that order.

The contents of the metadata files are very simple, containing only a few tags. The structure of the file is as follows:

- *XML declaration:* Defines the file as an XML file.

- *DTD:* Indicates the Document Type Definition, which describes how the document needs to be put together.

- *jdo tag:* Starts the definition of the metadata. Can only contain one or more package tags and any number of vendor extension tags.

- *package tag:* Specifies a wrapper for the classes you will define. Can contain one or more class tags and any number of vendor extension tags.

- *class tag:* Defines a class that can be persisted. Can contain any number of field and vendor extension tags.

- *field tag:* Defines a field that can be persisted or ignored. Can contain one collection, map, or array and any number of vendor extensions.

- *collection tag:* Describes the collection. Can contain any number of vendor extension tags.

- *map tag:* Describes a map. Can contain any number of vendor extension tags.

- *array tag:* Describes an array. Can contain any number of vendor extension tags.

- *extension tag:* Used for nonstandard JDO functionality or to convey information to the underlying implementation. Basically it contains the vendor name and a key-value pair.

Listing 10-2 contains a simple example of a metadata file. The file is for an Employee class. Notice that only one attribute is listed; all other attributes are by default persisted if they are not Java transient, static, or final.

Listing 10-2. Employee.jdo

```xml
<?xml version="1.0"?>
<!DOCTYPE jdo SYSTEM "jdo.dtd">
<jdo>
    <package name="com.apress.javapersist.chapter10.bo">
        <class name="Employee">
          <field name="age" persistence-modifier="none"></field>
        </class>
    </package>
</jdo>
```

How You Can Use JDO to Persist Your Objects

For this next section, I will break down a basic JDO application. I will show you the various parts of the application, explaining them one by one. I have broken the application down into separate commands. I tried to create one command for each piece of JDO functionality I will explain.

What I will show you how to create in this section is a simple HR application, building on the work done in earlier chapters. Essentially, you will see how to create a command line tool that will store employees and addresses. It can list stored objects and remove stored objects. It will allow you to associate addresses with employees. Finally, I will show some examples of using the application.

Example Business Objects

The Employee class for this example is a simple Java business object as seen in Listing 10-3. The class has a few attributes, but nothing JDO specific. You need to add a no-argument constructor as I have done here to support the reference implementation's enhancer. Some commercial enhancers add this if it is not present. Keep in mind though that in the final class there will be a public no-args constructor whether you added it or the enhancer did.

Listing 10-3. Employee.java

```
package com.apress.javapersist.chapter10.bo;

import java.util.*;

public class Employee {
    private long id;
    private String firstName;
    private String lastName;
    private String email;
    private Address address;

    public Employee() {
    }

    public Employee(long id) {
        this.id = id;
    }

    public long getId() {
        return id;
    }

    public String getFirstName() {
        return this.firstName;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public String getLastName() {
        return this.lastName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }

    public String getEmail() {
        return this.email;
    }
```

```java
    public void setEmail(String email) {
        this.email = email;
    }

    public Address getAddressea() {
        return address;
    }

    public void setAddress(Address address) {
        this.address = address;
    }

    public String toString() {
        return "id: " + id + "; "
            + "firstName: " + firstName + "; "
            + "lastName: " + lastName + "; "
            + "email: " + email + "; "
            + "address: [" + address + "]";
    }
}
```

You will also need to provide a metadata file for the employee object. The file in Listing 10-4 is a very simple metadata file. There are no special fields and no implementation specific enhancements. This file will be read by your enhancer and needs to be deployed with your class. Notice that even the application-specific `Address` attribute is not specified; because it is persistable, it will be taken care of by the implementation.

Listing 10-4. Employee.jdo

```xml
<?xml version="1.0"?>
<!DOCTYPE jdo SYSTEM "jdo.dtd">
<jdo>
    <package name="com.apress.javapersist.chapter10.bo">
        <class name="Employee">
        </class>
    </package>
</jdo>
```

The `Address` class in Listing 10-5 for your employees is another simple business object. The one interesting feature of this class is the `ArrayList` of residents. This class actually has a collection-based attribute that you can use for querying.

