Using and Understanding Java Data Objects

DAVID EZZIO

Apress™

Using and Understanding Java Data Objects

ISBN (pbk): 1-59059-043-0

Printed and bound in the United States of America 12345678910

Technical Reviewers: Regis Le Brettevillois, John Mitchell, Abe White

Editorial Directors: Dan Appleman, Gary Cornell, Martin Streicher, Karen Watterson, John Zukowski

Assistant Publisher: Grace Wong

Project Manager: Tracy Brown Collins

Copy Editor: Ami Knox

Production Editor: Julianna Scott Fein

Composition: Susan Glinert

Indexer: Valerie Robbins

Proofreader: Elizabeth Berry

Cover Designer: Kurt Krames

Production Manager: Kari Brooks

Manufacturing Manager: Tom Debolski

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

CHAPTER 8

Using JDO to Learn More

The JDO Learning Tools are a small but comprehensive set of programs that you can use for several purposes. You can use them as an introduction to JDO. You can use them to verify what you have learned about JDO. Most importantly, you can use them to learn more. The first five JDO Learning Tools are discussed in this chapter.

- `TestJDOHelper`
- `TestFactory`
- `MegaCups`
- `Library`
- `StateTracker`

The five learning programs are not examples of how your application might use JDO. Instead, they are atypical applications whose purpose is to illuminate the interactions with JDO. `TestJDOHelper` and `TestFactory` allow you to start using a JDO implementation and find out more about its capabilities. The `MegaCups` program demonstrates JDO's ability to handle multiple, concurrent updates. The `Library` program allows you to interactively populate a small town's library and run your own queries against its objects. Using it, you can test JDO's query language. The `StateTracker` program allows you to manipulate and view the managed and unmanaged state of persistent, transactional, and unmanaged apples. You can use it to execute all of JDO's explicit operations and many of its implicit operations, and you can see the consequences on managed objects.

In addition to the five learning programs, there are two sets of example programs, the rental application and the quote server, that are discussed in Chapters 9 through 11. The rental applications and the quote servers provide examples of how your application might use JDO. Between them, they cover many application architectures. The rental application is a prototype reservation system for a fictitious lighthouse rental company. It comes in three versions: the rental Swing application, the rental Web application, and the rental enterprise (EJB) application. Each version implements nearly the same set of requirements. Each chapter from 9 to 11 takes one version of the rental application as its main topic. The quote server application stores new quotes given to it and serves up a random quote upon demand. It is implemented in five types of Enterprise JavaBeans: a stateless CMT session bean,

a stateful CMT session bean, a stateless BMT session bean, a stateful BMT session bean, and a BMP entity bean. The quote server examples are discussed in Chapter 6 as well as Chapter 11.

The JDO Learning Tools 1.0 are copyright by Yankee Software and are provided to the community of Java programmers under the open source GNU General Public License. Instructions for downloading the JDO Learning Tools are found in the section "Step One: Download Open Source JDO Learning Tools" that follows in this chapter. You are encouraged to contribute improvements to make future releases of the JDO Learning Tools better for all of us.

The Ant Build Scripts

All of the programs provided with this book are built with Ant, a build tool from Apache's Ant project. Build scripts are provided for four different implementations:

- The reference implementation from Sun Microsystems
- The Kodo implementation from SolarMetric
- The Lido implementation from Libelis
- The IntelliBO implementation from SignSoft

A new script could be added for any other JDO implementation as long as the implementation supplies command line tools that can be used with Ant. The scripts were tested with version 1.4.1 of Ant, and they should work with any later 1.x version of Ant. It is strongly recommended that you use Ant to build the JDO Learning Tools.

Ant's build scripts are XML files. The major build scripts are in the *bookants* directory. The build scripts that compile the individual programs are in each project directory. The main build script, called *build.xml*, reads three property files in the *bookants* directory to set its configuration. The *global.properties* file contains properties that you should not have to change. The *custom.properties* file is tailored to the particular JDO tool that you are using. You will want to modify the appropriate property file for the tool, such as *jdori.properties*, and then make a copy of it that is named *custom.properties*. The *default.properties* file sets properties that must be customized to your installation.

The main Ant build script reads the property files in the following order:

1. global.properties
2. custom.properties
3. default.properties

Because all Ant properties are final, if a property is set in more than one property file, the value that sticks is the first value encountered. As a result, if the property *jdo.tool* is encountered in the *custom.properties* file as well as in the *default.properties* file, the value in the *custom.properties* file is the one used.

In order to work with a variety of JDO implementations, the build scripts are a bit more complicated than they would be in a typical development environment. The main *build.xml* script calls out to one of several possible tool scripts that are individually configured to the specific JDO implementation. There are four supplied with the code: *jdori.xml*, *kodo.xml*, *lido.xml*, and *intellibo.xml*. Only one tool script is used in any build environment. The tool script selection is controlled by the *jdo.tool* property in the *custom.properties* file. To minimize needless redundancy, the tool scripts call out to a common set of third-level scripts. Each package has its own compile script to compile the files that are specific to it. These are contained in *build.xml* scripts that are in the subdirectories of the source. Likewise, each container has a script for compiling and deploying the files that are specific to its needs. These are the *tomcat.xml* and *jboss.xml* scripts contained within the *bookants* directory.

As a result, when you type *ant target* in the *bookants* directory, the *target* is invoked in the following build scripts: in the *build.xml* script in the *bookants* directory; in the specific tool script, such as *jdori.xml*, in the *bookants* directory; in the *build.xml* that is in the target's source directory; and finally, if the build deploys on Tomcat or JBoss, then in the *tomcat.xml* or *jboss.xml* script in the *bookants* directory.

Many of the build targets create batch files for Microsoft Windows that run the programs. The current build scripts do not produce command files for other operating systems. As a result, if you are running on Linux or some other operating system, you will have to change the generated batch files so that they can be invoked on your operating system.

Getting Started

This section provides a step-by-step guide to getting the code and building the first five JDO Learning Tools.

Step One: Download Open Source JDO Learning Tools

Download the zip file containing the JDO Learning Tools from SourceForge.net by going to the following URL:

```
http://sourceforge.net/projects/jdo-tools
```

You can also obtain this code from the Apress Web site at `http://www.apress.com`. After the download, unzip the files to a directory of your choice. For simplicity, the instructions here will refer to the directory that you choose as the *bookcode-home* directory. Unless otherwise specified, all directory paths mentioned in this chapter are relative to your *bookcode-home* directory.

