Hands-on Java Data Objects

Presented by developerWorks, your source for great tutorials

`ibm.com/developerWorks`

Table of Contents

If you're viewing this document online, you can click any of the topics below to link directly to that section.

Section 1. About this tutorial

What is this tutorial about?

Java Data Objects (JDO) is a new technology from Sun Microsystems. While somewhat immature -- the 1.0 specification just came out -- JDO is very promising, and it fills a big gap in the area of database programming. For Java developers, JDO offers the first standardized, completely object-oriented approach to object persistence. Compared with the other options in this area, JDO's advantage is that it is fairly easy to work with and poses minimal disruption to the original Java source code. And, whereas both JDBC and EJB Container Managed Persistence (EJB CMP) can be challenging for even seasoned programmers, JDO works hard to simplify some of the most intricate aspects of database programming in the Java language. In this tutorial, we'll use discussion, code samples, and hands-on exercises to learn about the practical application of JDO.

Should I take this tutorial?

This tutorial is designed for intermediate to advanced Java developers. To get the most out of the tutorial, you should be experienced with the Java 2 platform and have a good understanding of how relational databases work. Some knowledge of JDBC will be helpful. This tutorial is especially recommended for developers who are looking for ways to deal with persistence in objects. If you don't want to take on the weight of EJB technology, yet you also don't want to deal with the relational semantics that come with JDBC, JDO may offer the perfect in-between solution for your needs.

Code samples and installation requirements

JDO can be used in conjunction with the Java 2 platform, Standard Edition platform. The JDO download from Sun Microsystems does come with a reference implementation, but at the time of this writing the reference implementation is considered unreliable. The tools aren't yet sufficiently robust for general usage.

A more solid JDO implementation is available for free (for educational purposes, with registration) from LIBeLIS. Exercises in this tutorial are based on the LIBeLIS JDO implementation. In addition to the JDO implementation, you'll need a JDBC driver and a relational database to complete the exercises. With the exception of the JDO implementation, all the tools used for the exercises are open source.

You'll need the following technologies and resources to complete the exercises in this tutorial:

- The *LIBeLIS Community Edition JDO* (LiDO). Site registration is required for download.
- The *Java 2 platform, Standard Edition.*
- A standard editor and a JDK to compile and run the examples.
- The *MySQL* open source relational database.
- The *MM.MySQL* open source JDBC drivers for use with MySQL.
- The jdo-source.zip binaries and source code for the examples.

See Resources on page 30 for more details on these items, plus pointers to additional information.

You will learn quite a bit about LiDO as we go along, so no prior knowledge of the JDO implementation is necessary. The same is true for MySQL, because LiDO will take care of most of the details of working with the database for you. You will, however, need to go through the MySQL installation process and learn how to create and drop databases. Once you've installed the MM.MySQL and JDBC drivers in the CLASSPATH, you won't have to do any maintenance or tasks specific to MM.MySQL.

A note about the tutorial

JDO is a hot-off-the presses technology. This tutorial was based on the original release date of the JDO 1.0 specification. The software used for the exercises was also new to the market at the time of writing. Given the newness of all the technologies in use, this tutorial may be a little more of a bumpy ride than you've come to expect -- but it's also an early look at an exciting technology. Think of it as an adventure!

Every effort has been made to ensure this tutorial provides more than enough information to get you started with JDO. You will definitely want to follow up on what you learn here, and stay abreast of the many changes to come as the technology develops. JDO is well worth learning about.

About the author

Paul Monday is a software architect at J.D. Edwards. He has six years of hands-on Java platform experience in a broad range of technologies including J2EE, Jini, and Jiro technology. After graduating from Winona State University in Winona, MN, Paul earned a Master's degree in Computer Science from Washington State University in Pullman, WA. His focus for the Master's degree was operating systems, specifically the Linux operating system. After receiving his degree, Paul worked for IBM on the SanFrancisco Project, Imation Corp. on a storage appliance initiative, and Retek Inc. on an enterprise application integration (EAI) project.

Paul has written two books, *SanFrancisco Component Framework: An Introduction*, co-authored with James Carey and Mary Dangler; and *The Jiro Technology Programmer's Guide and Federated Management Architecture*, co-authored with William Conner from Sun Microsystems. Both books were published with Addison-Wesley. He has also written a variety of articles for print and online technical resources.

In his spare time, Paul has worked with Eric Olson on *Stereo Beacon*, a distributed MP3 player that uses a centralized event service and distributed event implementations for communication. He has also built a mobile agent framework for the Jiro technology platform and is working to usher the Jiro project into the Jini community under the **fsp** (Federated Services Project) community project. Contact Paul at *pmonday@attbi.com*.

Section 2. Overview

Introduction to JDO

The Java Data Objects (JDO) specification from Sun Microsystems aims to provide Java programmers with a much-needed, lightweight view of object-oriented persistent data. JDO frees developers to interact with objects in a natural way, providing an alternative to JDBC or Enterprise JavaBeans with Bean Managed Persistence or Container Managed Persistence, the two previous options in this area. One of JDO's biggest advantages is that it lets us concentrate on developing a correct class model, rather than on developing a relational class model. Further, JDO attempts to abstract the actual data source from the class model. This allows different types of data sources to be "plugged in" to a deployed system. Rather than being stuck with a relational database on the back end, we can now use object databases or even XML files residing in the filesystem as our data sources.

EJB Container Managed Persistence (EJB CMP) has been a preferred method of dealing with object persistence on relational databases. But using EJB CMP entails adherence to the EJB programming model, the use of heavyweight development and deployment tools, and a steep learning curve for all the developers involved. By preserving the object semantics and moving the burden of programming-model adherence to a set of tools, JDO makes object-based persistence much more accessible to Java developers.

We'll use discussion, code examples, and hands-on exercises to learn about many JDO concepts. Because the JDO specification is significant in scope, we won't go very far into advanced JDO programming concepts here, but we will hit the highlights.

In this section of the tutorial, we'll take a high-level look at the JDO architecture and its parts, including tools for development and deployment. We'll learn more about the individual aspects of the architecture as we work with them to complete the exercises later in the tutorial. In the second half of this section, we'll go over the installation and setup requirements for each of the technologies we'll be using. By the end of this section, you will have had a first look at the JDO architecture.

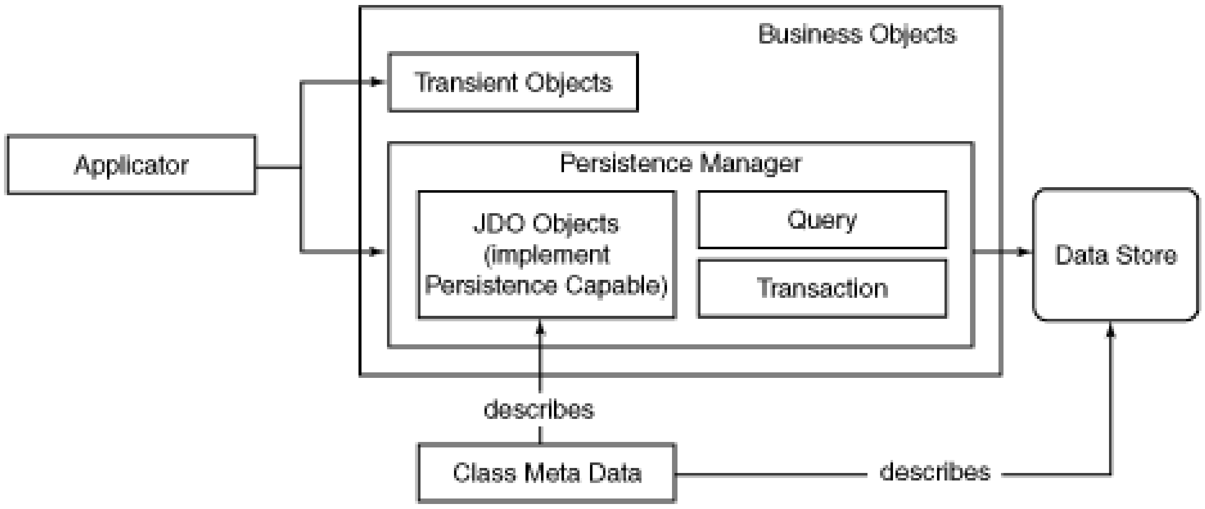
The JDO architecture

From a high-level perspective, an application that is built using JDO consists of the following components:

- **The application itself**, which must be compliant with JDO and its APIs
- A **set of business objects** consisting of the following two types:
  - *Transient business objects*, which are typical classes that will not be made persistent
  - *Persistent business objects*, which are classes that are accessed through a persistence manager and implement JDO's `PersistenceCapable` interface
- A **persistence manager**, which is used by the application to access queries and transactions

- An **underlying data store**, which is used by the persistence manager to provide persistence and operations against the data store
- **Class metadata**, which is an XML file that describes the relationships and contents of the persistent classes as well as the data store itself

The figure below illustrates the JDO architecture.



Persistence management with JDO

In theory, you need to use JDO only from the classes that make up your application (not the business objects themselves). But this scenario doesn't account for the times that business objects need to leverage query mechanisms to facilitate business operations. Most of your application's operations will originate with a call to JDO's `PersistenceManager`. The `PersistenceManagerFactory` facilitates the location of the correct `PersistenceManager` for a particular data source. More than one persistence manager can be active to enable the use of multiple data sources for an application.

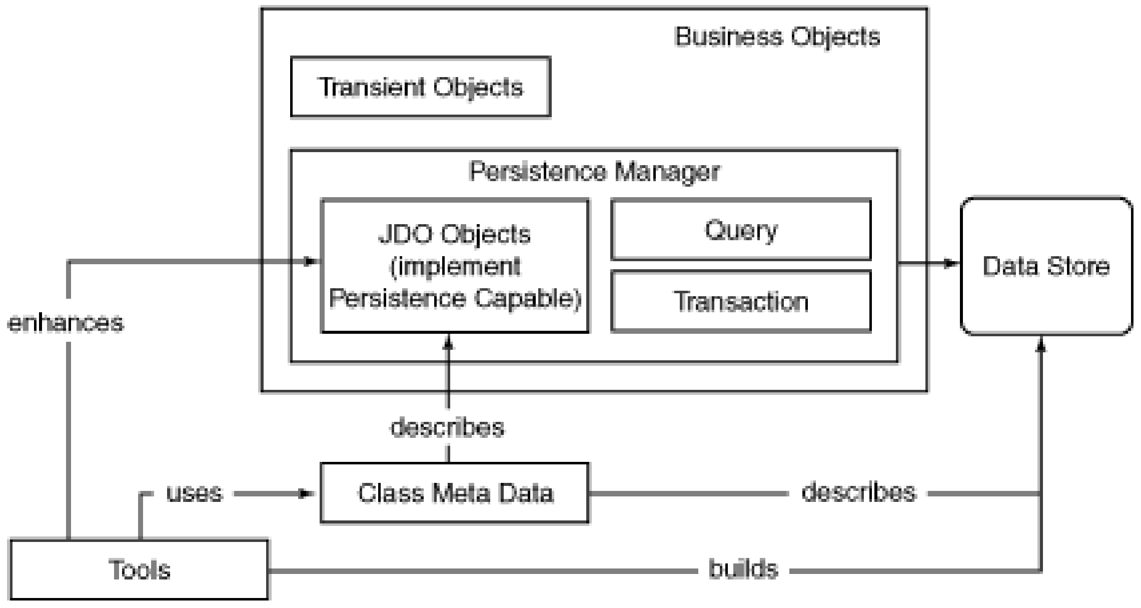
The JDO specification states that a business object that will be persistent must implement the `PersistentCapable` interface. Fortunately, you don't have to worry about coding the implementation; tools that come with the JDO implementation that you choose will take care of it for you.

Tools for development and deployment

JDO's development tools use class metadata to enhance classes that will be persistent, as well as to build table schemas to support those classes. We'll devote some time further on in the tutorial to talking about enhancements to class objects, as enhancement is among the most controversial of the JDO practices. Basically, JDO's tools change class bytecode by adding to it the `PersistenceCapable` interface and any code specific to a particular JDO implementation. (This, of course, assumes that these enhancements have not already been coded in by the developer.) The specification does not require the use of bytecode modification; source code modification could be used as well. Most JDO vendors will likely use the BCEL library from Apache to do bytecode modification rather than taking the potentially messy source code modification route.

The class enhancer will also generally add mutator methods that conform to the JavaBeans specification. To be in step with this aspect of the JDO implementation, you will likely want to adopt JavaBeans semantics from the beginning. All the classes developed in this tutorial use the simple JavaBeans method specification to establish a get/set method for each instance variable.

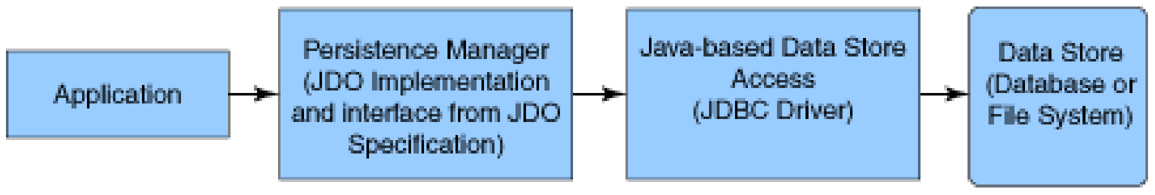
The figure below illustrates the relationship between JDO tools, the database, class metadata, and business objects that are going to be made persistent.



Software requirements

To truly leverage the JDO specification, you'll need to get your hands on a decent-sized stack of software beyond the Java 2 platform software. First, you'll need a JDO implementation, which will typically come bundled with the JDO interfaces from the JDO specification. You'll also need a relational database and a JDBC driver to support it.

The figure below illustrates the structure of the software stack to support JDO.


Application
Persistence Manager (JDO Implementation and interface from JDO Specification)
Java-based Data Store Access (JDBC Driver)
Data Store (Database or File System)

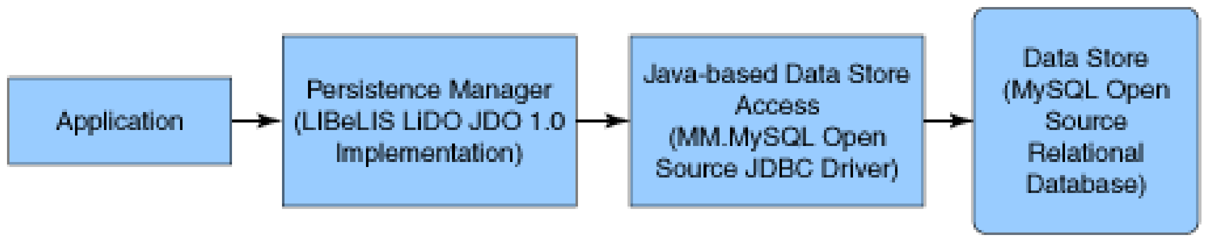
Obviously, the specifics of these requirements will vary according to the data source being used and the JDO implementation itself. All the examples and exercises in this tutorial make use of open source technology wherever possible. Where an open source implementation isn't available, "community edition" software completes the software stack, so all the software we'll be working with is freely available.