Listing 10-5. Address.java

```java
package com.apress.javapersist.chapter10.bo;

import java.util.*;

public class Address {
    private long id;
    private String streetLine1;
    private String streetLine2;
    private String city;
    private String state;
    private ArrayList residents = new ArrayList();

    public Address() {}

    public Address(long id) {
        this.id = id;
    }

    public long getId() {
        return id;
    }

    public String getCity() {
        return city;
    }

    public void setCity(String city) {
        this.city = city;
    }

    public String getState() {
        return state;
    }

    public void setState(String state) {
        this.state = state;
    }
```

```
    public String getStreetLine1() {
        return streetLine1;
    }

    public void setStreetLine1(String streetLine1) {
        this.streetLine1 = streetLine1;
    }

    public String getStreetLine2() {
        return streetLine2;
    }

    public void setStreetLine2(String streetLine2) {
        this.streetLine2 = streetLine2;
    }

    public Collection getResidents() {
        return residents;
    }

    public void addResident(Employee resident) {
        this.residents.add(resident);
    }

    public String toString() {
        return  "id: " + id + "; "
            + "line 1: " + streetLine1 + "; "
            + "line 2: " + streetLine2 + "; "
            + "city: " + city + "; "
            + "state: " + state + "; "
            + "residents: " + residents.size();
    }
}
```

The metadata file for the Address class in Listing 10-6 is slightly more interesting than the employee metadata file. This file contains a field that is a collection. Notice that you need to specify the type of object that will be in the collection.
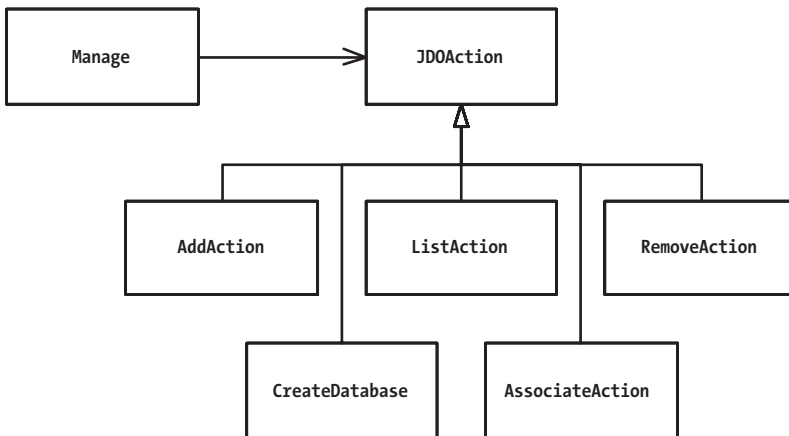
Listing 10-6. Address.jdo

```xml
<?xml version="1.0"?>
<!DOCTYPE jdo SYSTEM "jdo.dtd">
<jdo>
    <package name="com.apress.javapersist.chapter10.bo">
        <class name="Address">
            <field name="residents">
              <collection element-type="Employee"/>
            </field>
        </class>
    </package>
</jdo>
```

Performing the JDO Work

In this next section, you will see application code that will work on your business objects. It is in these classes that you will add, list, remove, and modify objects. Figure 10-2 shows how the various commands relate.

Figure 10-2. Human resources application main classes

All of your JDO commands will need certain functionality that can be shared—for instance, obtaining a reference to a persistence manager, or finding objects to work on. In Listing 10-7 you can see a JDOAction class that contains that functionality. The getPersistenceManager method is used by all subclasses when a persistence manager is needed.

Listing 10-7. JDO Action Is the Superclass of All Your JDO Commands

```java
package com.apress.javapersist.chapter10;

import java.io.IOException;
import java.io.InputStream;
import java.util.Collection;
import java.util.Iterator;
import java.util.Properties;

import javax.jdo.JDOHelper;
import javax.jdo.PersistenceManager;
import javax.jdo.PersistenceManagerFactory;
import javax.jdo.Query;

/**
 * @author rsperko
 *
 * This class is the super class of all our JDO commands. This class provides
 * utility methods for all our JDO work. Specifically this is where we would
 * obtain our connection to our persistence manager.
 */
public abstract class JDOAction {
    private static PersistenceManagerFactory factory = null;

    /**
     * Method getPersistenceManager returns a persistence manager object from
     * the class's static persistence manager factory object.
     */
    protected PersistenceManager getPersistenceManager() {
        if (factory == null) {
            Properties props = loadProperties();

            factory = JDOHelper.getPersistenceManagerFactory(props);
        }
        return factory.getPersistenceManager();
    }

    /**
     * Method loadProperties.
     * @return Properties
     */
    protected Properties loadProperties() {
        Properties props = new Properties();
```

```
        try {
            InputStream in =
                ClassLoader.getSystemResourceAsStream ("jdo.properties");
            props.load (in);
        }
        catch(IOException ioe) {
            ioe.printStackTrace();
        }
        return props;
    }

    /**
     * Method findAll.
     * @param className
     * @param string
     * @return Collection
     */
    protected Iterator findAll(PersistenceManager pm, String className,
        String queryStr) {
        try {
            if("extent".equals(queryStr)) {
                return pm.getExtent(getClass(className), false).iterator();
            }
            else {
                Query query = pm.newQuery(getClass(className), queryStr);
                return ((Collection) query.execute()).iterator();
            }
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }
        return null;
    }

    /**
     * Method getClass.
     * @param className
     */
    private static Class getClass(String className)
        throws ClassNotFoundException {
        return Class.forName("com.apress.javapersist.chapter10.bo."
            + className);
    }

    /**
```

```
    * Method all of our commands will override to do their work.
    */
   public abstract void execute(String typeName, String[] optional);

   /**
    * used by the help message to convey how this action will be used.
    */
   public abstract String exampleString();
}
```