Estimated time excluding download: 5 minutes

Step Two: Download Java SDK If Necessary

The examples in this book were tested with the Java Software Development Kit (SDK, also known as JDK) version 1.3.1. The various implementations and the examples will likely work with a later version of the JDK, but this has not been tested. If you do not already have JDK 1.3.1, go to the following URL:

```
http://java.sun.com/j2se/1.3
```

Follow the instructions that come with the JDK to set up the Java development and runtime environments and verify that they are working.

Estimated time excluding the download: 20 minutes

Step Three: Download Ant If Necessary

If you do not already have Ant version 1.4.1, or a later 1.x version, go to the following URL:

```
http://ant.apache.org
```

Follow the links to download a binary version of Ant. Follow Ant's instructions to set up Ant and verify that it is working.

Estimated time excluding download: 45 minutes

Step Four: Download J2EE Jar If Necessary

You need access to the *j2ee.jar* file that contains the public interfaces of J2EE. This file, which can be found in the J2EE SDK, is available at the following Web location:

```
http://java.sun.com/j2ee/download.html
```

Download it, and follow the directions to install it. This file is needed for all build targets because some of the common files use the J2EE framework to report messages.

Estimated time excluding download: 10 minutes

Step Five: Download JDO Implementation

The next step is to pick the JDO implementation that you want to use. The build scripts in this book work with any of the four JDO implementations mentioned earlier. The examples should work with any implementation, but you will have to create or find the tool script for implementations that are not on this list. It is possible that some vendors who are not on the list will provide a tool script for their implementation. Building a new tool script is not difficult, but you may want to start your exploration of JDO with the reference implementation. The reference implementation does have one serious drawback: the 1.0 version does not work well with EJB containers. As a result, the EJB examples in this book do not have build targets for the reference implementation.

To get the reference implementation, go to the Java Community Process page at the following URL:

```
http://jcp.org/aboutJava/communityprocess/final/jsr012/index.html
```

or go to the public access page maintained by the specification lead at the following URL:

```
http://access1.sun.com/jdo
```

Download the reference implementation and unzip to a directory of your choice.

To download one of the commercial implementations, go to the associated vendor's Web site and follow the directions. The home page for each is listed here.

```
http://www.solarmetric.com
http://www.libelis.com
http://www.signsoft.com
```

Estimated time for the reference implementation excluding download: 5 minutes. It will take longer for the commercial implementations because of the need to set up a license key and configure the JDBC settings.

Step Six: Configure Build Properties

In this step, you configure the property files. The properties in the *global.properties* file, found in the *bookants* directory, should not require changes, unless there are some operating system issues that need to be addressed.

The four tool property files provided are found in the *bookants* directory. Edit the property file that goes with the implementation that you have selected. For the reference implementation, that would be the *jdori.properties* file. This file has two properties. The *jdo.home* property should be set to the root directory where the implementation is installed. The second property, *jdo.tool*, is the name used to find the tool's build script. It should not be changed, unless you are writing a new tool script. After configuring the tool's properties file, make a copy named *custom.properties* in the *bookants* directory.

Next, edit the *default.properties* file. This file contains the following properties:

- *java.home*
- *jdbc.jar*
- *j2ee.home*
- *tomcat.home*
- *jboss.home*

Of these, the last two will be addressed in Chapters 10 and 11. The *jdbc.jar* property is required only if a relational implementation of JDO is used. These include the Kodo, Lido, and IntelliBO implementations. The *java.home* property provides the file path to the root directory where you installed the Java SDK. Set the *j2ee.home* property to point to the root directory where you installed the J2EE SDK.

Estimated time for this step: 10 minutes

Step Seven: Test the Build Environment

You are now ready to test the build environment. To begin, go to the *bookants* directory and type *ant*. You should see console output similar to the output in the following lines:

```
E:\Bookcode\bookants>ant
Buildfile: build.xml

Help:
     [echo] Please specify a particular build target, such as testfactory
     [echo]    or, enter the command:   ant -projecthelp
     [echo]    for a list of targets

BUILD SUCCESSFUL
```

By typing *ant -projecthelp*, you will see a list of targets for the build. The expected output will look similar to the output in Listing 8-1. You will see quite a few more main targets than the ones listed here. The additional targets that are not shown in Listing 8-1 are used in later chapters.

Listing 8-1. Expected Output from Running ant -projecthelp at the Command Line

```
E:\Bookcode\bookants>ant -projecthelp
Buildfile: build.xml
Default target:
 Help                   The default target for this build script

Main targets:
 Help                   The default target for this build script
 clean-out              removes all files in output directories
 megacups               build MegaCups example
 testfactory            build TestFactory example
 testjdohelper          build TestJDOHelper example
 learning-programs      Builds all learning-programs
 library                build Library example
 statetracker           build Statetracker example

Subtargets:
 are-we-ready
 help
 verify

BUILD SUCCESSFUL
```

If everything is working as expected at this point, then you are ready to try the most dangerous target in the build, the *clean-out* target. This target deletes all files in the *build* directory, *enhanced* directory, and *warfiles* directory under the *bookcode-home* directory. The expected output, since you have not yet built any files to delete, will look like the following:

```
E:\Bookcode\bookants>ant clean-out
Buildfile: build.xml
are-we-ready:
verify:
clean-out:
     [echo] Deleting files in build, enhanced, and warfiles

BUILD SUCCESSFUL
```

Use the *clean-out* target whenever you want the subsequent build to proceed from scratch.

The *testjdohelper*, *testfactory*, *megacups*, *library*, and *statetracker* targets are described in the following sections of this chapter. They can be built individually, or you can build them all at once by going to the *bookants* directory and typing *ant learning-programs*. The build should take a minute or so.

Estimated time for this step: 10 minutes

Hello, JDO!

The `TestJDOHelper` program is simple. Listing 6-2 in Chapter 6 shows the one class, `TestJDOHelper`, found in the `com.ysoft.jdo.book.factory` package.

The program takes one command line parameter that names the properties file configured for the selected JDO implementation. The main method of the class loads the properties from the file and then calls the `getPersistenceManagerFactory` method in `JDOHelper`. For more information about getting a persistence manager factory from `JDOHelper`, see Chapter 6. After verifying that a persistence manager can be obtained from the persistence manager factory, the program ends.

