# ObjectDB Developer's Guide

# Table of Contents

# Preface

Welcome to ObjectDB for Java/JDO Developer's Guide. Here you can learn how to develop database applications using ObjectDB and JDO (Java Data Objects), the revolutionary database programming technology by Sun Microsystems. The main purpose of this guide is to make you productive with ObjectDB and JDO in a very short time.

## Organization of this Guide

The first two chapters introduce ObjectDB:

Chapter 1 - About ObjectDB

Describes ObjectDB main features and editions.

Chapter 2 - A Quick Tour

Demonstrates basic database programming using ObjectDB.

The next chapters contain detailed instructions on programming with ObjectDB and JDO:

Chapter 3 - Persistent Classes

Explains what a persistence capable class is and which types are supported by JDO.

Chapter 4 - JDO Metadata

Shows how to define JDO metadata for persistence capable classes.

Chapter 5 – Database Connections

Shows how to use database connections and transactions in JDO.

Chapter 6 - Persistent Objects

Shows how to store, retrieve, update and delete database objects.

Chapter 7 - JDOQL Queries

Describes JDOQL, the JDO Query Language.

The last two chapters are dedicated to ObjectDB's tools:

Chapter 8 - ObjectDB Server

Explains how to manage an ObjectDB database server.

Chapter 9 - ObjectDB Explorer

Explains how to view and manipulate database files in the database explorer.

## Prerequisite Knowledge

A prior knowledge of database programming (SQL, JDBC or JDO) is not required in order to follow this guide, but a strong background and understanding of the Java language is essential.

## Additional Reading and Resources

This guide focuses mainly on practical issues in order to make the reader productive in a short time. After reading this guide you may want to extend your knowledge of JDO, by reading the JDO specification or a book on JDO. You can find references and links to books on JDO, the JDO specification and other JDO related resources at:

  http://www.objectdb.com/database/jdo/jdo-links/.

## Feedback

We would appreciate any comment or suggestion regarding this manual.

Please send your comments or questions to support@objectdb.com.

# Chapter 1.  About ObjectDB

ObjectDB for Java/JDO is a powerful Object Database Management System (ODBMS) written entirely in Java. It can efficiently handle a wide range of database sizes, from a few KBs to hundreds of GBs. It has a small footprint, high performance, and a flexible architecture (1-Tier, 2-Tier or N-Tier).

This introductory chapter contains the following sections:

## 1.1.  Why ObjectDB?

There are many benefits to using ObjectDB. This section describes some of them.

### ObjectDB is an Object Database

Developing object oriented applications with ObjectDB is easier and much more effective because the content of the database is application objects. Whereas, working with a Relational Database Management Systems (RDBMS) is difficult because developers must deal with tables, records and SQL in addition to the application's classes and objects.

### ObjectDB is JDO Compliant

ObjectDB is compliant with the JDO (Java Data Objects) standard, developed by Sun. Therefore, applications that use ObjectDB are not tied to ObjectDB. Switching to another JDO compliant database (for example, as a result of special customer needs or requests) can be done without modifying code. The JDO API is available today for most relational database systems (RDBMS), e.g. Oracle, IBM DB2 and Microsoft SQL Server. In addition, by using JDO you are backed up by a large community that's expanding every day. JDO already has websites, a forum, a mailing list, consulting and training companies, and many books dedicated to it.

### ObjectDB is Easy to Use

You can be productive with ObjectDB in a very short time, because it is very easy to use. In fact, it is the easiest JDO implementation. Most other JDO implementations map classes and objects to tables and records in an RDBMS. Using such mapping makes it possible to write clean Java code, but still requires some familiarity with relational database concepts and the tables and columns defined for an application in order to perform the necessary mappings and

maintain them throughout the life of a system. With ObjectDB, you can forget relational databases, tables, records, fields, SQL, JDBC and drivers, and focus your attention on the application's Java classes exclusively.

### ObjectDB is Very Portable

ObjectDB is written entirely in Java. Therefore, it can run on any environment that supports Java. You can easily move your application to other platforms (Windows, Unix, Macintosh and almost any other platform), simply by taking the ObjectDB and JDO jar files with you. You can even run your application on small devices like PocketPC (if Java is installed), because ObjectDB has a very small footprint (about 300KB at runtime).

### ObjectDB is Very Fast

ObjectDB has very good performance. It can store and retrieve thousands of objects per second. Even queries on millions of objects are very fast, when proper indexes are defined.

### ObjectDB has a Database Explorer

The ObjectDB database explorer, which is provided with every ObjectDB product, is a sophisticated tool that enables browsing, querying, editing and the construction of Java objects in a database, visually, without writing a single line of code. Many object databases do not have a visual browsing tool at all, or have a very limited one. Keep in mind that database programming without the ability to explore, query and edit the database content visually, is like developing software without a debugger.

## 1.2.  ObjectDB Operating Modes

ObjectDB for Java/JDO can operate in one of two modes: embedded mode or client server mode.

### Client-Server Mode

In client-server mode, an ObjectDB server is running on a separate JVM (in a separate process). Applications running on different JVM's communicate with the database server using TCP/IP. Many client processes (running on the same machine or on different machines) can safely access the database simultaneously, because the database server manages an automatic lock mechanism.

### Embedded Mode

In embedded mode, the application uses ObjectDB as a class library, so no server process is needed in another JVM. The database file is accessed directly by the application's process, using JDO and ObjectDB jar files that are added to the application's classpath. In this mode,

the database file is locked for a single process, and multi user support may be achieved only by multithreading (several threads) in a single process.

**Client-Server Mode vs. Embedded Database Mode**

The embedded database mode is easier to use and faster because no server has to run in the background and no TCP/IP communication is needed. Adding JDO and ObjectDB jar files to the classpath is the only requirement. Client-Server mode, however, is useful for accessing remote databases and for database access of several processes simultaneously. Moving from one mode to the other can be done simply by changing a single string (the connection URL), because the JDO API is the same for both modes.

## 1.3.  ObjectDB Editions

ObjectDB for Java/JDO is available in three editions: Free, Embedded and Server.

- **Server Database Edition**

  This is the most advanced edition. It supports all ObjectDB features and both the embedded database operating mode as well as the client-server operating mode.

- **Embedded Database Edition**

  This edition is limited to the embedded database operating mode. Otherwise, it is very similar to the server database edition.

- **Free Database Edition**

  This is a special edition for personal non-commercial use. It is based on the embedded database edition, but has some additional restrictions and limitations.

This developer's guide is for all ObjectDB editions. Therefore, you can assume that every feature described in this guide is supported by all the editions, unless explicitly specified otherwise.

## 1.4.  ObjectDB in Web Applications

ObjectDB for Java/JDO is especially designed for Java web applications. Each one of the ObjectDB editions can be easily integrated into any JSP/servlet web application. The JDO Guest Book sample provided with ObjectDB demonstrates a simple web application that uses ObjectDB and JDO.