Installation and setup

The specific software requirements to complete this tutorial are as follows:

- The open source MySQL relational database
- The open source MM.MySQL JDBC driver
- The LiDO Community Edition JDO 1.0 implementation from LIBeLIS

The figure below illustrates the particulars of the software stack for this tutorial.


Application
Persistence Manager (LIBeLIS LiDO JDO 1.0 Implementation)
Java-based Data Store Access (MM.MySQL Open Source JDBC Driver)
Data Store (MySQL Open Source Relational Database)

Refer to Code samples and installation requirements on page 2 of the tutorial to download each of the products. You'll find installation instructions at each of the Web sites. As you install the software, be sure to record the locations at which you install them.

In the next several panels, we'll discuss some of the subtleties of setting up and using the software for this tutorial.

MySQL database

The MySQL database, available as a Windows download, is one of the most popular open source databases available. It performs well and it scales well. Upon installation, you'll see an administration tool in the task bar; we use this to create and drop databases. You can access the administration tool in the `bin` directory by running the `winmysqladmin.exe` program.

You can also use the `MySqlManager.exe` program, also in the `bin` directory, as a graphical interface for running SQL statements against databases. It is strongly suggested that you run queries against the databases and tables in this tutorial. We'll discuss table schemas in the next section.

The *only* operation that you have to do directly against the database using the administration client is to create and drop the database that we'll use.

MM.MySQL JDBC drivers

MySQL doesn't provide a JDBC driver directly with the database. You will want to download the open source MM.MySQL JDBC Type IV drivers for MySQL. Once downloaded, keep the drivers in your classpath during development and run time.

The driver class is `org.gjt.mm.mysql.Driver`. You will see this class referenced in each of the programs that access persistent objects.

The JDO persistence manager and factory also need a JDBC URI that references the database (data source) that you are using for persistence. In our case, we will always use the `jdotut` database. The URI is `jdbc:mysql://localhost/jdotut`.

That's the extent of JDBC that you're required to understand for using a relational database from JDO. For the remainder of the tutorial we'll access data and objects in a Java-centric fashion.

LIBeLIS LiDO Community Edition

The final required piece of software is the JDO implementation itself. The LiDO Community Edition software from LIBeLIS is free for developers. Although it does not contain many of the advanced, graphical tools for development, the community edition has everything we need to get through this tutorial. You will have to register at the site to download this software.

After you've successfully downloaded the JDO implementation, ensure that the appropriate .jar files are in your classpath during development and run time. Although each implementation adds options and interfaces that can boost performance or get better behavior from a particular driver, we will be using only the JDO interfaces that are within the original specification. The LIBeLIS JDO implementation surfaces just once in the tutorial. In a call to the `PersistenceManagerFactory`, which is a part of the JDO specification, the proper factory to use is identified as `com.libelis.lido.PersistenceManagerFactory`. Other than that, every bit of code we'll be working with is from the JDO specification's `javax.jdo` package.

We'll use the class enhancer and a schema definition tool. The schema definition tool builds our table schemas for us so that we don't have to learn any table creation SQL or JDBC handling.

Summary

In this section of the tutorial, we've taken a high-level view of the JDO architecture and installed a generic software stack. To review, the software stack used for a typical JDO installation is as follows:

- The Java 2 platform, Standard Edition
- Database software (typically relational)
- JDBC drivers
- JDO tools and drivers

Each of the tools must support the others in the stack. Typically, this won't be a problem, because the Java platform and JDBC act as a buffer between the JDO persistence mechanisms and the data storage platform.

Section 3. A simple example using JDO

Overview

In this section we'll use the software we've installed to build a simple example that consists of addresses that should be persistent. Later in the tutorial, we'll use these same addresses for a more advanced exercise.

The goal of this section is to:

1. Demonstrate the development process with JDO
2. Show how JDO creates an object-oriented persistence mechanism and hides the details of the underlying database access techniques

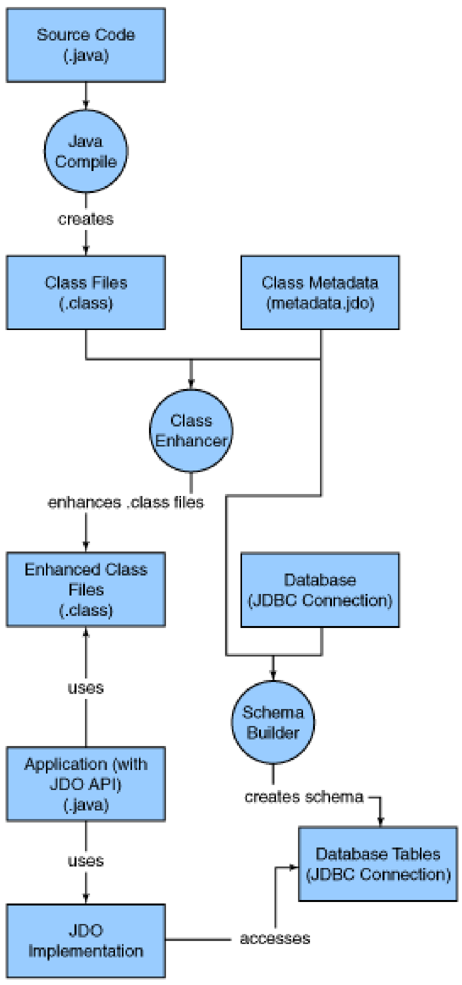
The development process

The development process with JDO is in many ways similar to a development process with EJB CMP, though it is lighter in weight. There are essentially four parts to a JDO development process:

- Creating the objects
- Defining object persistence
- Creating the application
- Deploying the application

Like all persistence mechanisms, the definition and deployment of the objects and application tends to "leak" into the development process, with the actual separation becoming blurred in the final packaging of the application.

Developing an end-to-end application with JDO is much more complex than developing a straightforward Java application. Many of the steps in the process are different from traditional Java development. Furthermore, many of the steps we'll discuss here will later "disappear" into tooling, in the form of either IDEs or build scripts. Regardless of the initial complexity, the benefits of using JDO are huge, particularly when it comes to the natural representation of object-oriented techniques and data source abstraction. Remember this as we go through our first exercise, which may be confusing at times. Things will ease up once you get into the rhythm of the development process.



The flow of development

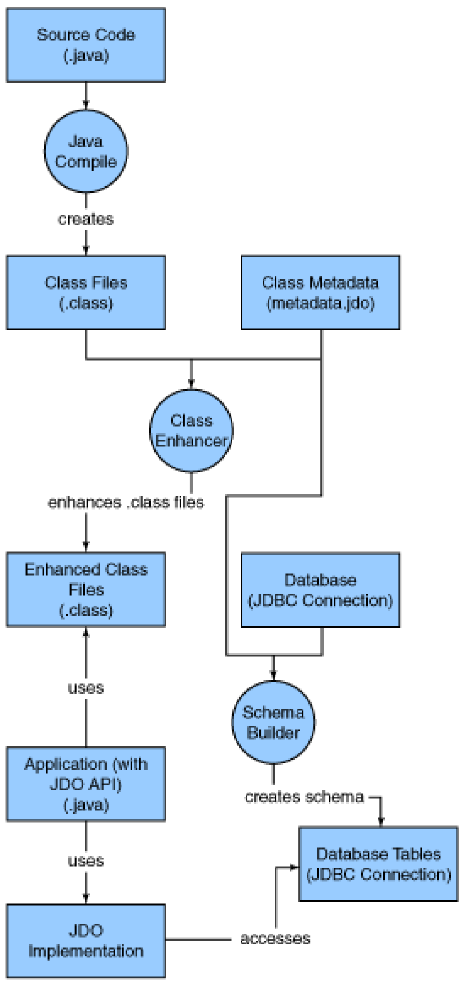
The diagram on the left shows the development process with JDO. You will want to enhance this process with decision points and iteration when you adopt a JDO environment.