The properties file that specifies the factory settings varies with the implementation used. For the reference implementation, the properties file is *jdori.properties*, located at *com/ysoft/jdo/book/factory/jdori*. The build copies it to *factory.properties* at *build/com/ysoft/jdo/book/factory*. If you are using one of the three mentioned commercial implementations, you will have to edit the appropriate file to configure the JDBC connection. When using the Kodo implementation, you must also add a valid license key to the properties file. The reference implementation does not use JDBC and does not require a license key.

Building and Running the TestJDOHelper Program

To build the `TestJDOHelper` program, go to the *bookants* directory and type *ant testjdohelper*. This build is part of the *ant learning-programs* build. Listing 8-2 shows some of the expected output when using the reference implementation.

Listing 8-2. Expected Output from Running ant testjdohelper

```
testjdohelper:
    [javac] Compiling 1 source file to E:\Bookcode\build
     [echo] creating runTestJDOHelper.bat
     [echo] Running TestJDOHelper
     [java] -- listing properties --
     [java] javax.jdo.option.RestoreValues=false
     [java] javax.jdo.option.ConnectionURL=fostore:FOStoreTestDB
     [java] javax.jdo.option.Optimistic=false
     [java] javax.jdo.option.ConnectionUserName=JDO
     [java] javax.jdo.option.ConnectionPassword=book
     [java] javax.jdo.option.NontransactionalWrite=false
     [java] javax.jdo.PersistenceManagerFactoryClass=
                  com.sun.jdori.fostore.FOStorePMF
     [java] javax.jdo.option.NontransactionalRead=false
     [java] javax.jdo.option.IgnoreCache=false
     [java] javax.jdo.option.RetainValues=false
     [java] javax.jdo.option.Multithreaded=false
     [java] Got the PMF okay
     [echo] created runTestJDOHelper.bat

BUILD SUCCESSFUL
```

Notice that the build creates the *runTestJDOHelper* batch file. The batch file can be used to run the test again. All generated batch files are placed in the *bookcode-home* directory. The `TestJDOHelper` program has no user interface. It runs to completion in a matter of seconds.

As soon as you can get `TestJDOHelper` to build and run, you have successfully completed the steps necessary to use the remaining client-server programs in this chapter and the next. Chapters 10 and 11 provide additional instructions for the configuration needed to build examples that deploy in the Tomcat and JBoss containers.

Interrogating the PersistenceManagerFactory

`JDOFactory` is a utility class that can interrogate the factory for its default settings and supported options. `JDOFactory` also illustrates the use of the adaptor pattern to localize the vendor dependencies inherent in construction. Unlike most other example programs with this book, `TestFactory` uses construction to acquire a persistence manager factory. The `TestFactory` program writes the results of the interrogation to the console. After obtaining a persistence manager, it terminates. The `TestFactory` class is contained in the `com.ysoft.jdo.book.factory.client` package.

Building and Running the TestFactory Program

To build the `TestFactory` program, go to the *bookants* directory and type *ant testfactory*. This build is part of the *ant learning-programs* build. Listing 8-3 shows some of the expected output when using the reference implementation.

Listing 8-3. Expected Output from Running ant testfactory

```
testfactory:
     [echo] creating runTestFactory.bat
     [echo] Running TestFactory
     [java] Starting TestFactory ...
     [java] Using adaptor class: com.ysoft.jdo.book.factory.jdori.JDORIAdaptor
     [java] The database (FOStoreTestDB.btd) exists
     [java] Using URL: (fostore:FOStoreTestDB)
     [java] Loaded factory adaptor: com.ysoft.jdo.book.factory.jdori.JDORIAdaptor
     [java]
     [java] Supported JDO Options
     [java]    javax.jdo.option.TransientTransactional
     [java]    javax.jdo.option.NontransactionalRead
     [java]    javax.jdo.option.NontransactionalWrite
     [java]    javax.jdo.option.RetainValues
     [java]    javax.jdo.option.RestoreValues
     [java]    javax.jdo.option.Optimistic
     [java]    javax.jdo.option.ApplicationIdentity
     [java]    javax.jdo.option.DatastoreIdentity
     [java]    javax.jdo.option.ArrayList
     [java]    javax.jdo.option.HashMap
     [java]    javax.jdo.option.Hashtable
     [java]    javax.jdo.option.LinkedList
     [java]    javax.jdo.option.TreeMap
```

```
     [java]    javax.jdo.option.TreeSet
     [java]    javax.jdo.option.Vector
     [java]    javax.jdo.option.Array
     [java]    javax.jdo.option.NullCollection
     [java]    javax.jdo.query.JDOQL
     [java] Unsupported JDO Options
     [java]    javax.jdo.option.NonDurableIdentity
     [java]    javax.jdo.option.ChangeApplicationIdentity
     [java]    javax.jdo.option.List
     [java]    javax.jdo.option.Map
     [java] Non-configurable properties
     [java]    Key: VendorName, value: Sun Microsystems
     [java]    Key: VersionNumber, value: 1.0
     [java] Initial PMF transaction settings
     [java]    Optimistic: true
     [java]    Non-trans read: true
     [java]    Non-trans write: false
     [java]    RetainValues: true
     [java]    RestoreValues: true
     [java] Connection information
     [java]    Connection driver: null
     [java]    Connection factory: null
     [java]    Connection factory2: null
     [java]    Connection URL: fostore:FOStoreTestDB
     [java]    Connection UserName: JDO
     [java] Caching info
     [java]    Ignore Cache: true
     [java] Threading setting for PM's
     [java]    Multithreading turned on: false
     [java] This PMF can be serialized
     [java] This PMF implements javax.naming.Referenceable
     [java] Obtained PersistenceManagerFactory
     [java] Just got 1 PersistenceManagers!
     [java] Closing FOStoreDB
     [java] -- All done!
     [echo] created runTestFactory.bat

BUILD SUCCESSFUL
```

Like the previous example, the build, after running `TestFactory`, also creates the batch file *runTestFactory* that can be used to run it again. The `TestFactory` program has no user interface. It runs to completion in a few seconds.

Consuming Java at the MegaCups Company

The people who work at the MegaCups Company love their coffee. In fact, they have gone over the edge and are in need of counseling. During the work day, they incessantly elbow each other around the coffee urn as they seek another cup of coffee to satisfy their never-ending craving for caffeine.

The company has a coffee urn set up in the kitchen that holds 40 cups of coffee. One worker, Mark, adds 20 cups of coffee to the urn every 14 seconds. Four other workers, Frank, Sam, Julie, and Susan, come around for a fresh cup of coffee every 2 seconds. Most real people wait more than 2 seconds before seeking a fresh cup of coffee, but these workers are computer simulations. For that reason, their sense of time is compressed.