Particularly note the getPersistenceManager method. This is the method that all of your subclasses will use to work the JDO implementation. This method loads the properties file that defines what JDO implementation to use and its configuration data.

The other methods in this class are helper methods. To keep the examples simpler, the findAll method that will be shared by the list, associate, and remove commands is in this class. Also, there is a helper method for turning a class base name into an actual class.

To get a better understanding of how the persistence manager is obtained, Figure 10-3 shows a sequence diagram of the relevant objects. Notice that there are two classes involved, the Properties object and the JDOHelper class. The persistence manager factory is an interface obtained from the JDO helper.
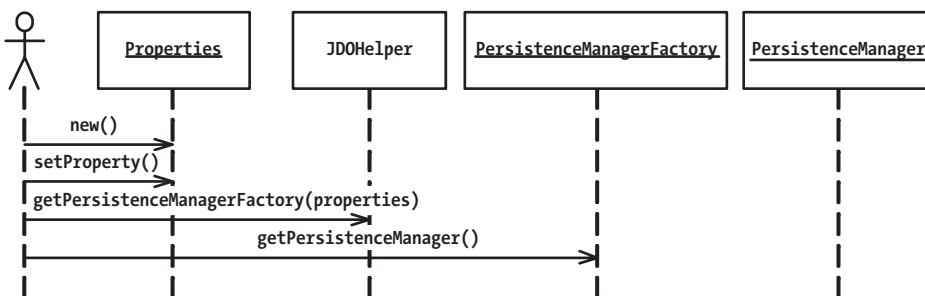

Figure 10-3. Obtaining a persistence manager

Now that you have a superclass where you can obtain your persistence manager, let's look at the class that can add an object to the JDO repository. Listing 10-8 shows the AddAction class, which does most of its work in the execute method. It is here that a transaction is started, an object is persisted, and the transaction is committed.

Listing 10-8. AddAction.java

```java
package com.apress.javapersist.chapter10;

import javax.jdo.PersistenceManager;

import com.apress.javapersist.chapter10.bo.Address;
import com.apress.javapersist.chapter10.bo.Employee;

public class AddAction extends JDOAction {

    /**
     * Add the object
     * @see com.apress.javapersist.chapter10.JDOAction#execute(String, String[])
     */
    public void execute(String typeName, String[] optional) {
        Object obj = buildObject(typeName, optional);
        PersistenceManager pm = getPersistenceManager();

        pm.currentTransaction().begin();
        // Mark the object as persistent
        pm.makePersistent(obj);
        pm.currentTransaction().commit();
        System.out.println("Object stored: " + obj);
        pm.close();
    }

    /**
     * Method buildObject.
     * @param className
     * @param optional
     * @return Object
     */
    private static Object buildObject(String className, String[] optional) {
        Object obj = null;
        if("Employee".equals(className)) {
            Employee emp = new Employee(Long.parseLong(optional[0]));
            emp.setFirstName(optional[1]);
            emp.setLastName(optional[2]);
            emp.setEmail(optional[3]);
            obj = emp;
        }
        else if("Address".equals(className)){
            Address add = new Address(Long.parseLong(optional[0]));
```

```
        add.setStreetLine1(optional[1]);
        add.setStreetLine2(optional[2]);
        add.setCity(optional[3]);
        add.setState(optional[4]);
        obj = add;
    }
    return obj;
}


/**
 * @see com.apress.javapersist.chapter10.JDOAction#exampleString()
 */
public String exampleString() {
    return "   Manage add Employee id FName LName eMail\n"
        + "   Manage add Address id line1 line2 city state";
}
}
```