The first step is to build our source code (the .java files). Theoretically, the object source we build in this step has no awareness of persistence. For example you can build a set of source files for people (PersonImpl.java), addresses (AddressImpl.java), and departments (DepartmentImpl.java) without knowing that the classes will end up in a database. These source files are turned into .class files with a standard Java compiler. In practice, knowing that we'll be using JDO for object persistence lets us build better performance techniques into the object model.

The next step is determining what classes require persistence. Once this has been determined, the .class file output from the previous steps is enhanced with a *class enhancer*. This step will most likely be carried out by the person who developed the implementation classes, because he or she has essential knowledge of the implementation. To use the class enhancer, you must build a metadata file (in XML format) that identifies which classes require persistence, as well as any additional information about the classes. The .class files are combined with the metadata by the class enhancer. The output is modified .class files.

Class enhancement makes it difficult to integrate the build process with existing IDEs. It is also the most controversial aspect of development with JDO. Modifying .class files is a well-studied art form, and tools have been developed to aid in direct class modification. Nonetheless, it's difficult to get your head around a process that involves creating .class files that don't correspond to the source code. If you use build tools such as Ant this will all be done for you seamlessly, under the covers.

The next step is using the enhanced .class files and the JDO interface to build application objects. Whereas the persistent objects don't use the JDO APIs and implementation, application objects will



use the JDO API to control persistence, transaction, and query mechanisms against persistent objects. The primary point of contact for an application is the `PersistenceManager` that is associated with a particular data source. We'll work with the persistence manager throughout this tutorial.

As we're building the application, we'll also be working on the actual database schema. Typically, the JDO implementation includes a tool to automatically create the database schema from the metadata and enhanced class files created in the previous steps.

Creating an address class

We'll start the first exercise by creating a simple address class. You will note in the code sample that follows that I haven't separated the class interface from the implementation. While it is better programming practice to maintain the separation (even when using JDO), doing so in this case would have added no value, and would have added to the length of the tutorial. I chose to keep the code as brief as possible.

```
package com.stereobeacon.jdo.tutorial;

public class AddressImpl {
        public String addressLine1;
        public String addressLine2;
        public String city;
        public String state;
        public String zip;

        public AddressImpl(String addr1, String addr2, String city, String state, Strin
                addressLine1=addr1;
                addressLine2=addr2;
                this.city=city;
                this.state=state;
                this.zip=zip;
        }
        public String getAddressLine1() {
                return addressLine1;
        }

        public void setAddressLine1(String addressLine1) {
                this.addressLine1 = addressLine1;
        }

        public String getAddressLine2() {
                return addressLine2;
        }

        public void setAddressLine2(String addressLine2) {
                this.addressLine2 = addressLine2;
        }

        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 getZip() {
                return zip;
        }

        public void setZip(String zip) {
                this.zip = zip;
        }

}
```

Notes about the address class

When viewing the source in the previous panel, recall that:

- The address class will later be contained within an employee class.
- The JDO specification separates object implementation from awareness of persistence.
- All attributes should adhere to the JavaBeans specification for get/set methods.

Compile the code using any standard Java compiler. At this point, you will simply have a class implementation with no persistence capabilities. You can use the class like any other Java class; your address information will simply disappear when the JVM exits.

The class is in the package `com.stereobeacon.jdo.tutorial` enclosed in the source download (see Resources on page 30).

Enhancing the address class with persistence

Class enhancement requires two inputs:

- A compiled Java class file
- Metadata about the classes that should be enhanced

Enhancing an original class file results in the file being modified to fit the requirements of the JDO specification, as well as the particular JDO implementation in use. For example, once it has been enhanced the class will implement the interface `PersistentCapable`. All classes that will be persistent must implement this interface. You will notice that in our `AddressImpl` class we left this detail to the class enhancer. A more advanced programmer could implement the `PersistenceCapable` class himself, but at this point we're tying our object implementation to JDO.

The metadata format for describing a class that will be persistent is dictated by the JDO specification, though there is some room for vendor-specific metadata attributes. A very simple metadata file can be used for the `AddressImpl` class, as shown in the next panel.

Class metadata

```
<?xml version="1.0"?>
<!DOCTYPE jdo SYSTEM "jdo.dtd">
<jdo>
   <package name="com.stereobeacon.jdo.tutorial">
     <class name="AddressImpl">
         <field name="addressLine1"/>
         <field name="addressLine2"/>
         <field name="city"/>
         <field name="state"/>
         <field name="zip"/>
     </class>
   </package>
</jdo>
```

Note that the metadata (an XML file) includes a DTD. Also note that the `package` tag allows classes from one package to be grouped into one body. The `class` tag identifies classes within the package that should be persistent. Finally, each field that will be persistent within the class is identified. In this case, we've only used the minimum metadata requirements to get `AddressImpl` to be persistent. The JDO specification provides many additional options for classes, such as the ability to specify a primary key and whether fields can have null values.

Running the enhancer

To run the class enhancer supplied with LiDO, you'll have to point the enhancer at the metadata file. The enhancer will then find the .class files that require modification. The following command will point the enhancer in the right direction:

```
java com.libelis.lido.Enhance -metadata metadata.jdo
```

Note the problem created with this step. Without tight coupling with an IDE, your class files will now be different from your source files. Many IDEs automatically regenerate class files, which would wipe out any changes made by the enhancer. Hopefully, the next generation of IDEs will have JDO enhancements built into the tools.

In the accompanying source code, the `metadata.jdo` file contains all of the class definitions used throughout the tutorial. This command is contained in the enhance.bat file found in the source code download.

Building the database schema to support the class

With some implementations of JDO, you may need to build a database to support your persistent objects. More likely, you will be given a tool that automates the process. Our edition of LiDO provides a command-line tool to build the database schema. Simply reuse the metadata we built for the previous step, and give the name of the database driver (in this case the MM.MySQL driver) and the URI that identifies the database to the driver. As previously mentioned, you will need to create the `jdotut` database prior to running this command. Databases can be created within the MySQL administration console.

Run the following command to create the database schema:

```
java com.libelis.lido.ds.jdbc.DefineSchema
         -driver org.gjt.mm.mysql.Driver
         -database jdbc:mysql://localhost/jdotut
         -metadata metadata.jdo
```

After running this command, you will have tables that store any persistent addresses we have made. This command is contained in the batch file defineschema.bat in the source code.

Table definition

Following is the table schema generated by the LiDO schema tool for our address class. Study it carefully before moving on to the description.

Name	**Type**	**Key**
LIDOID	bigint(20)	Yes
ivAddressLine1	varchar(255)	No
ivAddressLine2	varchar(255)	No
ivCity	varchar(255)	No
ivState	varchar(255)	No
ivZip	varchar(255)	No

The first thing you will notice is that the column names match our instance variable names, except for one. The first column, `LIDOID`, was added by the tool because we did not have a unique key defined in the metadata about the object. As an object or application programmer, this field will not be accessible, though LiDO will use the field to track instances.

Table creation is outside of the scope of the JDO implementation so you will find that different tools use different mechanisms, table name standards, and resulting schemas.