Doing the math, you can see that the workers sometimes find that the coffee urn is empty. If this happens too often, they complain to the manager, who either promises to do something about it or ignores the complaint. If he ignores the complaint, the workers quit. Because of the hectic pace at the MegaCups Company, the work day is short, lasting only 1 minute.

In the `MegaCups` program, there is only one persistent object, the coffee urn in the kitchen. Everyone comes to this coffee urn to either add or draw coffee. Each addition or subtraction is done transactionally and the result is committed. Each worker runs in his own thread and uses a separate persistence manager. As a result, each worker acts on his own `CoffeeUrn` object in memory that represents in his transaction the persistent state of the coffee urn found in the datastore. The properties file that configures the persistence manager factory for this example specifies a datastore transaction. As the transactions clash, you can see exactly how your selected JDO implementation handles the transactional semantics for datastore transactions.

The `com.ysoft.jdo.book.coffee` package contains three classes: `MegaCups`, `Worker`, and `CoffeeUrn`. The `Worker` class is contained in the source file for the `MegaCups` class. Excluding comments, blank lines, and lines with a solitary brace, there are approximately 200 lines of code in the two source files. Of these, approximately 20 percent have something to do with the explicit use of JDO. Much of the code that uses JDO explicitly is concerned with setting the program's and the datastore's initial state. To start a transaction, draw a cup of coffee from the coffee urn, and commit the transaction requires only two lines of code that explicitly use JDO.

The `MegaCups` program was created after a prolonged discussion that occurred on *JDOCentral.com* about the semantics of JDO transactions. It illustrates the behavior of datastore transactions. The behavior that you see depends on the transactional semantics of the datastore and the JDO implementation. Most implementations use some form of pessimistic locking in the datastore. For a detailed description of JDO's transactional semantics, see Chapter 4.

Building and Running the MegaCups Program

To build the MegaCups program, go to the *bookants* directory and type *ant megacups*. This build is part of the *ant learning-programs* build.

The program acquires a persistence manager factory that is configured by the property file. For the reference implementation, the properties file is *jdori.properties* contained in the *com/ysoft/jdo/book/coffee* directory. The build copies it to *factory.properties* at the *build/com/ysoft/jdo/book/coffee* directory.

Listing 8-4 shows some of the expected output from the build when using the reference implementation.

Listing 8-4. Expected Output from Running ant megacups

```
megacups:
    [javac] Compiling 2 source files to E:\Bookcode\build
     [echo] returned from com/ysoft/jdo/book/coffee/build.xml
     [copy] Copying 1 file to E:\Bookcode\build\com\ysoft\jdo\book\coffee
     [copy] Copying 1 file to E:\Bookcode\enhanced\com\ysoft\jdo\book
     [java] done.
     [echo] creating runMegaCups.bat

BUILD SUCCESSFUL
```

Unlike the previous build targets, the build for the MegaCups program does not run the program. Instead, you must change to the *bookcode-home* directory and execute the *runMegaCups* batch file. The MegaCups program does not have a user interface. It takes about a minute to run. The expected output from running the program will start off looking something like the output shown in Listing 8-5. This output was obtained from using the reference implementation. Notice the number that follows "kitchen" within the brackets. This number is incremented each time one cup of coffee is drawn from the urn. It numbers the order of changes to the coffee urn in the kitchen.

Listing 8-5. Sample Output from the MegaCups *Program*

```
E:\Bookcode>runMegaCups
Using property file: com/ysoft/jdo/book/coffee/factory.properties
This program will end in one minute
Mark found: CoffeeUrn [Kitchen-0] contains 0 cups
Sam found: CoffeeUrn [Kitchen-0] contains 0 cups
Julie found: CoffeeUrn [Kitchen-0] contains 0 cups
Susan found: CoffeeUrn [Kitchen-0] contains 0 cups
```

```
Frank found: CoffeeUrn [Kitchen-0] contains 0 cups
Mark added coffee to CoffeeUrn [Kitchen-0] contains 20 cups
Sam drank a cup of coffee from CoffeeUrn [Kitchen-1] contains 19 cups
Julie drank a cup of coffee from CoffeeUrn [Kitchen-2] contains 18 cups
Susan drank a cup of coffee from CoffeeUrn [Kitchen-3] contains 17 cups
Frank drank a cup of coffee from CoffeeUrn [Kitchen-4] contains 16 cups
Sam drank a cup of coffee from CoffeeUrn [Kitchen-5] contains 15 cups
Julie drank a cup of coffee from CoffeeUrn [Kitchen-6] contains 14 cups
Susan drank a cup of coffee from CoffeeUrn [Kitchen-7] contains 13 cups
Frank drank a cup of coffee from CoffeeUrn [Kitchen-8] contains 12 cups
```

If you are using the reference implementation, then the datastore files require a one-time initialization. If you see the following error:

```
Using property file: com/ysoft/jdo/book/coffee/factory.properties
javax.jdo.JDOFatalDataStoreException: com.sun.jdori.fostore.FOStoreLoginException:
      Could not login user JDO to database FOStoreTestDB.
NestedThrowables:
org.netbeans.modules.mdr.persistence.StorageIOException
```

then you want to go to the *factory.properties* file located in the *build/com/ysoft/jdo/book/coffee* directory and uncomment the second line, so that it reads as follows:

```
#set this to true to create valid datastore files
com.sun.jdori.option.ConnectionCreate=true
```

After running the `MegaCups` program again, the files *FOStoreTestDB.btd* and *FOStoreTestDB.btx* should exist and be larger than zero bytes in size. To prevent the `MegaCups` program from continually creating new datastore files, recomment the option string in the *factory.properties* file after valid datastore files have been created.

The `MegaCups` program accepts a number of optional command line parameters. You can specify the number of workers and their names by using the following parameters:

```
-names Tom Dick Harry
```

Pick the names you like and add as many as you want. The first person named is given the responsibility to fill the coffee urn. You can also prevent anyone from filling the coffee urn by specifying this option:

```
-nofilling
```

Multiple invocations of the `MegaCups` program can run simultaneously. Most commercial datastores support multiuser access, but the datastore for the reference implementation does not support concurrent access from multiple JVMs.