**Operating Modes**

Naturally, the server database edition, supporting client-server mode, provides the highest flexibility for web applications and advanced features, like remote database administration and manipulation. However, web applications may also use the ObjectDB embedded database edition and the free database edition in embedded mode as long as the database and the web

server are on the same machine. Because embedded database mode supports multithreading, and the web server handles simultaneous requests using threads, even in embedded database mode, multi user access to a database is supported.

## JDO Web Hosting

The embedded database mode is especially important if you are considering deploying a Java web application on a virtual web hosting server rather than on a dedicated server. Using embedded mode, it is easy to deploy a web application with ObjectDB on any web hosting system that supports servlets/JSP. Just drop the JDO and ObjectDB jar files into the WEB-INF/lib directory of the web application, and database support (in embedded mode) is made available.

# Chapter 2.  A Quick Tour

This chapter introduces basic ObjectDB and JDO concepts, using two sample programs. We start with the **HelloWorld** sample program, which is not JDO portable because it uses some ObjectDB extensions, but it is a good sample to start with because of its simplicity:

We then proceed with the **JDO Person** sample program, which demonstrates the process of building a minimal JDO portable application, step by step:

Both sample programs are contained in ObjectDB's **samples** directory.

## 2.1.  Hello World

The HelloWorld sample program manages a list of strings in the database. Each time the program is run another string is stored in the database and all the strings in the database are printed to standard output.

The output of the first run is expected to be:

```
Hello World 0
```

The output of the second run is expected to be:

```
Hello World 0
Hello World 1
```

After two runs, the database contains a list of two strings "Hello World 0" and "Hello World 1".

### Program Source Code

The program consists of a single source file, **HelloWorld.java**, containing a single class:

```
1   // A simple program that manages a list of strings in a database.
2
3   import java.util.*;
4   import javax.jdo.*;
5   import com.objectdb.Utilities;
6
7   public class HelloWorld {
8
9       public static void main(String[] args) {
10
11              // Create or open a database and begin a transaction:
12              PersistenceManager pm =
```

```
13                Utilities.getPersistenceManager("hello.odb");
14          pm.currentTransaction().begin();
15
16          // Obtain a persistent list:
17          ArrayList list;
18          try {
19              // Retrieve the list from the database by its name:
20              list = (ArrayList)pm.getObjectById("Hello World", true);
21          }
22          catch (JDOException x) {
23              // If not found - construct and store a new list:
24              list = new ArrayList();
25              Utilities.bind(pm, list, "Hello World");
26          }
27
28          // Add a new string to the persistent list:
29          list.add("Hello World " + list.size());
30
31          // Display the content of the persistent list:
32          Iterator itr = list.iterator();
33          while (itr.hasNext())
34              System.out.println(itr.next());
35
36          // Close the transaction and the database:
37          pm.currentTransaction().commit();
38          pm.close();
39      }
40 }
```

### How it works

Lines 3-5          Three import statements are required: for Java collections (line 3), for JDO (line 4) and for ObjectDB extensions (line 5).

Lines 11-14        A PersistenceManager instance representing a local database file, **hello.odb**, is obtained using the Utilities.getPersistenceManager(...) static method (lines 12-13). If a database file does not exist in that path, a new database file is created automatically.

To enable updating the content of the database, a transaction is begun (line 14), because in JDO, operations that affect the content of the database must always be contained within an active transaction.

Lines 16-26        The data structure of this program is an ArrayList containing String instances. ObjectDB, as a pure object database, can simply store a memory data structure in the database as is. If the database is not empty (not the first run), a previously stored ArrayList instance is expected to be retrieved from the database by its name "Hello World" using the getObjectById(...) method (line 20). You can ignore the second argument of the getObjectById(...) method that is discussed in section 6.3. If such an ArrayList is not found in the database (on the first run for example), an exception is thrown and a new

11

empty ArrayList instance is created (line 24). The new ArrayList instance is bound to the database with the name "Hello World", using the Utilities.bind(...) static method (line 25). An object bound to the database during an active transaction, is expected to be stored physically in the database when the transaction is committed (line 37).

Note: Memory objects that represent database objects (like list in this example) are called *persistent objects*. An object retrieved from the database is always persistent. A new object becomes persistent when it is bound to the database.

Lines 28-29    A new string ("Hello World 0", "Hello World 1", etc.) is added to the ArrayList.

Lines 31-34    The ArrayList is iterated (by a standard Java iterator) and its content is printed.

Lines 36-38    On transaction commit (line 37), new persistent objects (like a new ArrayList that becomes persistent in line 25), and modified persistent objects (like a retrieved ArrayList, which is modified in line 29) are automatically stored in the database. Objects in memory that are reachable from persistent objects (like the String instance, which is added in line 29) are also stored in the database automatically.

At the end, the database is closed (line 38). Usually, it is best to close the database in a finally block, as demonstrated in the next sample program, to ensure database closing in every situation, including exceptions.

**ObjectDB Extensions**

The com.objectdb.Utilities class implements ObjectDB extensions. Storing an ArrayList directly in the database with a specified name (line 25), is supported by ObjectDB but not by JDO. In JDO, only instances of special user defined classes (which are called *persistent classes*) can be stored in the database directly, and other types (such as Java collections) can be stored only as fields of persistent classes. In addition, objects are stored without names in JDO. Another extension is used to obtain a PersistenceManager instance (line 13). Obtaining a PersistenceManager in a portable JDO way, which is slightly more complicated, is demonstrated in the next program sample.

## 2.2.  Defining a Persistent Class

As noted above, in a portable JDO application, only instances of persistent classes are stored directly in the database. Predefined Java types like ArrayList, String and Date can be stored in the database only as fields of persistent classes.

### Persistent Classes

Persistent classes are user defined classes whose instances can be stored in the database using JDO. The JDO specification refers to persistent classes as *persistence capable classes*, but both terms are equivalent. Chapter 3 is dedicated to persistent classes.

To become persistent, a class has to:

- be declared in a JDO metadata file in XML format.
- include a no-arg constructor.
- implement the javax.jdo.spi.PersistenceCapable interface.

The PersistenceCapable interface includes more than 20 abstract methods, so implementing it explicitly is far from trivial. Therefore, persistent classes are usually defined without implementing that interface explicitly, and a special tool, the *JDO enhancer*, adds the implementation automatically.

### The Person Persistent Class

The following Person class serves as a persistent class:

```
 1 // The Person persistent class
 2
 3 public class Person {
 4
 5     // Persistent Fields:
 6     private String firstName;
 7     private String lastName;
 8     private int age;
 9
10     // Constructors:
11     public Person() {}
12     public Person(String firstName, String lastName, int age) {
13         this.firstName = firstName;
14         this.lastName = lastName;
15         this.age = age;
16     }
17
18     // String Representation:
19     public String toString() {
20         return firstName + " " + lastName + " (" + age + ")";
21     }
22 }
```

The Person class is an ordinary Java class with a no-arg constructor (line 11). If it is declared as persistent in a JDO metadata file, it can easily become persistent using the JDO enhancer.