Building an application

In the application that follows, we will create an instance of class `Address`. This instance will be made persistent, and the application will exit. You will notice several distinct parts to the program, as follows:

- JDO setup
- Transaction manipulation
- Object life cycle
- Persistence interface to JDO

Here is the application code:

```
package com.stereobeacon.jdo.tutorial;

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

public class Test1 {
    public static final String DBURL = "jdbc:mysql://localhost/jdotut";
    public static final String DBDRIVER = "org.gjt.mm.mysql.Driver";

    public static void main(String[] args) {
       try {
          /*
           * SETUP JDO
           */
          // first create and set up a persistence manager
          // factory
          PersistenceManagerFactory pmf =
             (PersistenceManagerFactory) Class
                .forName("com.libelis.lido.PersistenceManagerFactory")
                .newInstance();
         pmf.setConnectionURL(DBURL);
         pmf.setConnectionDriverName(DBDRIVER);

         // now retrieve the persistence manager associated
         // with the parameters setup in the factory
         PersistenceManager pm = pmf.getPersistenceManager();

          /*
           * START A TRANSACTION
           */
          // retrieve the current transaction
         Transaction t = pm.currentTransaction();
          // start a new transaction
         t.begin();

          /*
```

```
           * CREATE AN OBJECT INSTANCE
           */
          AddressImpl a =
            new AddressImpl("7700 Technology Way", "", "Denver", "CO", "80237");

          /*
           * MAKE THE OBJECT PERSISTENT
           */
         pm.makePersistent(a);

          /*
           * COMMIT THE OBJECT TO STORAGE
           */

         // commit the transaction, all changes made between
         // the begin and commit are set to the db
         t.commit();

          /*
           * CLEANUP AND EXIT
           */
          pm.close();
        } catch (Exception e) {
          e.printStackTrace();
        }
    }
}
```

Notes about the application code

Now, let's look at the details of each step of building the application code.

1. **Setup JDO**: JDO leverages class factories throughout the architecture. Setting up a persistence manager factory is how the underlying JDO implementation determines which `PersistenceManager` should be used for a particular data source. Once the factory has the relevant information, you retrieve the `PersistenceManager` from the factory with the `getPersistenceManager` call.

2. **Start a transaction**: Objects in JDO are first-class transaction participants. This implies that we will need to make changes to objects within a transaction. (You will want to consult the JDO specification regarding transactions since there are a variety of supported options.) We call the persistence manager to obtain a reference to the current transaction and, in this case, we `begin` a new transaction.

3. **Create an object instance**: Object creation occurs just like it normally would. You should remember that the actual run-time call occurs against the enhanced class.

4. **Make the object persistent**: Assuming the class enhancer was run against them, the `makePersistent` method call makes whatever objects are passed to it persistent, though they won't be visible until after the transaction commit.

5. **Commit the object to storage**: Committing the object to storage unlocks the object instance and makes it available to other processes that may be looking at the same type of object instances.

6. **Clean up and exit**: Close the persistence manager when you're done using it.

Running the application

It's been a trek, but we're ready to run the application. In this section we'll only run the application that builds the first instance, then we'll check out the changes to the table. In a Windows operating environment, a batch program like the one below will suffice:

```
set CLASSPATH=C:\java\mm.mysql-2.0.14\mm.mysql-2.0.14-bin.jar;
set CLASSPATH=C:\java\mm.mysql-2.0.14\lib\jdbc2_0-stdext.jar;%CLASSPATH%
set CLASSPATH=C:\java\mm.mysql-2.0.14\lib\jta-spec1_0_1.jar;%CLASSPATH%
set CLASSPATH=c:\java\LiDO\lib\j2ee.jar;%CLASSPATH%
set CLASSPATH=c:\java\LiDO\lib\lido-rt.jar;%CLASSPATH%
set CLASSPATH=c:\java\LiDO\lib\lido-dev.jar;%CLASSPATH%
set CLASSPATH=c:\java\LiDO\lib\lido-rdb.jar;%CLASSPATH%
set CLASSPATH=c:\java\LIDO\bin;%CLASSPATH%
set CLASSPATH=.;%CLASSPATH%

java com.stereobeacon.jdo.tutorial.Test1
```

Note the inclusion of the JDBC drivers, the Java 2 Enterprise Edition jar file (which was included with the JDO implementation), the LiDO jar files, and the bin directory from LiDO. The bin directory is included because LiDO needs the license for the product in the CLASSPATH, and the license is held in the bin directory.

Running the test will yield no visible output. Running the test multiple times will add the same record to the underlying database several times. The reason for this behavior (rather than a duplicate key exception) is that we placed the responsibility for generating a unique key with LiDO. Another option would have been to define a primary key in the metadata about the classes.

Once you've fixed up any CLASSPATH issues, run the addressCreate.bat file to start the sample.

Viewing the results

Using the MySqlManager tool shipped with MySQL, we can run a query against the `jdotut.csjt_addressimpl` table to return all of the addresses that exist in the table. The simple SQL select statement that we will use is:

```
select * from jdotut.csjt_addressimpl;
```

The result of the SQL call is:

```
LIDOID ivAddressLine1 ivAddressLine2 ivCity ivState ivZip
------ -------------- -------------- ------ ------- -----
1           7700  Technology Way Denver      CO 80237

(1 row(s) affected)
```

Summary

In this section, we've learned about the JDO development process. Starting with a conceptual discussion and following up with an example exercise, we've walked through each step of the development process. To review, the JDO development process includes the following steps:

1. Defining classes
2. Building a metadata file that contains information about class persistence
3. Running a class enhancer that uses the metadata to generate enhanced class bytecodes
4. Running a schema definition tool that uses the metadata and the enhanced class bytecodes to build a table definition in a database
5. Building an application that leverages the JDO interfaces to make classes persistent

Because the process is rather involved, working with JDO can appear more convoluted than working with JDBC. This is mostly due to class enhancement, which doesn't take place in a typical Java development environment. On the other hand, the JDO process is much more lightweight than that of typical EJB persistence mechanisms. We can complete the entire development process from a command line with no servers or bulky deployment scenarios. Furthermore, JDO doesn't require us to know anything about JDBC, beyond what we need to know to point the driver and URI to the database used for persistence.

In the next two sections, we'll scale the example up a bit. As we do so, you should start to see the benefits of being able to stay focused on the object paradigm.

Section 4. A more complex example

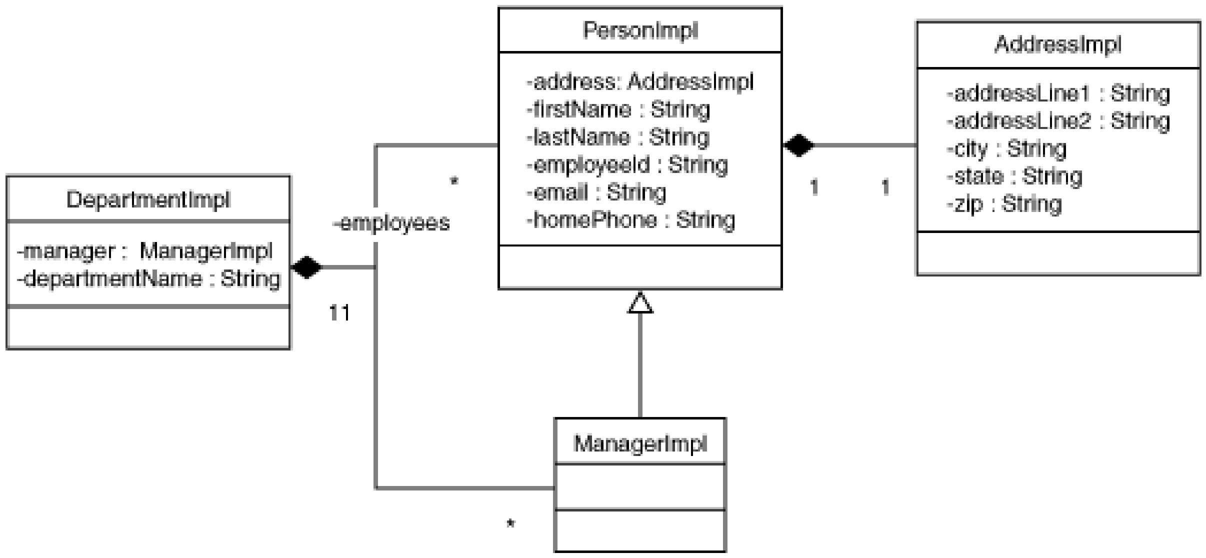
Overview

In this section we'll take our application to the next level, throwing a simple inheritance example, containment, and a few extra objects into the code. The only major differences between this exercise and the last one is in the metadata. Our development process will be exactly the same as in the previous section:

1. Compile the classes.
2. Enhance the classes with an updated metadata file.
3. Build the table schema.
4. Compile the application.
5. Run the application.