The Console User Interface

The JDO Learning Tools described so far do not provide an interactive user interface. The next two programs in the JDO Learning Tools use a simple console user interface. Although the workings of this console interface are incidental to the purposes of this book, the interface is also unfamiliar to you. This section gives you an idea of what the console interface is like. It also gives you an idea of how you can modify the programs that use it.

When the interface comes up, it prompts you with two lines:

```
enter command:
-->
```

You can control the configuration of the prompt by modifying the properties in the *package.properties* file found in the *com/ysoft/jdo/book/common/console* directory. The default configuration works well for highlighting the user's input in the listings provided in this chapter.

There are only three things to remember about the console interface. One, it will not do anything unless you enter a command. Two, every program has at least two commands, *quit* and *help*. The *quit* command terminates the program. The *help* command lists all of the commands, except *help*, that the program recognizes. Three, if you enter a command string that the interface does not recognize, it outputs a question mark and prompts again, as the following interaction shows:

```
enter command:
--> whatever
?
enter command:
-->
```

Some of the command strings are wordy. In some cases, there is more than one command string that will activate the command. The additional command strings provide flexibility in activating a command. For example, the command string

```
add data object
```

may have the additional command strings

```
add
add object
```

There are only two ways to tell what the additional command strings are. Either try something and see if it is accepted, or look at the source code.

Each command is implemented in a separate class that extends the base class `Command`, which is found in the `com.ysoft.jdo.book.common.console` package. In the constructor for the derived command class, there is a call to the `super` constructor of `Command`. One of the parameters to this constructor is a list of command strings. For example, the `Add` command in the `Library` program has the following constructor:

```
public Add(UIClient c)
   {
   super(c, new String[]
         {
         "add data object",
         "add",
         "add object",
         });
   }
```

Any of the command strings activate the command. The first one is the preferred command string, and for that reason, it is shown in the help listing. If you want, you can add more command strings to any command. Within the application, each command string should be unique.

When you enter a command, you may be prompted for command parameters. Often you cannot get out of the command without entering the additional information. Since no lives are at stake, enter something to make the interface happy.

The command classes are package-level classes contained within the source file for the public application class. For example, the source file *Library.java* contains the source for the public class `Library`, and also the source for all of the command classes, such as `Add`, `Delete`, and `Find`. Adding additional commands is easy. Pick a command class that is close in behavior to what you want, then copy, paste, and modify. Don't forget to add an instance of the new class to the application's list of `Command` objects.

Querying the Small Town Library

The `Library` program prototypes a simple system for a small town library. Figure 2-3 in Chapter 2 shows the UML model for application data classes used by the `Library`

program. There are four application data classes in Figure 2-3: `Book`, `Category`, `Borrower`, and `Volunteer`. Using the `Library` program, you can manipulate the persistent objects, define queries, and view the results.

The `Library` program is built from six primary source files, the four application data classes, the `Library` class and its command classes, and the `LibraryHandler` data service. These are contained in the `com.ysoft.jdo.book.library` and `com.ysoft.jdo.book.library.client` packages. All of the explicit use of JDO occurs within the `LibraryHandler` data service. The console interface handles all exceptions thrown by the implicit use of JDO. The persistence manager factory is configured by a properties file that varies by the JDO implementation. For the reference implementation, the file is named *jdori.properties* and is found in the *com/ysoft/jdo/book/library* directory. The build copies it to the *build/com/ysoft/jdo/book/library* directory and renames it to *factory.properties*.

Building the Library Program

To build the program, go to the *bookants* directory and type *ant library*. This build is part of the *ant learning-programs* build. Listing 8-6 shows some of the expected output from the build when using the reference implementation.

Listing 8-6. Expected Output from Running ant library

```
library:
    [javac] Compiling 7 source files to E:\Bookcode\build
     [echo] returned from com/ysoft/jdo/book/library/build.xml
     [copy] Copying 1 file to E:\Bookcode\enhanced\com\ysoft\jdo\book
     [copy] Copying 1 file to E:\Bookcode\build\com\ysoft\jdo\book\library
     [echo] Enhancing the persistent classes of library
     [java] done.
     [echo] creating runLibrary.bat

BUILD SUCCESSFUL
```

To run the `Library` program, go to the *bookcode-home* directory and type *runLibrary*.

Using the Library Commands

When you enter the *help* command after starting the `Library` program, you see all the commands that it supports. The expected output is shown in Listing 8-7.

Listing 8-7. Example of Help Output from the `Library` *Program*

```
E:\Bookcode>runlibrary
enter command:
--> help
commands:
   quit
   begin
   commit
   rollback
   get pm config
   view attributes
   define query variable
   find all
   find
   find in results
   add data object
   delete data object
   view volunteer
   view borrower
   view book
   view category
   borrow book
   return book
   modify volunteer
   modify book
   populate database
   clear database
enter command:
-->
```

As Listing 8-7 shows, the program recognizes a large number of commands. Begin by populating the database. This will add seven books, six categories, three borrowers, and one volunteer to the datastore. The properties file sets to false all the Boolean properties of the `PersistenceManager` and `Transaction`. This can be verified by executing the *get pm config* command.

The three commands *begin*, *commit*, and *rollback* allow you to control the transactional boundaries. When you execute a *begin* command, a JDO datastore transaction starts. If you then execute a series of *find all* commands, you can view the summary information for all objects in the datastore. There is an apparent bug in the 1.0 reference implementation that prevents the capture of the identity string on the first transaction for an object. Finding all objects and executing a *commit*

works around this bug. If you find all the objects again, you will see the value of the identity strings in the output.

A large number of commands allows you to manipulate the state of the persistent objects. The *add*, *delete*, *view*, *borrow*, *return*, and *modify* commands allow you to add books, categories, borrowers, and volunteers; delete the same; view all the information about them; borrow books; return them; and modify information about books and volunteers. The *view* commands track the last list of objects that were presented. As a result, if you view a category and it lists two books, then the *view book* command will present that list of two books to choose from.

Running Queries in the Library Program

The `Library` program's primary purpose is to exercise the JDO query language. The *find* command will query against the extent, while the *find in results* command will query against the last collection of query results. The *view attributes* command shows the names and types of all the attributes of the application data classes. Finally, the *define query variable* command allows you to define a query variable for use in navigating collections within JDOQL. The effect of defining a query variable lasts only until the next query is executed. Consequently, the query variable must be defined before executing the query that will use it.

As an example, suppose that you want to know what categories of books interest Tom. Listing 8-8 shows the user interactions to find that information.