Figure 10-4 illustrates exactly how an object is persisted. Notice how simple it is to add an object to a JDO data store. All it takes is a call to makePersistent wrapped in a transaction. If the object is enhanced, it is added and ready to be managed by JDO.
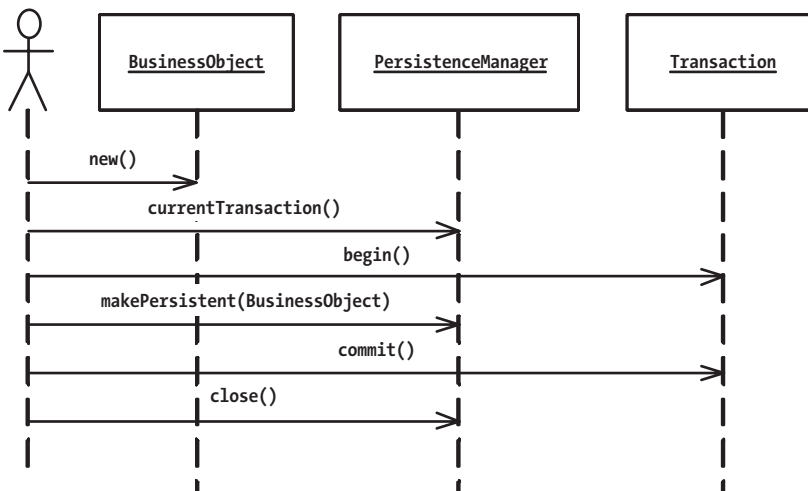

Figure 10-4. Persisting an object

There is an additional helper method in this class for building actual objects that will be added. This method is fairly fragile in that it expects a specific number

of arguments in a specific order. It would be a fairly simple exercise to make that code more robust.

Next you want to see the objects you have stored in your repository. The ListAction class in Listing 10-9 is very light because most of your querying functionality is in your superclass to help with other actions. If you look back at the JDOAction class, you see that the findAll method simply calls getExtent or creates a query and executes it. The ListAction then prints out the contents that are returned.

Listing 10-9. ListAction.java

```java
package com.apress.javapersist.chapter10;

import java.util.Iterator;
import javax.jdo.PersistenceManager;

public class ListAction extends JDOAction {
    private Object className;

    /**
     * List the objects that match the query
     * @see com.apress.javapersist.chapter10.JDOAction#execute(String, String[])
     */
    public void execute(String typeName, String[] optional) {
        PersistenceManager pm = getPersistenceManager();
        Iterator iter = findAll(pm, typeName, optional[0]);

        // Run through the objects listing each one
        while( iter.hasNext() ) {
            System.out.println(iter.next());
        }
    }

    /**
     * Show how to use this action
     * @see com.apress.javapersist.chapter10.JDOAction#exampleString()
     */
    public String exampleString() {
        return "    Manage list Employee \"firstName == \\\"LName\\\"\n"
            + "    Manage list Employee extent";
    }
}
```

In Figure 10-5, you can see the how the classes interact to perform the query itself. A `Query` object is created from the persistence manager. The query is then executed and the resulting objects are available to be worked on.
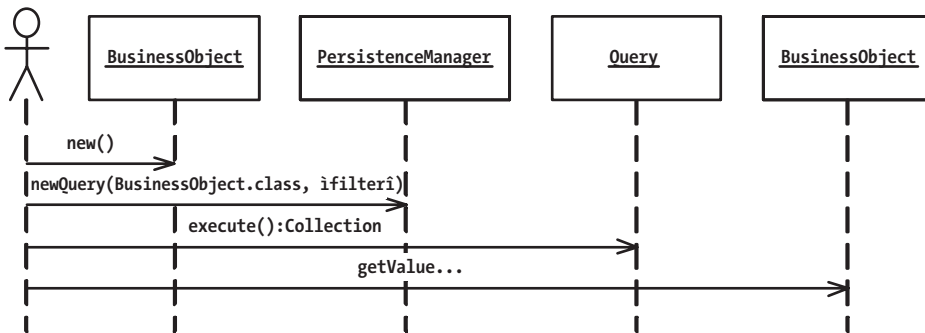

Figure 10-5. Querying for objects

Obviously a management tool should allow removal of employees from the HR application. Listing 10-10 contains the `RemoveAction` class. This class queries the repository using the `findAll` method in the `JDOAction`. It then iterates through the results, removing each from the data store. Removing from the data store is as simple as calling `deletePersistent` on an object within a transaction.