**JDO Metadata**

Every persistent class must be declared as persistent in a special XML file, called the JDO metadata file. The Person class must be declared in a metadata file named either **package.jdo** (generic name) or **Person.jdo** (name of the class). More details about metadata files, including their location and naming rules, are provided in chapter 4, which is devoted to JDO metadata.

The following **package.jdo** file is located in the same directory as **person.class**:

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <!DOCTYPE jdo SYSTEM "http://java.sun.com/dtd/jdo_1_0.dtd">
3
4 <jdo>
5     <package name="">
6          <class name="Person" />
7     </package>
8 </jdo>
```

Every JDO metadata file contains a single <jdo> element (lines 4-8). Each package is represented by a <package> sub element (e.g. lines 5-7) containing <class> elements for all the persistent classes in the package (e.g. line 6). Both <package> and <class> elements must have a name attribute. An empty name is specified for the <package> element to represent the default package (in which the class is defined). More information can be provided in the metadata file (as shown in chapter 4), but this minimal form is sufficient here.

## 2.3.  Storing and Retrieving Objects

The **JDO Person** sample manages a collection of Person instances in the database. Three arguments have to be specified in order to run the program **Main** class:

```
> java Main George Bush 57
```

Each time the program is run, another Person instance is constructed (according to the specified first name, last name and age) and stored in the database.

**Program Source Code**

In addition to the two files from the previous section (**Person.java** and **package.jdo**), the program contains the following **Main.java** source:

```
1 // Main of the JDO Person sample.
2
```

```
 3 import java.util.*;
 4 import javax.jdo.*;
 5
 6 public class Main {
 7
 8     public static void main(String[] args) {
 9
10         // Check the arguments:
11         if (args.length != 3)
12         {
13             System.out.println(
14                 "Usage: java Main <first name> <last name> <age>");
15             System.exit(1);
16         }
17
18         try {
19             // Obtain a database connection:
20             Properties properties = new Properties();
21             properties.setProperty(
22                 "javax.jdo.PersistenceManagerFactoryClass",
23                 "com.objectdb.jdo.PMF");
24             properties.setProperty(
25                 "javax.jdo.option.ConnectionURL", "persons.odb");
26             PersistenceManagerFactory pmf =
27                 JDOHelper.getPersistenceManagerFactory(properties);
28             PersistenceManager pm = pmf.getPersistenceManager();
29
30             try {
31                 // Begin the transaction:
32                 pm.currentTransaction().begin();
33
34                 // Create and store a new Person instance:
35                 Person person = new Person(
36                     args[0], args[1], Integer.parseInt(args[2]));
37                 pm.makePersistent(person);
38
39                 // Print all the Persons in the database:
40                 Extent extent = pm.getExtent(Person.class, false);
41                 Iterator itr = extent.iterator();
42                 while (itr.hasNext())
43                     System.out.println(itr.next());
44                 extent.closeAll();
45
46                 // Commit the transaction:
47                 pm.currentTransaction().commit();
48             }
49             finally {
50                 // Close the database and active transaction:
51                 if (pm.currentTransaction().isActive())
52                     pm.currentTransaction().rollback();
53                 if (!pm.isClosed())
54                     pm.close();
55             }
56         }
57
58         // Handle both JDOException and NumberFormatException:
59         catch (RuntimeException x) {
60             System.err.println("Error: " + x.getMessage());
```

15

```
61          }
62      }
63 }
```

**How it works**

Lines 3-4     Two import statements are required: for Java collections (line 3), and for JDO (line 4). ObjectDB extensions are not used in this program.

Lines 19-28   A PersistenceManager instance representing a local database file, **person.odb**, is obtained using JDO portable code (slightly more complicated than the equivalent code in section 2.1). A detailed explanation of this code is provided in chapter 5.

Lines 31-32   A transaction on the database is begun.

Lines 34-37   A new Person instance is constructed (lines 35-36) and becomes persistent (line 37). The makePersistent(...) method binds the object to the database, but without specifying a name (as done by the bind(...) method in section 2.1). This is the JDO portable way to add objects to the database. The new persistent object is physically stored in the database only when the transaction is committed (line 47).

Lines 39-44   The Person instances in the database are represented in JDO by an Extent instance (line 40). The second argument of the getExtent(...) method indicates whether the Extent should also represent instances of subclasses (changing false to true has no effect here, because Person has no subclasses). The Extent of the class can be iterated using an ordinary Java Iterator (lines 41-43). Unlike ordinary Java iteration, when the end of Extent iteration is reached it is recommended to use closeAll() (line 44) or close(...) to release resources.

Lines 46-47   Updates are physically applied to the database when the transaction is committed (line 47).

Lines 49-55   The database is closed (lines 53-54) in a finally block, to ensure it is always properly closed, even in case of an error (when an exception is thrown). If a transaction is still active (e.g. exception is thrown in lines 34-44), it should be ended before closing the database. Ending the transaction with rollback() discards all the changes done during its activity (line 52).

Lines 58-61   When working with JDO, instances of JDOException and its subclasses may be thrown as exceptions. Because JDOException is a subclass of RuntimeException, the compiler does not enforce the throws declaration if a

> catch block does not exist in the method. Therefore, extra caution is required by the developer to make sure that at some level a proper catch block (like in lines 58-61) exists.

An attempt to run the program now results in the following error:

```
Error: Class 'Person' is not PersistenceCapable
```

Something is still missing to make this program work - the JDO enhancement.

## 2.4.  On The Fly JDO Enhancement

On the fly JDO enhancement is the easiest way to use the JDO enhancer. To use it, an additional main class, named eMain (enhancer Main) is defined:

```
 1 // An enhancer main class for the JDO Person Sample
 2
 3 public class eMain {
 4
 5     public static void main(String[] args) {
 6
 7         // Enhance the Person class if necessary:
 8         com.objectdb.Enhancer.enhance("Person");
 9
10         // Now run the real main:
11         Main.main(args);
12     }
13 }
```

Class eMain replaces the original Main class as a program entry point at development time. It starts by enhancing necessary classes (line 8). JDO enhancement includes modifying class files in order to add implementation of the PersistenceCapable interface, where necessary, at the byte code level. When this process is completed the real main is called (line 11). Of course, this arrangement should only be used during development. Eventually, the application should be deployed with ready to use enhanced classes and without the eMain class.

The enhancer should print the following message (unless **Person.class** is already enhanced):

```
[ObjectDB Enhancer] 1 new persistence capable class has been enhanced.
```

If you see the following message, something went wrong:

```
[ObjectDB Enhancer] 1 new persistence aware class has been enhanced.
```

The enhancer did not find the metadata file in which Person is declared as persistent. Therefore, it was not able to enhance the Person class as a persistent class (a persistence **aware** class is not a persistence **capable** class). Additional information on JDO enhancement, persistence capable classes and persistence aware classes is provided in section 3.3.

# Chapter 3.  Persistent Classes

Persistent Classes are user defined classes whose instances can be stored in a database using JDO. Instances of these classes that represent objects in the database are called *persistent objects* or *persistent instances*. Objects that do not represent anything in the database (including instances of persistent classes that live only in memory) are called *transient objects* or *transient instances*. The JDO specification refers to persistent classes as *persistence capable classes*.

This chapter contains the following sections:

## 3.1.  Persistent Classes

Only classes that represent data in the database should be declared persistent because persistent classes have some overhead. Classes that are not declared persistent are called *transient classes* and can have only transient objects as instances.

To become persistent, a class has to:

- be declared in a JDO metadata file in XML format.
- include a no-arg constructor.
- implement the javax.jdo.spi.PersistenceCapable interface.

The PersistenceCapable interface includes more than 20 abstract methods, so implementing it explicitly is not trivial. It is easier to define a class without implementing the PersistenceCapable interface, and have the JDO Enhancer add the interface implementation automatically, as explained in section 3.3. ObjectDB's JDO Enhancer also adds a no-arg constructor if it is missing, so out of the three requirements described above, developers only have to provide the JDO metadata declaration. Chapter 4 explains how to write the JDO metadata declaration.

Aside from the requirements described above, a persistent class is like any other Java class. It can include constructors, methods, fields, attributes (final, abstract, public), inheritance (even from a non persistent class), interface implementations, inner classes, etc. The class members

(constructors, methods and fields) can have any access modifiers (i.e. public, protected, package or private).

## 3.2.  Persistent Fields and Types

Storing a persistent object in the database does not store methods and code. Only the state of the object as reflected by its *persistent fields* is stored. Persistent fields, by default, are all the fields that are not defined as static, final or transient, and have *persistent types*. Every persistent class is a persistent type. The following predefined system types are also persistent types:

- All the primitive types - boolean, byte, short, char, int, long, float and double.
- Selected classes in package java.lang: Boolean, Byte, Short, Character, Integer, Long, Float, Double, Number and String.
- Selected classes in package java.util: Date, Locale, HashSet, TreeSet, ArrayList, LinkedList, Vector, HashMap, TreeMap and Hashtable, and the interfaces - Collection, Set, List and Map.
- java.math.BigInteger and java.math.BigDecimal
- Any array of persistent type, including multi dimensional arrays

Some of the persistent types above are defined as optional by JDO (arrays and most of the collection classes), but ObjectDB supports all of them. You cannot extend the list to include other system types. The only way to add support for additional types is to define new persistent classes. For example, the class java.awt.Image is not supported by JDO. You can store images in byte[] fields or you can define a new persistent class for this purpose (images can also be stored as external files, storing only the file names or paths in the database).

Static and final fields can never be persistent. The transient modifier can be used to exclude other fields from being stored in the database. It is recommended to explicitly define fields of non persistent types as transient, even though they are considered transient in JDO by default. A field with a transient modifier can still become persistent by an explicit instruction in the JDO metadata (useful when a field has to be transient in serialization but persistent in JDO)

Of course, persistent fields should not hold non persistent values at runtime when the object is stored in the database. For example, a field whose type is Number (which is an abstract class) must have a reference to an object whose type is persistent at storage time, e.g. an Integer, or null value. Similarly, a field whose type is one of the persistent collection interfaces (Collection, Set, List and Map) cannot refer to an unsupported collection, or a collection that contains objects of non persistent types, at storage time.

As an extension to JDO, ObjectDB supports storing instances of any of the persistent types defined above in the database (as demonstrated in section 2.1 for ArrayList). Portable JDO

applications, however, should only store instances of user defined persistent classes in the database directly and use the other persistent types only as fields in these classes.

## 3.3.  JDO Enhancement

The ObjectDB JDO Enhancer is a post compilation tool that modifies the byte code of compiled classes. Classes to be enhanced must be located in class files and not in jar files. However, enhanced classes can be packed in jar files.

Two types of classes must be enhanced:

### Persistence Capable Classes

Incomplete persistent classes that do not implement the PersistenceCapable interface explicitly (as is usually the case) must be enhanced. The enhancer modifies the byte code of these classes, and the result is classes that do implement the PersistenceCapable interface. The JDO specification refers to these classes as persistence capable classes.

### Persistence Aware Classes

Classes that access or modify persistent fields are referred to as *persistence aware classes*. Most persistent classes are also persistence aware, according to this definition, but usually the term 'persistence aware' refers to classes that are not persistent but contain code that accesses or modifies persistent fields of other classes. Directly accessing a single persistent field is sufficient for a class to become persistence aware. Accessing a persistent field indirectly by calling a method or by getting its value as an argument to a method, on the other hand, does not make the class persistence aware. One exception, however, is in working with persistent array fields (persistent fields of types like int[] or Object[]). Any class in which a persistent array is modified, no matter how the array is accessed, is considered a persistence aware class by ObjectDB. This exception only applies to persistent arrays and not to persistent collections (such as ArrayList).

Persistence aware classes must be enhanced in order to track persistent fields. ObjectDB must know when a persistent field is modified during a transaction because the change must be applied to the database when the transaction is committed. ObjectDB must also know when a persistent field is accessed because it might require loading additional data and objects from the database (as part of transparent persistence support). Enhancement replaces direct persistent field accesses with equivalent methods that perform the same accesses but also report to ObjectDB. Tracking changes in persistent arrays (in enhanced classes), which is optional in JDO, is also fully supported by ObjectDB.

Persistence aware classes that are not persistent themselves are supported by ObjectDB and JDO, but it might be a good practice to avoid them. This can be achieved by defining all

persistent fields as private, or by accessing the fields directly only in persistent classes. Because persistent classes are automatically enhanced also as persistence aware, working with persistent fields only in persistent classes simplifies things.

## Command Line Enhancement

The classic method to enhance classes is to run the enhancer from the command line.

First, the classpath has to be set to include ObjectDB and JDO jar files. For example:

```
On Windows command:
 > set classpath=%classpath%;.;c:\objectdb\lib\odbfe.jar;c:\objectdb\lib\jdo.jar

On Unix sh / bash shell:
  % CLASSPATH=${CLASSPATH}:.:/objectdb/lib/odbfe.jar:/objectdb/lib/jdo.jar
On Unix csh shell:
  %                             setenv                          CLASSPATH
${CLASSPATH}:.:/objectdb/lib/odbfe.jar:/objectdb/lib/jdo.jar
```

The enhancer can be run without arguments to get usage instructions:

```
> java com.objectdb.Enhancer
ObjectDB JDO Enhancer - Version 1.00
Copyright (C) ObjectDB Software 2003. All rights reserved.


Usage: java com.objectdb.Enhancer [ <options> | <class> | <filename> ] ...