Listing 8-8. User Commands to Find All the Categories That Interest Tom

```
enter command:
--> begin
Okay
enter command:
--> define variable
Enter query variable declaration:
--> Book b
Okay
enter command:
--> find
Find what type of objects:
   1. Book
   2. Borrower
   3. Volunteer
   4. Category
Enter selection:
--> 4
```

```
Enter query string:
--> books.contains(b) && b.borrower.name == "Tom"
Find
   Class: com.ysoft.jdo.book.library.Category
   Filter: books.contains(b) && b.borrower.name == "Tom"
   Variables: Book b
Found 2 objects in the results
   category [OID: 103-12] "Sportsman"
   category [OID: 103-11] "Outdoors"
enter command:
-->
```

Although there are only four application data classes, the object model of the library supports a variety of queries. For example, to find the books that have been borrowed by a volunteer, use the `Book` extent and the following query string:

```
borrower.volunteer != null
```

Your answer for the default object population should be as follows:

```
Found 2 objects in the results
   book [OID: 102-13] "Gone Sailing" checked out: Mon Aug 26 08:23:10 EDT 2002
   book [OID: 102-12] "Gone Hunting" checked out: Mon Aug 26 08:23:10 EDT 2002
```

The OID values may very well be different in your datastore, and the date when the books were checked out will certainly be different.

To find all the books that are in categories that interest Harry, use the `Book` extent, and define the query variables as shown here:

```
Book b; Category c;
```

Then use the following query string:

```
categories.contains(c) && (c.books.contains(b) && b.borrower.name == "Harry")
```

Your answer for the default object population should be as follows:

```
Found 1 objects in the results
   book [OID: 102-16] "Gone to Work" checked out: Mon Aug 26 08:23:10 EDT 2002
```

Now that you have the general idea, perhaps you are ready for a challenge. Can you find all the categories that have books borrowed by more than one borrower? Hint: output similar to the following is expected from the query when it runs on the default population.

```
Found 2 objects in the results
   category [OID: 103-12] "Sportsman"
   category [OID: 103-11] "Outdoors"
```

Monitoring the State of Persistent Apples

The `StateTracker` program allows you to use nearly all of the explicit and implicit operations of JDO while monitoring the persistent, transactional, and unmanaged fields of persistent, transactional, and unmanaged apples.

Figure 8-1 diagrams the relationships of the main classes and interfaces of the `StateTracker` program. The `StateTracker` class implements the user interface and is the client of the `Monitor` and `StateHandler` services. It creates new apples and worms and modifies the state of existing ones. The source file *StateTracker.java* contains all of the command classes of the `StateTracker` program. The application classes and interfaces shown in Figure 8-1 are contained in the `com.ysoft.jdo.book.statetracker` and `com.ysoft.jdo.book.statetracker.client` packages.

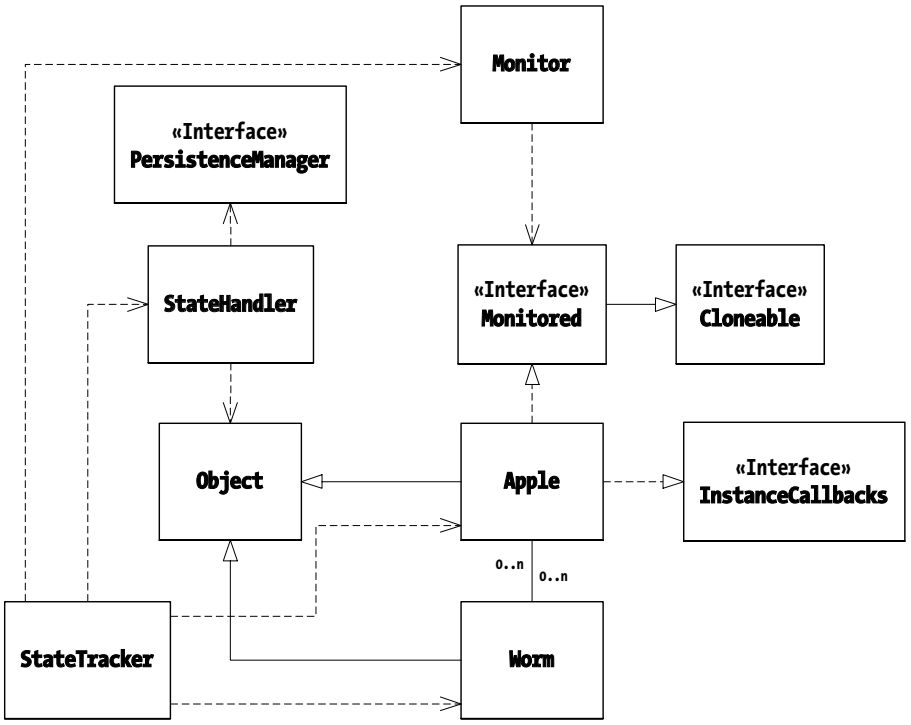

Figure 8-1. The classes and interfaces of the `StateTracker` *program*

Figure 8-1 diagrams ten classes and interfaces. There are two application data classes, `Apple` and `Worm`. These have an n-m relationship to each other. The worms in the `StateTracker` program have the transcendental ability to exist in more than one apple at a time. The major purpose of the worms is to provide persistent fields that are not in the apple's default fetch group. The `Apple` class divides its fields into three groups, persistent, transactional, and unmanaged. The persistent group has the five fields shown in Listing 8-9. The fields were selected to give a representative sample of persistent fields. The first three fields are in the default fetch group, while the remaining two are not. The set of transactional (but not persistent) fields and the set of unmanaged fields are identical in type and similarly named.

Listing 8-9. The Persistent Fields of the Apple Class

```
private String    persistentName;
private int       persistentSize;
private Date      persistentPicked;
private HashSet   persistentWorms;
private Worm      persistentHeadWorm;
```

The remaining classes and interfaces in Figure 8-1 serve the following purposes. The `Apple` class implements the `Monitored` interface. The `Monitor` service uses the `Monitored` interface to determine the apple's management state without affecting it. The `Monitored` interface also ensures that `Apple` has a public `clone` method that is used by the `StateTracker` to snoop on the state of an apple without affecting the apple's managed state. The `StateHandler` is the application service that uses the persistence manager. It handles objects, and knows nothing about apples and worms. The `Apple` class implements `InstanceCallbacks`. These callback methods serve two purposes: they capture the persistent object's identity string, and they provide notification to the user interface when the callbacks occur.