Listing 10-10. RemoveAction.java

```java
package com.apress.javapersist.chapter10;

import java.util.Iterator;
import javax.jdo.PersistenceManager;

/**
 * @author rsperko
 *
 * Remove any objects that match the criteria
 */
public class RemoveAction extends JDOAction {

    /**
     * Remove all instances that match the query
     * @see com.apress.javapersist.chapter10.JDOAction#execute(String, String[])
     */
    public void execute(String typeName, String[] optional) {
        PersistenceManager pm = getPersistenceManager();
        Iterator results = findAll(pm, typeName, optional[0]);
```

```
        pm.currentTransaction().begin();
        // Run through the list of objects deleting each one
        while(results.hasNext())
          pm.deletePersistent(results.next());
        pm.currentTransaction().commit();
    }

    /**
     * Show how to use this action
     * @see com.apress.javapersist.chapter10.JDOAction#exampleString()
     */
    public String exampleString() {
        return "    Manage remove Employee \"lastName == \\\"LName\\\"";
    }
}
```

Finally, I want to illustrate modifying a persistent object. The `AssociateAction` class, shown in Listing 10-11, will modify both an employee and an address by associating one object with the other. For this to work, the class will query the data store for the objects you want to associate. The `execute` method will then create a transaction, associate the objects, and commit the transaction.

Listing 10-11. AssociateAction.java

```
package com.apress.javapersist.chapter10;

import javax.jdo.PersistenceManager;

import com.apress.javapersist.chapter10.bo.Address;
import com.apress.javapersist.chapter10.bo.Employee;

/**
 * @author rsperko
 *
 * This class is used to associate one object with another.
 */
public class AssociateAction extends JDOAction {
    /**
     * Associate one object with another
     * @see com.apress.javapersist.chapter10.JDOAction#execute(String, String[])
     */
    public void execute(String typeName, String[] optional) {
        if("Employee".equals(typeName)) {
```

```
        PersistenceManager pm = getPersistenceManager();
        Employee employee = (Employee) findAll(pm, typeName,
            optional[0]).next();
        Address address = (Address) findAll(pm, optional[1],
            optional[2]).next();
        System.out.println("About to associate: " + employee + " with "
            + address);

        pm.currentTransaction().begin();
        // Actually associate the two classes
        employee.setAddress(address);
        address.addResident(employee);
        pm.currentTransaction().commit();
    }
}

/**
 * Show how to use this action
 * @see com.apress.javapersist.chapter10.JDOAction#exampleString()
 */
public String exampleString() {
    return "    Manage assoc Employee \"id==0\" Address \"id==1\"";
}
}
```

The beauty of JDO here is that no work needs to be done other than committing the transaction. If there were an error, the transaction could roll back and everything would return to its original state.

Using the Sun JDO Reference Implementation

There are a couple aspects of the example management application that are specific to Sun's reference implementation. The first is an optional method that creates the database. This same class could be extended to create databases or initialize databases for other implementations.

In this case, I will show you how to create the database file. You can see in Listing 10-12 that this is done by simply adding another property before obtaining the persistence manager factory. The next step is beginning and committing a transaction. That is all there is to it.

Listing 10-12. CreateDatabaseAction.java

```java
package com.apress.javapersist.chapter10;

import java.util.Properties;

import javax.jdo.JDOHelper;
import javax.jdo.PersistenceManager;
import javax.jdo.PersistenceManagerFactory;
import javax.jdo.Transaction;

public class CreateDatabaseAction extends JDOAction {
    /**
     * Method createDatabase will instantiate a new database if needed.
     * @param typeName
     */
    public void execute(String typeName, String[] optional) {
        if("jdori".equals(typeName))
            createJDORIDatabase();
        else
            System.out.println("ERROR: I don't know how to create a "
                + typeName + " database");
    }


    /**
     * Method createDatabase.
     */
    private void createJDORIDatabase() {
        try {
            Properties props = loadProperties();
            props.put("com.sun.jdori.option.ConnectionCreate", "true");
            PersistenceManagerFactory pmf =
                JDOHelper.getPersistenceManagerFactory(props);
            PersistenceManager pm = pmf.getPersistenceManager();
            Transaction tx = pm.currentTransaction();
            tx.begin();
            tx.commit();
            pm.close();
        } catch(Exception e) {
            System.err.println("Exception creating the database");
            System.err.println(e);
            System.exit(-1);
        }
    }
```

```
    /**
     * Show how to use this action
     * @see com.apress.javapersist.chapter10.JDOAction#exampleString()
     */
    public String exampleString() {
        return "    Manage create jdori";
    }
}
```