The class diagrams

Notice the addition of several classes in the UML diagram below. The `AddressImpl` itself remains unchanged.



The classes we'll focus on for this example are as follows:

- `DepartmentImpl` represents a department of employees with a single manager. Notice there is an instance variable, `employees`, of type `Collection`. It will contain people of the type `PersonImpl`.
- `PersonImpl` represents a person. A person contains an `AddressImpl`.
- `ManagerImpl` represents a manager and is a subclass of `PersonImpl`.

Metadata updates

In general, a metadata update is a matter of simply adding additional classes to the metadata, as shown here:

```
<?xml version="1.0"?>
<!DOCTYPE jdo SYSTEM "jdo.dtd">
<jdo>
   <package name="com.stereobeacon.jdo.tutorial">
     <class name="AddressImpl">
         <field name="addressLine1"/>
         <field name="addressLine2"/>
         <field name="city"/>
         <field name="state"/>
         <field name="zip"/>
     </class>

     <class name="PersonImpl">
         <field name="employeeId" primary-key="true"/>
         <field name="address"/>
         <field name="email"/>
         <field name="firstName"/>
         <field name="lastName"/>
         <field name="homePhone"/>
     </class>

     <class name="ManagerImpl"
               persistence-capable-superclass=
                 "com.stereobeacon.jdo.tutorial.PersonImpl">

     </class>

     <class name="DepartmentImpl">
         <field name="departmentName"/>
         <field name="employees">
            <collection element-type="com.stereobeacon.jdo.tutorial.PersonImpl"/>
         </field>
         <field name="manager"/>
     </class>
   </package>
</jdo>
```

Notes about the code

You should pay attention to the use of subclassing in the code in the previous panel. In the declaration of the `ManagerImpl` class, note the additional tag attribute `persistence-capable-superclass`, which identifies the superclass that is also persistence capable. The identification of collections is also interesting. Contained in the field declaration of `employees` within `DepartmentImpl` is additional information about the collection (recall that `employees` is of type `Collection`). The `element-type` attribute identifies the type of the classes contained within the generic collection.

We're taking a very simplistic view of metadata in this tutorial, showing only the attributes that are necessary to get up and running. In the JDO specification you'll find a section entitled "XML Metadata"; read this section to learn more about controlling the data in your application.

A main program

You should find no surprises in the following portion of the driver program. The only changes are in the construction of the objects.

```
// start a new transaction
t.begin();
{
         // create 3 new addresses
         AddressImpl a1 =
                 new AddressImpl("1111 Avalanche Drive", "", "Denver", "CO", "80237");

         AddressImpl a2 =
                 new AddressImpl("2222 Twins St.", "", "Rochester", "MN", "55901");
         AddressImpl a3 =
                 new AddressImpl("3333 Skunk St.", "", "Manitowoc", "WI", "11111");

         // create 2 employees and 1 manager
         PersonImpl p1 =
                 new PersonImpl(a1,"Paul", "Monday", "11",
                         "pmonday@attbi.com", "303-555-1212");
         PersonImpl p2 =
                 new PersonImpl(a2,"Anson", "Williams", "12",
                         "anson@isaak.com", "777-555-1212");
         ManagerImpl m1 =
                 new ManagerImpl(a3,"Fox", "Mulder", "21",
                         "abducted@fox.net", "222-555-1212");

         Vector v = new Vector(2);
         v.add(p1);
         v.add(p2);
         DepartmentImpl dept = new DepartmentImpl(
                 v, m1, "ARCHITECTURE");

         pm.makePersistent(dept);
}

// commit the transaction, all changes made between
// the begin and commit are set to the db
t.commit();

// start a new transaction
t.begin();
{
         // create 3 new addresses
         AddressImpl a1 =
                 new AddressImpl("4444 Lombardi St.", "", "Green Bay", "WI", "45001");
         AddressImpl a2 =
                 new AddressImpl("55 Woods Way", "", "Miami", "FL", "12001");
         AddressImpl a3 =
                 new AddressImpl("79 Borg Avenue", "", "Voyager", "MN", "55890");

         // create 2 employees and 1 manager
         PersonImpl p1 =
                 new PersonImpl(a1,"Arty", "Garhunkle", "19",
                         "troubled@bridge.net", "777-555-1212");
         PersonImpl p2 =
                 new PersonImpl(a2,"ElTigre", "Madera", "80",
                         "par@forcourse.com", "250-555-1212");
         ManagerImpl m1 =
                new ManagerImpl(a3,"Jerry", "Smith", "85",
                        "jerry@public.edu", "313-555-1212");

        Vector v = new Vector(2);
        v.add(p1);
        v.add(p2);

        DepartmentImpl dept = new DepartmentImpl(
                v, m1, "QUAKE");

        pm.makePersistent(dept);
}

// commit the transaction, all changes made between
// the begin and commit are set to the db
t.commit();
```

Notes, and running the program

In the `makePersistent` method call, only the department object is passed in, yet all of the nested objects will be properly persisted. Pretty cool, huh?

At this point, you should realize that there are other signatures for `makePersistent`. One takes an array of objects and the other takes a collection of objects. These additional signatures allow you to persist more than one object in a single method call. (You'll get better performance with the group call.)

Before running this example, make sure you've done the following:

- Enhanced the new .class files with enhance.bat
- Dropped the `jdotut` database and re-created it
- Run the defineschema.bat command to build the new schema

Finally, run the populateDepartments.bat command to activate the above program. You can run printDepartments.bat at any time to see the contents of our departments.

Summary

In this section we added some complexity to the example code, which allowed us to work with some additional attributes of the class metadata. We also wrote a longer, slightly more complex driver to populate the database.

The more robust set of data will be useful for the next section, where we'll learn about querying and manipulating persistent data.

Section 5. Managing and querying data objects

Overview

So far, we have spent a lot of time building and discussing the setup and creation of objects. In this section we'll learn about manipulating the objects we've created. Three high-level tasks are key to this section:

1. Retrieving object instances
2. Changing persistent data
3. Deleting persistent data

Retrieving object instances will take up the bulk of this section. There are several mechanisms to deal with object retrieval. We will quickly discuss one, the *extent*, then spend more time on the JDO query language (JDOQL). Next, we'll look at how to change data that is already persistent. And, finally, we'll remove some objects to show you how to shrink your data.

Extents

An *extent* is a JDO construct that is leveraged both for performance and as a basis for JDO object queries. An extent represents all of the instances of a class or a class and its subclasses. Extents could be thought of as collections, but with some room for optimization. A JDO implementation can implement an extent such that only batches of returned objects are actually immediately available in memory. In this case, the remaining objects become populated by the JDO implementation one by one, or in groups, as the iterator encounters objects that are not available in memory. The ability to lazily load objects is an improvement over collections, which can require the JDO provider to load thousands of objects to represent a table.

You will also notice, in later panels, that extents form the basis of most object queries. An extent is passed into a query as a candidate collection of objects for the query to work on.