Building the StateTracker Program

To build the `StateTracker` program, go to the *bookants* directory and type *ant statetracker*. This build is part of the *ant learning-programs* build. Listing 8-10 shows some of the expected output from the build when using the reference implementation.

Listing 8-10. Expected Output from Running ant statetracker

```
statetracker:
    [javac] Compiling 7 source files to E:\Bookcode\build
     [echo] returned from com/ysoft/jdo/book/statetracker/build.xml
     [copy] Copying 1 file to E:\Bookcode\enhanced\com\ysoft\jdo\book
     [java] done.
     [echo] creating runStatetracker.bat

BUILD SUCCESSFUL
```

To run the `StateTracker` program, go to the *bookcode-home* directory and type *runStatetracker*.

Getting Started with the StateTracker Commands

If you enter the *help* command after starting the `StateTracker` program, you see all the commands that it supports. The expected output is shown in Listing 8-11.

Listing 8-11. Example of Help Output from the `StateTracker` *Program*

```
E:\Bookcode>runstatetracker
Using adaptor class: com.ysoft.jdo.book.factory.jdori.JDORIAdaptor
The database (FOStoreTestDB.btd) exists
Using URL: (fostore:FOStoreTestDB)
enter command:
--> help
commands:
   quit
   begin
   commit
   rollback
   active
   find all
   add apple
   select apple
```

```
   modify apple
   add worm
   delete worm
   snoop
   view
   get JDO state
   make persistent
   delete persistent
   make transactional
   make nontransactional
   make transient
   evict
   evict all
   refresh
   refresh all
   retrieve
   tickle default fetch group
   dirty
   toss exception
   configure
   configuration
   open
   is open
   close
enter command:
-->
```

As Listing 8-11 shows, the program recognizes a large number of commands. The commands *begin*, *commit*, *rollback*, and *active* allow you to control and monitor transactional boundaries. A good place to start is to add a few worms. Their only attribute is a name. For example, the following interaction adds one new worm named Henry:

```
enter command:
--> add worm
Enter worm's name:
--> Henry
Okay, but worms are made persistent only by being in a persistent apple
enter command:
-->
```

A new worm remains unmanaged. It becomes persistent only when it is reached from a persistent apple. After creating a few worms, make a new apple. Listing 8-12 shows a sample interaction that adds a new McIntosh apple with three worms.

Listing 8-12. User Commands to Create a New McIntosh Apple with Three Worms

```
enter command:
--> add apple
Enter apple's name:
--> McIntosh
Enter apple's size (> 0):
--> 3
Enter date picked (mm-dd-yy):
--> 10-15-02
   Date accepted:Tue Oct 15 00:00:00 EDT 2002
Add a worm?:
   1. true
   2. false
Enter selection:
--> 1
Pick a worm:
   1. Worm Henry
   2. Worm Martha
   3. Worm Jack
Enter selection:
--> 1
Add a worm?:
   1. true
   2. false
Enter selection:
--> 1
Pick a worm:
   1. Worm Martha
   2. Worm Jack
Enter selection:
--> 1
Add a worm?:
   1. true
   2. false
Enter selection:
--> 1
```

```
Pick a worm:
   1. Worm Jack
Enter selection:
--> 1
Add a worm?:
   1. true
   2. false
Enter selection:
--> 2
Pick the head worm:
   1. Worm Henry
   2. Worm Martha
   3. Worm Jack
Enter selection:
--> 2
Okay, the new transient apple has been added to the selection list
enter command:
-->
```

The new apple remains unmanaged. To see this, first use the *select apple* command to select the apple from the current list of apples. Then use the *get JDO state* command to determine the current JDO management state of the selected apple.

```
enter command:
--> select apple
Select an apple:
   1. Apple transientName: McIntosh
Enter selection:
--> 1
Okay
enter command:
--> get JDO state
Apple transientName: McIntosh is in JDO state transient
enter command:
-->
```

Because all actions so far have occurred on unmanaged state, there has been no reason to start a transaction.

Next, execute the *begin* and the *make persistent* commands to make the unmanaged McIntosh apple persistent. Now execute the *view* and *get JDO state* commands to determine the unmanaged, transactional, and persistent state of the apple and to determine its management state. Using any implementation, you should see output like the following, which was produced by the reference implementation:

```
enter command:
--> view
Viewing managed state for: OID: 105-12 [JVM ID:4066855]
   transient state: McIntosh, 3, 10-15-02, Worm Martha,
         3 worms {Worm Martha,Worm Jack,Worm Henry}
   transactional state: McIntosh, 3, 10-15-02, Worm Martha,
         3 worms {Worm Henry,Worm Martha,Worm Jack}
   persistent state: McIntosh, 3, 10-15-02, Worm Martha,
         3 worms {Worm Henry,Worm Martha,Worm Jack}
enter command:
--> get state
OID: 105-12 [JVM ID:4066855] is in JDO state persistent-new
enter command:
-->
```

Notice that all three states are the same. That is because the values entered when the apple was created were copied to all three sets of fields, as a way to reduce the amount of user input. As expected, the management state after the *make persistent* command is persistent-new.

If you have not executed the *configure* command, then all transactional attributes are off at this point, and the *configuration* commands returns the following output:

```
--> configuration
Current transaction properties: active, !Opt, !RetainV, !RestoreV, !NTR, !NTW
enter command:
-->
```

At this point, commit the transaction and ask for the apple's management state. You should see output that looks like the interaction in Listing 8-13, which was produced by the reference implementation.

Listing 8-13. Sample Output from Committing the New McIntosh Apple

```
enter command:
--> commit
Synchronization.beforeCompletion called
OID: 105-12 [JVM ID:4066855] jdoPreStore
OID: 105-12 [JVM ID:4066855] jdoPreClear
Synchronization.afterCompletion called with status: committed
Okay
enter command:
--> get state
OID: 105-12 [JVM ID:4066855] is in JDO state hollow
enter command:
-->
```

After committing the transaction, you will get an exception if you execute the *view* command because a transaction is not active. Instead run the *snoop* command. The *snoop* command produces a view of the object without affecting the managed state or requiring a transaction. The expected output will look like the output in Listing 8-14.