The last Sun Reference Implementation–specific part is the jdo.properties file that tells the application that it will use that JDO implementation, and this is shown in Listing 10-13. To change JDO implementations, you could simply replace this file and away you go. As I mentioned before, replacing or modifying a property file is a lot nicer than having to recompile the code when making these changes.

Listing 10-13. jdo.properties

```
javax.jdo.PersistenceManagerFactoryClass=com.sun.jdori.fostore.FOStorePMF
javax.jdo.option.ConnectionURL=fostore:jdoriDB
javax.jdo.option.ConnectionUserName=sa
javax.jdo.option.ConnectionPassword=
```

Last but not least is the class that will tie it all together. Listing 10-14 contains the actual `Manage` class that handles the command line and calls the various `JDOAction` subclasses. Looking at the constructor, you see your actions created and added to a `HashMap`. The `executeAction` runs the actions with arguments.

The `usage` method first displays a list of all the actions in the `HashMap`. Next it asks each method to show an example of how it can be used. All of the work gets delegated to the actions themselves.

Listing 10-14. Manage.java

```
package com.apress.javapersist.chapter10;

import java.util.HashMap;
import java.util.Iterator;

/**
 * @author rsperko
 *
 * This class is used to interact with any Java Data Objects implementation.
 * In order to change the JDO implementation that is used, modify the
 * "jdo.properties" file.
```

```
    *
    * Call this class with no arguments to find out how to use it.
    */
public class Manage {
    HashMap actions = new HashMap();

    public Manage() {
        actions.put("create", new CreateDatabaseAction());
        actions.put("list", new ListAction());
        actions.put("add", new AddAction());
        actions.put("remove", new RemoveAction());
        actions.put("assoc", new AssociateAction());
    }

    private void executeAction(String[] args) {
        if(args.length < 2) {
            usage();
            return;
        }
        String actionStr = args[0];
        String typeName = args[1];
        String[] optional = new String[args.length-2];
        System.arraycopy(args, 2, optional, 0, optional.length);

        JDOAction action = (JDOAction) actions.get(actionStr);
        if(action == null) {
            usage();
            return;
        }

        action.execute(typeName, optional);
    }

    public static void main(String[] args) {
        Manage manage = new Manage();
        manage.executeAction(args);
    }


    /**
     * Method usage.
     */
    private void usage() {
        System.out.println("usage:");
```

```
        System.out.println("java com.apress.javapersist.chapter10.Manage <act"
            + "ion> <object type> [arg1] [arg2] [argn]");
        System.out.println("    actions: " + actions.keySet());
        System.out.println();
        System.out.println("Examples:");
        for(Iterator iter = actions.values().iterator(); iter.hasNext(); ) {
            System.out.println(((JDOAction) iter.next()).exampleString());
        }
    }
}
```

Summary

Java Data Objects is a Java API from Sun that is intended to allow write once, store anywhere. Sun went to great lengths to make the standard easy to use and as transparent as possible to developers. Along with the specification, Sun also released a reference implementation to illustrate how the specification should be implemented.

JDO consists of an API and an implementation. The primary classes in the implementation are JDOHelper, PersistenceManagerFactory, PersistenceManager, Query, and Transaction. Most work will be done in JDO with these classes.

In order to make the framework a success, Sun has proposed adding class or source enhancing to give the ability to closely manage persistent classes. Enhancing consists of running a tool against a source file or a compiled class file. The Java code is modified to allow fine-grained control over how the object is used and what happens when values are modified.

Sun has also specified that a class that is enhanced with one enhancer should work with other JDO implementations. This allows some implementations to completely forgo creating their own enhancers and relying on the enhancer provided in the reference implementation.

Querying in JDO is done using JDOQL, which is a combination of classes and strings. It is an object-oriented way to query for objects in a data store. The general way it works is a class is provided to the query, then filters are applied that limit what results come back from the query. JDOQL gives a very rich object-oriented method of querying.

Transactions in JDO are similar to other transactions we have discussed in this book. They do follow the ACID properties. Further transactions are managed through the PersistenceManager class.