An extent example

Below is a portion of a program that will retrieve all instances of `PersonImpl` as well as its subclass `ManagerImpl`.

```
{
      System.out.println("Retrieving all people");
      // Retrieving all departments
      Extent ext = pm.getExtent(PersonImpl.class, true);
      Iterator i = ext.iterator();
      while (i.hasNext()) {
         PersonImpl person = (PersonImpl) i.next();
         System.out.println(
            "Person:  "
            + person.getLastName()
            + ", "
            + person.getFirstName());
      }
      ext.closeAll();
}
```

Note that an extent is retrieved through the persistence manager. The parameters are as follows:

1. The class of instances that the extent is based on
2. Whether subclasses should be included

Finally, note that the extent should be closed. This closes all iterators over an extent. After this point, all iterators will return `false` from their `hasNext` method.

Upon retrieval of the extent, we get an iterator (just like a collection) and iterate through the contents. The contents of the collection that are printed are all of the people that were created, as well as all of the managers, since manager is a subclass of people.

The code for this class is in `com.stereobeacon.jdo.tutorial.ExtentTest`. You can run it using the extentTest.bat file. In the program, there is also an example showing the retrieval of the `DepartmentImpl` classes.

JDOQL

You can consider the JDO Query Language (JDOQL) in two parts: the API to manage queries, and the query language itself. The API to create queries is on the JDO `PersistenceManager`. A variety of options for query creation are specified at create time, such as the elements the query will execute against (a collection, class, or extent) and a query *filter*, which is similar to a `select` clause in SQL.

Once a query is created, most often with a filter limiting the number of objects returned by the query, the query itself can be managed through the `Query` class API. The query API allows the setting of variables and parameters for the filter (so you don't have to construct a new filter string for every query), options for caching, and options for ordering the returned objects.

We'll look at three queries, focusing primarily on the filter, although we'll also examine the basic elements of how to construct and execute a query using JDOQL.

The code for the queries that follow is in the `com.stereobeacon.jdo.tutorial.RunQueries` class, which you can run using the runQueries.bat file.

A simple object query

We'll start with a simple query to select one of our two departments. Because our examples are simple, it would often be just as easy to load the extent and iterate through the results. Despite this, it's a good idea to know how to work with JDOQL. Queries pay off when table sizes are large, as is the case with most enterprise applications.

```
         Extent ext = pm.getExtent(DepartmentImpl.class, false);
         String filter = "departmentName==name";
         Query q = pm.newQuery(ext, filter);
         String parameter = "String name";
         q.declareParameters(parameter);
         Collection c = (Collection)q.execute("ARCHITECTURE");
```

There are three distinct steps to the above query:

1. First, we create the query with the persistence manager, `pm`, by passing it an extent to operate on and a filter. In the filter, note that the `departmentName` correlates to an instance variable in the class `DepartmentImpl`, which also happens to be the extent that the query will execute against. Next, in the filter, note that we have a variable that does not correlate to an instance variable, `name`. The `name` variable in the filter will be substituted at query execution time with whatever department we want to find. The query is created with `pm.newQuery(ext, filter)`, but it has not retrieved instances at this point.

2. Next, we set up the query object itself with any additional settings we want. In this case, we have to tell it that `name` is a `String`. We do this by passing a string to the `declareParameters` method call on the query. For each parameter listed, the `execute` method on the query will require that a corresponding value be passed in. Multiple parameters are denoted in the parameter string by being delimited by a comma (,).

3. Finally, the query is executed with the `execute` method. Queries return instances of `Collection`, which we then iterate through for information.

A multiple-parameter query

This example presents two critical differences from the previous one:

- No extent is used as the basis for the query; rather, a class, `PersonImpl`, is used as the basis for the query.

- Multiple parameters are used to search for a first name and last name of an employee.

```
String empFilter = "lastName==last & firstName==first";
String empParms = "String last, String first";
Query empQuery = pm.newQuery(
         PersonImpl.class,
         architectureDepartment.getEmployees(),
         empFilter);
empQuery.declareParameters(empParms);
Collection empCollection = (Collection)empQuery.execute(
                                 "Williams", "Anson");
```

First, notice the query syntax. We will be searching for employees where the last name and first name are equal to parameters that are passed in. The query syntax is defined in the JDO specification and loosely follows SQL syntax. The two parameters, `last` and `first` are defined to be strings.

A new query is created by retrieving the collection of employees from the previously located architecture department. The first parameter `PersonImpl.class` is required as a way to denote the types of objects in the collection. The second parameter is the candidate collection, and the third parameter is the query filter itself. As with a single-parameter query, the next step is to declare the parameters of the filter with the `declareParameters` method on the query object.

When the query has been executed, the two parameters are passed to the query. Notice that, rather than bundling the parameters in an object array, they are passed consecutively. Up to three parameters can be passed consecutively for convenience. Any more than that should be bundled as an array.

A contained-object query

The final query example is what is known as a *containment* query. Basically, we want to search all of the departments for a particular person (rather than the last query, which searched a single department for a particular person).

To build the containment query, we use a second query concept (the first being parameters) known as a *query variable*. The variable helps constrain the results of a query. In the example below, the variable is used to specify that all of the `PersonImpls` with the last name equal to the query parameter should be selected. Basically, the variable allows us to further constrain the employee collection in the department.

```
Extent deptExtent = pm.getExtent(DepartmentImpl.class, false);
String personVar =
         "com.stereobeacon.jdo.tutorial.PersonImpl person";
String containFilter =
         "employees.contains(person) & person.lastName==name";
String nameParam = "String name";
Query personQuery = pm.newQuery(deptExtent, containFilter);
q.declareParameters(nameParam);
q.declareVariables(personVar);

Collection personColl =
         (Collection) personQuery.execute("Williams");
```

Notice that variables are declared in essentially the same way as parameters. Variables do not have associated parameters to pass on the query execute clause because the variable is an element that is used within the filter, not for parametric setting of filter information.

Finally, note that this query and the multiple object query *did not* work with LiDO Version 1.0. The LIBeLIS team is aware of this bug and working to fix it.

Changing objects and removing instances

In general, creating, changing, and deleting object instances does not require special object semantics. On the other hand, special steps must be taken, as you write the application, to ensure creations, changes, and deletions get saved to the underlying persistence mechanism.

You have already seen the `makePersistent` method for creating an object instance. The code in the next panel uses the parallel `deletePersistent` method on the persistence manager to remove the first employee from a department.

When an object is modified through the set methods that parallel the instance variables (such as `setEmployees`) that object instance is "dirty." As a result, a mechanism is triggered to re-save the object data on the next transaction commit.

Note that the `architectureDepartment` instance variable was populated prior to entering the code block in the next panel.

Removing an object instance

```
Collection employees = architectureDepartment.getEmployees();
pm.retrieveAll(employees);
// remove the first element
PersonImpl removedEmployee = null;
if(!employees.isEmpty()) {
         Iterator employIt = employees.iterator();
         removedEmployee = (PersonImpl)employIt.next();
         employees.remove(removedEmployee);
         architectureDepartment.setEmployees(employees);
         pm.deletePersistent(removedEmployee);
}
```

In the above code, we first retrieve the collection of employees from a particular architecture department. Next, we call the `retrieveAll` method on the persistence manager. This loads the entire collection into memory (an operation that could be a problem if table sizes were too large).

To successfully remove an employee we must:

- Remove the person from the collection within the department
- Delete the persistent representation of the person from the database

If we just delete the persistence representation from the table, the department's collection will receive a null pointer exception when it tries to retrieve it. On the other hand, if we only remove the person from the collection in the department, we'll end up with a person that isn't contained by any departments and may be left as "garbage" in an ever-increasing collection of database records.

Object ownership

The removal scenario brings up the concept of *object ownership*. Ownership is a very difficult concept to model and implement in a system that handles persistence. Unfortunately, JDO does not resolve the object model for you; you'll still have to think about how your application will manage the object model. Some constructs in JDO do help with these scenarios, but you will frequently have to modify the object model as well as the "rules" that an application adheres to for leveraging the persistent data. You can start becoming an expert in your scenario by spending several hours with the JDO specification as well as your chosen JDO vendor implementation.