<class> - class name (without .class suffix) found in the CLASSPATH
<filename>  -  absolute  or  relative  file  specification  (including  *?
wildcards)


<options> include:
  -cp        classpath for input user classes
  -d         output path for enhanced classes
  -s         include sub directories in search
  -noopt     disable enhancement optimization
```

You can specify class files for enhancement explicitly or by using wildcards. The output message tells how many persistence capable and persistence aware classes have been enhanced (classes that are already enhanced are excluded). The JDO metadata is expected to be found automatically. If it is not found, classes are enhanced as persistence aware but not as persistence capable, so always notice the output message that distinguishes persistence capable classes from persistence aware classes that are not persistent.

```
> java com.objectdb.Enhancer test/*.class test/pc/*.class
[ObjectDB Enhancer] 2 new persistence capable classes have been enhanced.
[ObjectDB Enhancer] 1 new persistence aware class has been enhanced.
```

Class files can also be searched automatically in sub directories using the -s option (the "*.class" expression is in quote marks here to avoid auto resolving at the shell level in some environments):

```
> java com.objectdb.Enhancer -s "*.class"
```

Use the -d option to redirect the output classes to a different directory, keeping the original class files unchanged:

```
> java com.objectdb.Enhancer -s "*.class" -d enhanced
```

Instead of specifying class files, you can specify class names (e.g. test.X) and packages (e.g. test.pc.*):

```
> java com.objectdb.Enhancer test.X test.pc.*
```

Use the -cp option to specify an alternative classpath in which to look for the classes (the default is the classpath in which the enhancer is running):

```
> java com.objectdb.Enhancer -cp src test.X test.pc.*
```

**On the Fly JDO Enhancement**

Command line enhancement is useful for testing and for build scripts (like ANT for example), but in most Java IDEs a plugin is required to integrate a JDO enhancer into the IDE Build command. A simple alternative that does not require a plugin and works on any Java IDE (and also from the command line) is to add a new simple main class to a project that applies on the fly enhancement:

```
package test;

/** Additional main - On the Fly JDO Enhancer */
public class eMain {
    public static void main(String[] args) {

        // Always start by calling the enhancer:
        com.objectdb.Enhancer.enhance("test.pc.*,test.X");

        // Now move to the original entry point:
        RealMain.main(args);
    }
}
```

The eMain class ("enhancer Main") becomes the new entry point at development time. First the enhancer is called, and when enhancement is completed, the original entry point is called. The overhead to run the enhancer on every run is negligible because the ObjectDB Enhancer is very fast. Notice that persistent classes should not be referenced in eMain. Otherwise they might be loaded by the JVM too early, before enhancement is completed.

The com.objectdb.Enhancer.enhance(...) static method accepts as an argument a string specifying the classes to be enhanced. The syntax is adopted from the import statement. You

can specify full class names (e.g. test.X) as well as entire packages (e.g. test.pc.*), where elements are separated by commas.

## 3.4.  InstanceCallbacks

The javax.jdo.InstanceCallbacks interface represents events in a persistent object's lifecycle. By implementing the InstanceCallbacks interface, a persistent class can handle these events.

Four methods are defined in the InstanceCallbacks interface:

- void jdoPostLoad()

  Called when a persistent object is being loaded from the database, just after the default fetch group fields (see chapter 4) are initialized. A possible action is to initialize non persistent fields of the object. Other persistent objects, and even persistent fields in this loaded object that are not in the default fetch group, should not be accessed by this method.

- void jdoPreStore()

  Called when a persistent object is going to be stored in the database during transaction commit. A possible action is to apply last minute changes to persistent fields in the object just before the store.

- void jdoPreClear()

  Called when the persistent fields are going to be cleared at transaction end (for example on rollback). This event is rarely used. Other persistent objects and even persistent fields in this loaded object that are not in the default fetch group should not be accessed by this method.

- void jdoPreDelete()

  Called during the execution of deletePersistent(...), just before an object is deleted. A possible action is to delete depended persistent objects as demonstrated below.

Because the four events are defined in the same interface, to implement one of them you have to implement the entire interface with all the four methods, keeping the methods that are irrelevant empty. For example:

```
package test;
import javax.jdo.*;

class Line implements InstanceCallbacks {
  private Point p1, p2;
  public Line(Point p1, Point p2) {
    this.p1 = p1;
    this.p2 = p2;
  }
  public void jdoPostLoad() {
  }
  public void jdoPreClear() {
```

```
  }
  public void jdoPreStore() {
  }
  public void jdoPreDelete() {
    PersistenceManager pm = JDOHelper.getPersistenceManager(this);
    pm.deletePersistent(p1);
    pm.deletePersistent(p2);
  }
}
```

When a Line instance is deleted, the two referred Point instances are deleted with it.

## 3.5.  Automatic Schema Evolution

Most of the changes to persistent classes do not affect the database. This includes adding, removing and changing constructors, methods and non persistent fields. Changes to persistent fields (schema changes), however, do affect the database. New persistent objects are stored using the new schema (the new class structure), and old persistent instances, which were stored using the old schema, have to be converted to the new schema. ObjectDB implements an automatic schema evolution mechanism, which enables transparent use of old schema instances. When an old instance is loaded into the memory it is automatically converted into an instance of the new up-to-date persistent class.

The conversion is straightforward. New persistent fields that are missing in the old schema are initialized with default values (0, false or null). Old persistent fields that are missing in the new class are just ignored. When a type of a persistent field is changed, if casting of the old value to the new type is valid in Java (for example from int to float and from float to int) the old value is converted automatically to the new type. If casting is illegal (for example from int to Date) the field is initialized with a default value as a new field.

When an upgraded object is stored again in the database, it is stored using the new schema. Until then, the conversion is done only in memory each time the object is loaded, and the content of the object in the database remains in its old schema without any change.

# Chapter 4.  JDO Metadata

A JDO metadata file is an XML file with a '.jdo' suffix, containing information about one or more persistent classes. All the persistent classes, and only them, have to be declared in a metadata file. The metadata is used first by the JDO Enhancer, and later by the JDO system. Therefore, at deployment, .jdo metadata files must be packaged with an application's .class files and the other resources (possibly in a jar file).

This chapter contains the following sections:

## 4.1.  JDO Metadata Files

During JDO enhancement and later at runtime, ObjectDB determines whether or not each class is persistent. It searches for JDO metadata description of each class in several .jdo files in a pre defined order. If a metadata description is found, the class is persistent, and if not, the class is transient.

Metadata for class a.b.X (a.b is the package name, X is the class name), whose class file is a/b/X.class, is searched in the following paths (in the order shown):

```
META-INF/package.jdo
WEB-INF/package.jdo
package.jdo
a/package.jdo
a/b/package.jdo
a/b/X.jdo
```

Metadata for class X in the default package is searched in the following paths (in the order shown):

```
META-INF/package.jdo
WEB-INF/package.jdo
package.jdo
X.jdo
```

A metadata file with the name X.jdo must be dedicated to a single class whose name is X. Metadata for multiple classes can be specified in a package.jdo file located in META-INF, WEB-INF or in any other path at the level of the classes or above. Determining where to put

the metadata of every persistent class is your responsibility. When the metadata for a class is found, the search is stopped. Therefore, only the first metadata for a class in the search order specified above has effect.

## 4.2.  Metadata for Classes

We start with a basic JDO metadata file:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE jdo SYSTEM "http://java.sun.com/dtd/jdo_1_0.dtd">

<jdo>
  <package name="">
    <class name="A" />
  </package>
  <package name="test">
    <class name="B" />
    <class name="C" persistence-capable-superclass="B" />
    <class name="D" requires-extent="false" />
  </package>
</jdo>
```

A JDO metadata file is an XML file with a single root element - <jdo>. The <jdo> root has one or more <package> sub elements. Each <package> element has one or more <class> sub elements. Both <package> and <class> elements have a required name attribute. The metadata above defines class A (in the default package) and classes B, C and D (in package test), as persistent.