Listing 8-14. Sample Output from Snooping on the Hollow Apple

```
enter command:
--> snoop
Viewing raw state for: OID: 105-12 [JVM ID:8083121]
   transient state: McIntosh, 3, 10-15-02, OID: 106-21,
         3 worms {OID: 106-21, OID: 106-22, OID: 106-20}
   transactional state: null, 0, no date, null, null worms
   persistent state: null, 0, no date, null, null worms
enter command:
-->
```

There are four things to notice in Listing 8-14. One, the JVM ID has changed from Listing 8-13. The change occurs because the *snoop* command views a clone of the original apple. (Remember from Chapter 5 that, by default, cloning a persistent object gets a snapshot of the current memory state without invoking transparent persistence.) Two, the persistent state has Java default values. This is expected since the object is in the hollow management state. Three, the transactional state also has Java default values. This is an unexpected outcome. The specification describes eviction only in terms of clearing the persistent state. It says nothing about clearing the nonpersistent and transactional state. In fact, the specification strongly implies that eviction does not clear the nonpersistent and transactional

state. This behavior has been reported as a bug in the JDO reference implementation at the Java bug parade (`http://developer.java.sun.com/developer/bugParade`).

Finally, note that the transient state is not changed, except that all of the worms are now unnamed. The `transientWorms` field and the `transientHeadWorm` field both point to persistent objects even though the fields are unmanaged. When the new apple was created, the same worms were used to set the unmanaged, transactional, and persistent fields. As a result, all worm fields are referring to the same worms, which became persistent when the apple became persistent. When the *snoop* command executes, it clones the apple, but it does not clone the worms. The worm's `toString` method catches the exception that results from trying to examine the persistent `name` of a `Worm` outside of a transaction, and it recovers from the exception by returning the worm's identity string instead of its name.

Brief Look at the Other StateTracker Commands

The previous section describes many commands that the `StateTracker` program recognizes. This section describes the remaining commands.

Many of the command strings are self-describing. For example, the *make transactional* command will make a nontransactional object transactional. In some cases, for the command to succeed, the implementation must support the appropriate implementation options. An unmanaged object cannot be made transactional unless the implementation supports, as the reference implementation does, the `javax.jdo.option.TransientTransactional` feature. Most commands operate on the selected apple, but some operate on a set of apples. The *evict all* command evicts all persistent-clean apples (and worms) and the *refresh all* command refreshes all transactional apples and worms.

The *configure* command sets the five properties of the `Transaction` object: *Optimistic*, *RetainValues*, *RestoreValues*, *NontransactionalRead*, and *NontransactionalWrite*. The *is open* command checks whether the persistence manager is open. The *open* and *close* commands open and close the persistence manager.

The *toss exception* command sets a flag in the transaction's `Synchronization` object that will cause it to throw a `JDOUserException` on the next commit. It's a one-shot setting that does not block a subsequent commit. The *tickle default fetch group* command reads a couple of fields of the default fetch group and outputs a message with their values. It will cause the default fetch group to load in most cases. The *dirty* command will call the `makeDirty` method in `JDOHelper` for the selected managed field.

The `StateTracker` program has been invaluable in writing this book, and you will find it very useful for test driving your selected JDO implementation.

Using the Commercial Implementations

Three commercial implementations are featured in this book, but you can use the JDO Learning Tools with any implementation that supports Ant build scripts. The code provided with this book provides build scripts for the three implementations Kodo, Lido, and IntelliBO. The 1.0 release of the JDO Learning Tools supports the Microsoft Windows operating systems. Undoubtedly, the open source community will extend support to other JDO implementations and operating systems. You will have to find or write a script for any other implementation. The JDO Learning Tools are demanding programs, and you may very well find bugs in the commercial implementations as a result of using these programs. If you find bugs, report them to the vendor.

Using the Kodo Implementation

When using the Kodo implementation, configure the *kodo.properties* file, and copy it to *custom.properties*. The main *build.xml* script will invoke the tool script *kodo.xml*. The tool script runs the Kodo verifier, Kodo enhancer, and Kodo schema tool. The Kodo implementation supports live schema generation and evolution. As a result, when the schema tool is invoked, there is no need to take any further steps to update the database schema. Because the schema tool is fairly time consuming, the Kodo build script with the JDO Learning Tools optimizes the use of the schema tool by detecting whether the enhanced classes have changed. This detection is not foolproof. To force schema generation, use the *-Dschema=generate* option with the *ant* command as shown in the following command line:

```
ant -Dschema=generate megacups
```

The *learning-programs* target sets this property for you.

Using the Lido Implementation

When using the Lido implementation, configure the *lido.properties* file, and copy it to *custom.properties*. The main *build.xml* script invokes the tool script *lido.xml*. The tool script runs the Lido enhancer and schema tool. Because the use of the schema tool is somewhat time consuming, it runs only when the schema property is defined. This can be done at the command line when invoking Ant, as the following line shows:

```
ant -Dschema=generate megacups
```

The *learning-programs* target sets this property for you.

The Lido tool script has been configured to have the schema tool output SQL files. These files are placed in the *bookcode-home* directory. Their file names are composed of the lowest package directory plus the SQL extension, such as *coffee.sql*. After generating the SQL files, you will need to execute the portion of the SQL script that is appropriate for your schema evolution. You may have to drop tables if you alter the definition of persistent fields in the application data classes.

Using the IntelliBO Implementation

When using the IntelliBO implementation, configure the *intellibo.properties* file, and copy it to *custom.properties*. The main *build.xml* script invokes the tool script *intellibo.xml*. The tool script runs the IntelliBO verifier, enhancer, and schema tools. Because the use of the schema tool is somewhat time consuming, it runs only when the schema property is defined. This can be done at the command line when invoking Ant, as the following line shows:

```
ant -Dschema=generate megacups
```

The *learning-programs* target sets this property for you.

The IntelliBO tool script has been configured to have the schema tool output SQL files. These files are placed in the package directory under the *enhanced* directory. You will find three SQL files: *create.sql*, *drop.sql*, and *select.sql*. After generating the SQL files, you will need to execute the portion of the SQL scripts that are appropriate for your schema evolution.

Summary

The first five JDO Learning Tools range from elementary to advanced in their use of JDO. Each of them can help you understand the capabilities and limitations of JDO. Do not let the console interface, which is rudimentary, hide their true merit from you. By using them and understanding the results that you obtain, you can become a JDO expert. Better than any book, they can teach you the underlying logic of JDO's behavior. The understanding that you gain will help make your first project that uses JDO a complete success.

The next chapter examines a Swing client-server application that satisfies the requirements of a simple reservation system. Unlike the five JDO Learning Tools programs covered in this chapter, the JDO Learning Tools programs presented in the remaining chapters have the flavor of real-world applications.