The code for the delete and modify example is in the `com.stereobeacon.jdo.tutorial.DeleteObject` class, which you can run using the deleteObject.bat file.

Summary

In this section, we've explored mechanisms for managing the life cycle of persistent objects. We started with using an *extent*, which represents large collections of objects and forms the basis set for most queries. We then discussed a variety of query techniques for retrieving specific collections of instances, all of which were based in the JDOQL. Finally, we discussed how to change an object and remove objects that have already been stored in a database.

With all of the examples in this section (and those contained in the source code), we have only scratched the surface of some of the versatility of the JDO Query Language (JDOQL) and different ways to manage memory and CPU performance when selecting objects. While this section lays some groundwork for querying, it is important that you spend time with the JDO specification and vendor documentation. Not only can vendors extend JDO metadata to provide clues for improving performance, but some JDO implementations may be better than others at getting performance out of individual databases.

Section 6. Wrapup and resources

The future of JDO

JDO is an excellent first version of a specification, and as an object persistence mechanism it fills a gaping hole. On the other hand, JDO does have considerable competition to contend with in this area. The SanFrancisco Application Component Framework was built with an object persistence layer over five years ago (although SanFrancisco Classic is no longer available); EJB technology is holding ground with entity beans and CMP; and, of course, XML data binding and Java serialization are worthy contenders. Each of these mechanisms offers its own admirable solution to the problem of object persistence, but none of the mechanisms truly fit into the object-oriented programming paradigm. In the end, JDBC often wins the bid for persistence because it is unabashedly relational, just like the data that is typically stored in databases.

JDO will prove to be an effective mechanism for solving many of the challenges that today's object-oriented programmers have been struggling with, but it won't solve them all. Using JDO entails making subtle changes to how we program, such as enforcing object ownership rules, adding pointers to object owners from contained objects, and making other changes to enhance the performance and navigability of object maps. On the application side, we'll have to ensure we don't leave objects in databases, and also make sure we're using the JDO options that will get optimum performance for a particular user interface.

With all of those downsides, I'm still sold on JDO. The technology is as important as entity beans, and it weighs a lot less. A lightweight alternative to entity beans is something EJB programmers have been screaming for, and a mechanism to integrate JDO and EJB could make this happen (although a detailed discussion is beyond this tutorial). Perhaps most important of all is the fact that JDO is more natural to an object-oriented programmer than JDBC, and it is a better fit for the enterprise than Java serialization or XML data binding.

Whether or not JDO is ready to be used for production-level application development, it is worthy of consideration. Hopefully, having completed this tutorial, you will make it one of your options the next time you're evaluating persistence mechanisms for your object paradigm.

Tutorial summary

In this tutorial, we've explored the essential facets of JDO, including:

- The JDO architecture and environment
- The development process with JDO
- Examples that scaled to several classes
- Simple and complex queries
- The creation, selection, alteration, and removal of object instances

While this has been by no means a comprehensive exploration of the JDO specification, we have worked through the high points of the specification. Upon completing the tutorial, you should feel you have a working knowledge of JDO.

To further pursue your interest in JDO, see Resources on page 30 . There you will find a link to the original JDO specification, as well links to download some sample drivers, like the one from LIBeLIS.

JDO is a very new technology, and implementations are just coming out that adhere to the 1.0 specification. Several times in writing the tutorial I was caught relying on the JDO *proposed final draft* rather than on a final draft. As a result, there have been many changes along the way. Please be sure to take a look at the latest draft specification and implementation results.

In closing, I would like thank the very helpful and responsive people at LIBeLIS for their assistance and virtually immediate support throughout the writing of this tutorial.

Resources

Downloads

- Download jdo-source.zip, the source file for all the examples used in this tutorial.
- The *Java 2 platform, Standard Edition* (http://java.sun.com/j2se/) is available from Sun Microsystems.
- The *Java Data Objects specification, information, and reference implementation page* (http://access1.sun.com/jdo/) contains the most current information about JDO.
- We used the *LIBeLIS Community Edition JDO 1.0* (http://www.libelis.com) as the JDO implementation (site registration is required for download).
- The *MySQL open source relational database* (http://www.mysql.org) is used for data persistence in the tutorial.
- The *MM.MySQL open source JDBC drivers* (http://mmmysql.sourceforge.net/) are used for Java access to the MySQL database.
- Download *Apache Ant* (http://jakarta.apache.org/ant/), now available in version 1.5 beta 2, from the Jakarta Project Web site.

Articles and tutorials

- Dennis Sosnoski profiles Castor, another data-binding technology, in his article "*XML in Java: Data binding with Castor*" (*developerWorks*, April 2002, http://www-106.ibm.com/developerworks/java/library/x-bindcastor/index.html)
- The tutorial "*Data binding with JAXB*" by Daniel Steinberg (*developerWorks*, January 2002, http://www-106.ibm.com/developerworks/education/r-xjaxb.html) will get you started with this data-binding technology from Sun Microsystems.
- The tutorial "*Building Web-based applications with JDBC* " by Robert Brunner

developerWorks, December 2001, http://www-106.ibm.com/developerworks/education/r-jdbcw.html) is a good introduction to the fundamentals of programming with JDBC.

- For more advanced JDBC operations, see Robert Brunner's second tutorial in his JDBC series "*Advanced database operations with JDBC*" (*developerWorks*, November 2001, http://www-106.ibm.com/developerworks/education/r-jdbc3.html).
- Rick Hightower examines Container Managed Persistence in his two-part tutorial series: "An introduction to Container Managed Persistence and Relationships," *Part 1* (http://www-106.ibm.com/developerworks/education/r-wscomp.html) and *Part 2* (*developerWorks*, March 2002, http://www-106.ibm.com/developerworks/education/r-wscomp2.html).
- You'll find consistently maintained, up-to-date information about Java Data Objects at *JDO Central* (http://www.jdocentral.com).
- Keep your eyes open for *Java Data Objects* by Robin Roos (http://www.ogilviepartners.com/JdoBook.html), which is forthcoming from Addison-Wesley (August 2002).

Additional resources

- You'll find hundreds of articles about every aspect of Java programming in the *developerWorks* *Java technology zone* (http://www-106.ibm.com/developerworks/java/).
- See the *developerWorks* Java technology zone *tutorials page* (http://www-105.ibm.com/developerworks/education.nsf/dw/java-onlinecourse-bytitle?OpenDocument&Co for a complete listing of more free tutorials from *developerWorks*.
- IBM research teams around the world are constantly at work developing and researching new technologies. Keep an eye on the *IBM research homepage* (http://www.research.ibm.com/) for the latest discoveries in information technology.

Feedback

Please send us your feedback on this tutorial. We look forward to hearing from you!

Colophon

This tutorial was written entirely in XML, using the developerWorks Toot-O-Matic tutorial generator. The open source Toot-O-Matic tool is an XSLT stylesheet and several XSLT extension functions that convert an XML file into a number of HTML pages, a zip file, JPEG heading graphics, and two PDF files. Our ability to generate multiple text and binary formats from a single source file illustrates the power and flexibility of XML. (It also saves our production team a great deal of time and effort.)

You can get the source code for the Toot-O-Matic at www6.software.ibm.com/dl/devworks/dw-tootomatic-p. The tutorial Building tutorials with the Toot-O-Matic demonstrates how to use the Toot-O-Matic to create your own tutorials. developerWorks also hosts a forum devoted to the Toot-O-Matic; it's available at www-105.ibm.com/developerworks/xml_df.nsf/AllViewTemplate?OpenForm&RestrictToCategory=11. We'd love to know what you think about the tool.