In addition to the required name attribute, a <class> element can have one or more of the following optional attributes:

### persistence-capable-superclass

The persistence-capable-superclass attribute usually specifies the direct super class if it is also persistent. In the above metadata example, class C is probably defined as a subclass of class B (using extends). But there is also another possibility. Class C might be a subclass of a non persistent class X, which is a subclass of class B. That is also a valid structure because JDO enables declaring a persistent class as a subclass of a non persistent class. Of course, in that case the fields of class X are not persistent fields, and when an instance of class C is stored, only persistent fields from classes B and C are stored. The closest persistent super class in the inheritance hierarchy must be specified using the persistence-capable-superclass attribute. This attribute is omitted only if there is no persistent super class anywhere in the inheritance hierarchy. When the super class is in the same package the package name can be omitted. Otherwise the full name of the super class, which includes the package name, has to be specified.

**requires-extent (true | false)**

By default, JDO manages an extent for every persistent class. An extent enables iteration over persistent instances of a class (including or excluding instances of subclasses) as well as execution of queries against the class instances. However, maintaining an extent for a class has some overhead in terms of time and storage space. When extent management is not needed, it can be omitted by specifying requires-extent="false", as shown above for class D. If requires-extent="false" is specified for a class, it must also be specified for its super class as declared in the persistence-capable-superclass attribute.

**identity-type and objectid-class**

The identity-type and objectid-class attributes, which are defined in the JDO specification, are ignored by ObjectDB if specified (as a pure object database, ObjectDB always uses datastore identity with its own object-id class).

## 4.3.  Metadata for Fields

Unlike persistent classes, which must be listed in the JDO metadata, persistent field descriptions can often be omitted. In most cases, the default management of fields by ObjectDB is adequate. Metadata for fields is required only for changing the default. Therefore, only fields with modified behavior should be specified in the metadata.

The following file demonstrates JDO metadata for fields:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE jdo SYSTEM "http://java.sun.com/dtd/jdo_1_0.dtd">

<jdo>
  <package name="test">
    <class name="A">
      <field name="f0" persistence-modifier="persistent" />
      <field name="f1" persistence-modifier="none" />
      <field name="f2" persistence-modifier="transactional" />
      <field name="f3" default-fetch-group="true" />
      <field name="f4" default-fetch-group="false" />
      <field name="f5" embedded="true" />
      <field name="f6" embedded="false" />
      <field name="f7" null-value="exception" />
      <field name="f8" null-value="none" />
      <field name="f9" null-value="default" />
    </class>
  </package>
</jdo>
```

A single persistent class, test.A, is declared by a <class> element containing <field> sub elements for 10 of its fields (there may be also other persistent fields in class A that are not specified in the metadata). Every field can have zero or more <field> elements. A <field>

element may have several attributes (not just two as demonstrated above), but the name attribute specifying a name of a field in the class is always required.

### persistence-modifier (persistent | none | transactional)

The default rules for determining if a field is persistent are explained in <u>section 3.2</u>. The persistence-modifier attribute makes it possible to change the default. Specifying a persistent value, as demonstrated by field f0, changes a field that is transient by default to persistent. For example, a field with a transient modifier in the Java source (useful for defining fields as transient in serialization and persistent in JDO), or a field whose declared type is java.lang.Object or some interface, but holds at runtime only values of persistent types. Specifying a none value, as demonstrated by field f1, changes a field that is persistent by default to transient, as an alternative to the Java transient modifier (for example, when a field has to be persistent in serialization). A field that is declared as transactional, like f2 above, has similar behavior to transient because its value is never stored in the database. The main difference is that, on transaction rollback, it returns automatically to its value at the beginning of the transaction.

### default-fetch-group (true | false)

The default-fetch-group attribute indicates that a field should be managed in a group with other fields. When a persistent object is retrieved from the database its fields are not ready yet. Only when the program accesses a field is the field value loaded automatically by ObjectDB from the database. If the field belongs to the default fetch group, values for all the fields in the group are also loaded. The default fetch group should include fields that are needed often and are relatively small. By default, the group contains all the fields with primitive types (e.g. int), types defined in java.lang (e.g. String and Integer), types defined in java.math (e.g. BigInteger), and java.util.Date. Collections, arrays and references to user defined classes are excluded by default. The default-fetch-group attribute can change the default, as demonstrated by fields f3 and f4.

### embedded (true | false)

The embedded attribute is relevant for persistent reference fields. It indicates whether or not the content of the referred object should be stored as part of the referring object, as an *embedded object*. Embedded objects can reduce storage space and improve efficiency, but they do not have an object ID and cannot be shared by references from multiple objects. In addition, embedded objects of persistent classes are not included in the extents of their classes, so they cannot be queried directly. When the embedded attribute is not specified, ObjectDB embeds objects by default, for all fields except fields whose type is a user defined persistent classes or java.lang.Object. To use embedded objects for fields of user defined persistent classes, a metadata has to be specified as demonstrated by field f5. Wrapper

objects, strings, dates, collections and arrays are embedded by default. To use them as non embedded (useful when the field is very large or rarely used) a metadata has to be specified, as demonstrated by field f6.

### null-value (exception | none | default)

The null-value attribute is also intended for persistent reference fields. It indicates whether the field can accept null values or not. If an exception value is specified, as demonstrated by field f7, a JDOUserException is thrown on any attempt to store a persistent object with a null value in that field. If the null-value attribute is not specified or specified with a none value, as demonstrated by field f8, null values are allowed. If a default value is specified, as demonstrated by field f9, null values are replaced at storage time with default values (e.g. new Integer(0) for a java.lang.Integer field, "" for a java.lang.String field, empty collection, 0 size array, and so on).

### primary-key

The primary-key attribute, defined in the JDO specification, is irrelevant when using datastore identity, which is the only object identity supported by ObjectDB. To enforce unique values in a field you can define a unique index as explained in <u>section 4.5</u>.

## 4.4.  Arrays, Collections and Maps

Special XML sub elements are available for array, collection and map fields:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE jdo SYSTEM "http://java.sun.com/dtd/jdo_1_0.dtd">

<jdo>
  <package name="test">
    <class name="B">
      <field name="f0" embedded="true">
        <array embedded-element="true" />
      </field>
      <field name="f1" embedded="true">
        <collection embedded-element="true" />
      </field>
      <field name="f2" embedded="true">
        <map embedded-key="true" embedded-value="true" />
      </field>
    </class>
  </package>
</jdo>
```

A <field> element representing a persistent field whose type is collection, map or array, can have a <collection>, <map> or <array> sub element, respectively.

**embedded-element, embedded-key, embedded-value (true | false)**

The embedded-element attribute indicates whether objects in a collection or array should be stored as embedded objects or not. To understand the difference between embedded and embedded-element, think about a collection field containing instances of a user defined persistent class. Specifying embedded=true in the <field> element (which is the default for collections and arrays anyway) indicates that the contained references are embedded, but **not** the referenced objects themselves because persistent class instances are not embedded by default. Specifying embedded-element=true in the <collection> sub element indicates that the objects are also embedded, not just their references. If embedded=false is specified in the <field> element, the objects are never embedded regardless of the embedded-element attribute, and even the references are stored externally. System types are embedded by default, so a collection of strings or dates is fully embedded by default. Specifying embedded-element=false changes it in a way that every String or Date is stored as a non embedded object (with a unique object ID). The embedded-key and embedded-value indicate whether the keys and the values of a map should be stored as embedded objects or not, similar to embedded-element, which is only for collections and arrays.

**element-type, key-type, value-type**

The key-type and value-type attributes defined in the JDO specification for the <Map> element are ignored by ObjectDB. The element-type attribute of the <collection> element has effect only in index declarations, as explained in the next section.

## 4.5.  Index Definition

Querying a large extent without indexes may take a significant amount of time because it requires iteration over all the class instances, one by one. Using proper indexes the iteration can be avoided, and complex queries over millions of objects can be executed quickly. Index management introduces overhead in terms of maintenance time and storage space, so deciding which fields to define with indexes should be done carefully. Indexes are not supported by the free database edition.

The JDO specification does not define a standard method of index declaration, so the declaration syntax is specific to ObjectDB. Every <class> element can have <extension> sub elements declaring indexes. The JDO standard provides <extension> for vendor specific declarations, so the metadata file remains JDO portable. Because index declarations are specified directly in the <class> element, <field> elements are not necessarily required for the fields for which indexes are declared.

The following metadata declares simple indexes for two fields. Field f0 has an ordinary index and field f1 has a unique index. A field that has a unique index must have unique values

among persistent instances of the class. An exception is thrown on any attempt to store an object with a value that is the same in some other persistent instance of the same class (or its subclasses).

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE jdo SYSTEM "http://java.sun.com/dtd/jdo_1_0.dtd">

<jdo>
  <package name="test">
    <class name="A">
      <extension vendor-name="objectdb" key="index" value="f0" />
      <extension vendor-name="objectdb" key="unique-index" value="f1" />
    </class>
  </package>
</jdo>
```

Simple indexed fields (as shown above) can have any of the following types:

- embedded value types:
- primitive type (boolean, byte, short, char, int, long, float or double)
- embedded wrapper (Boolean, Byte, Short, Character, Integer, Long, Float or Double)
- embedded java.lang.String
- embedded java.util.Date
- external (non embedded) reference of any type

There are two types of indexes: *value indexes* for value type fields, and *reference indexes* for non embedded reference fields. Value indexes manage a reversed list of persistent objects for every value in the indexed field. Both equality (==, !=) and comparison (<, <=, >, >=) queries are supported by value indexes. Reference indexes manage a reversed list of persistent objects for every reference in the indexed field. Only equality (==, !=) queries are supported by reference indexes.

Fields of value types (primitive types, wrapper types, String and Date) are embedded by default, but if, for instance, a String field is defined as non embedded, an index on that field is a reference index. In that case equality queries will check reference equality (as == operator in Java) and not value equality (as the equals(...) method).

A reference field cannot have a direct index if it is defined as embedded (because the referenced objects are embedded and do not have object IDs). However, in this case, the fields of the embedded object can have indexes, as shown in the following metadata:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE jdo SYSTEM "http://java.sun.com/dtd/jdo_1_0.dtd">

<jdo>
  <package name="test">
    <class name="B">
      <field name="f0" embedded="true" />
```

```
        <extension vendor-name="objectdb" key="index" value="f0.x" />
        <field name="f1" embedded="true" />
        <extension vendor-name="objectdb" key="index" value="f1.y.z" />
    </class>
  </package>
</jdo>
```

Field f0 holds a reference to a persistent object containing a persistent field named x. Because an x value is stored directly in every instance of B (field f0 is embedded), field x can have an index as a direct field of B (reference index or value index according to its type). This can be extended further to nested embedded objects as demonstrated by field f1.

Indexes can also be applied to embedded array and collection fields to accelerate contains(...) queries. Arrays of value types (including embedded wrappers, strings and dates) get value indexes, and array of reference types get reference indexes. The same is true with collections, except that the types of the elements must be declared explicitly using element-type attributes because it is not specified in the Java code.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE jdo SYSTEM "http://java.sun.com/dtd/jdo_1_0.dtd">

<jdo>
  <package name="test">
    <class name="C">
      <field name="f0">
        <collection element-type="int" />
      </field>
      <extension vendor-name="objectdb" key="index" value="f0" />
      <field name="f1">
        <collection element-type="java.lang.String" />
      </field>
      <extension vendor-name="objectdb" key="unique-index" value="f1" />
      <field name="f2">
        <collection element-type="B" />
      </field>
      <extension vendor-name="objectdb" key="index" value="f2" />
      <field name="f3">
        <collection element-type="B" embedded-element="true" />
      </field>
      <extension vendor-name="objectdb" key="index" value="f3.x" />
    </class>
  </package>
</jdo>
```

For fields f0 and f1, which are collections of values, value indexes are declared. For field f2, which is a collection of external references, a reference index is declared. Field f3, which is a collection of embedded objects, cannot have a direct index, but fields of its embedded elements, like x, can have an index, as demonstrated above.

# Chapter 5.  JDO Connections

This chapter describes three essential interfaces for using JDO:

The PersistenceManagerFactory interface represents a factory of database connections.

Its main role is to provide PersistenceManager instances.

The PersistenceManager interface represents a database connection.

Every operation on a database requires a PersistenceManager instance.

The Transaction interface represents a transaction on a database.

Every operation that modifies the content of the database requires an active transaction.

The focus of this chapter is on setting up a runtime environment for using JDO. Explanations on how to use this environment to do database operations, such as storing, retrieving, updating and deleting database objects, are provided in the next chapter (chapter 6).

## 5.1.  javax.jdo.PersistenceManagerFactory

Most applications require multiple database connections during their lifetime. For instance, in a web application it is very common to establish a separate database connection for every web request. In general, holding a database connection open for longer than necessary is not recommended in multi user applications because of the resources that every connection consumes.

Database connections are managed in JDO by the PersistenceManagerFactory interface. A PersistenceManagerFactory instance represents a specific database address (local or remote), and connections to that database are obtained using its getPersistenceManager() method. Each time this method is called, a PersistenceManager instance representing a connection to the database is obtained. To improve efficiency, a PersistenceManagerFactory instance may manage a pool of free PersistenceManager instances. From the developer's point of view, connection pool management is transparent. The functionality of PersistenceManager instances returned by the getPersistenceManager() method is the same for both new database connections and database connections obtained from a pool.

Both PersistenceManagerFactory and PersistenceManager are defined as JDO interfaces, not as classes. Every JDO implementation, including ObjectDB, defines classes that implement these interfaces. When using ObjectDB you are actually working with instances of ObjectDB classes, but to make your application JDO portable these objects are accessed through the standard JDO interfaces.

### Obtaining a PersistenceManagerFactory

The following code demonstrates how to obtain a PersistenceManagerFactory instance:

```
import java.util.Properties;
import javax.jdo.*;
     :
     :
  Properties properties = new Properties();
  properties.setProperty(
    "javax.jdo.PersistenceManagerFactoryClass", "com.objectdb.jdo.PMF");
  properties.setProperty(
    "javax.jdo.option.ConnectionURL", "local.odb");

  PersistenceManagerFactory pmf =
    JDOHelper.getPersistenceManagerFactory(properties);
```

The requested PersistenceManagerFactory is specified using a Properties instance. Each property consists of two strings, a name that identifies the property, and a value for that property. The two properties that are set in the code above are usually essential:

- javax.jdo.PersistenceManagerFactoryClass

  Specifies the real type of the constructed PersistenceManagerFactory instance. When using ObjectDB, the value should always be "com.objectdb.jdo.PMF" (the name of ObjectDB's class that implements the PersistenceManagerFactory interface).

- javax.jdo.option.ConnectionURL

  Specifies a database location. To access a database file directly using embedded mode, specify either its absolute path or its relative path.

The JDOHelper.getPersistenceManagerFactory(...) static method constructs and returns a new PersistenceManagerFactory instance, based on the specified properties.

### Using a Properties File

Properties can also be specified in a file, as demonstrated by the following properties file:

```
javax.jdo.PersistenceManagerFactoryClass=com.objectdb.jdo.PMF
javax.jdo.option.ConnectionURL=local.odb
```

Assuming the name of the file is **jdo.properties**, it can be loaded by a class in the same directory using:

```
import java.io;
import java.util.Properties;
import javax.jdo.*;
       :
       :
  InputStream in = getClass().getResourceAsStream("jdo.properties");
  try {
    Properties properties = new Properties();
    properties.load(in);
```

```
  PersistenceManagerFactory pmf =
    JDOHelper.getPersistenceManagerFactory(properties);
}
finally {
  in.close();
}
```

Notice that the code above may throw an IOException if the **jdo.properties** file is not found. In addition, the call to getClass() should be replaced by some other expression (e.g. MyClass.class) in a static context (i.e. a static method and a static initializer) in which this is undefined.

## Connection Properties

The javax.jdo.option.ConnectionURL property specifies the database location and whether embedded mode or client server mode should be used to access that database. To use embedded mode, an absolute path or a relative path of a local database file has to be specified. To use client server mode, a url in the format **objectdb://host:port/path** has to be specified. In this case, a database server is expected to be running on a machine named **host** (could be domain name or ip address) and listening to the specified **port** (the default is 6136 when not specified). The **path** indicates the location of the database file on the server, relative to the server root path (see chapter 8 for more details). Client server mode is only supported by the ObjectDB server database edition.

The following code demonstrates a connection to a database located at path **/my.odb** in a database server running on **localhost** (the same machine) and listening to the default port (6136):

```
Properties properties = new Properties();
properties.setProperty(
  "javax.jdo.PersistenceManagerFactoryClass", "com.objectdb.jdo.PMF");
properties.setProperty(
  "javax.jdo.option.ConnectionURL", "objectdb://localhost/my.odb");
properties.setProperty("javax.jdo.option.ConnectionUserName", "john");
properties.setProperty("javax.jdo.option.ConnectionPassword", "itisme");

PersistenceManagerFactory pmf =
  JDOHelper.getPersistenceManagerFactory(properties);
```

The ConnectionUserName and ConnectionPassword properties are usually required in a client server connection to enable user identification and permission checking. Username and password are optional in a client server connection only when the server configuration allows anonymous access to the database. If not specified here, username and password can also be specified later when obtaining a PersistenceManager instance, as explained in the next section. In embedded mode, in which permissions are not managed, these properties are always optional and ignored if specified.

Connection properties can also be managed later using PersistenceManagerFactory's methods:

```
// Using the setter methods:
pmf.setConnectionURL("objectdb://localhost/data.odb");
pmf.setConnectionUserName("john");
pmf.setConnectionPassword("itisme");

// Using the getter methods:
System.out.println("Database URL: " + pmf.getConnectionURL());
System.out.println("Username: " + pmf.getConnectionUserName());
// No getter for the password because of security considerations.
```

## Transaction Properties

JDO defines five optional modes for working with transactions, represented by five flags that can be set or cleared. All these optional modes are supported by ObjectDB. Flags can be specified for a specific Transaction (as shown in section 5.3) or at the PersistenceManagerFactory level as a default for all the transactions of PersistenceManager instances obtained from that factory.

Setting these flags for a PersistenceManagerFactory can be done using boolean properties:

```
properties.setProperty("javax.jdo.option.Optimistic", "false");
properties.setProperty("javax.jdo.option.NontransactionalRead", "true");
properties.setProperty("javax.jdo.option.NontransactionalWrite", "false");
properties.setProperty("javax.jdo.option.RetainValues", "true");
properties.setProperty("javax.jdo.option.RestoreValues", "false");

PersistenceManagerFactory pmf =
  JDOHelper.getPersistenceManagerFactory(properties);
```

Actually, the settings above are redundant because they are the default flag values in ObjectDB (but not in JDO, which does not specify a default). Opposite values should be specified in order to change the default.

In addition, the PersistenceManagerFactory class includes getter and setter methods:

```
// Using the setter methods:
pmf.setOptimistic(true);
pmf.setNontransactionalRead(false);
pmf.setNontransactionalWrite(true);
pmf.setRetainValues(false);
pmf.setRestoreValues(true);

// Using the getter methods:
boolean flag1 = pmf.getOptimistic();
boolean flag2 = pmf.getNontransactionalRead();
boolean flag3 = pmf.getNontransactionalWrite();
boolean flag4 = pmf.getRetainValues();
boolean flag5 = pmf.getRestoreValues();
```

Transaction modes are discussed in section 5.3, which deals with transactions.

## 5.2.  javax.jdo.PersistenceManager

The javax.jdo.PersistenceManager interface represents database connections. A PersistenceManager instance is required for every operation on a database. It may represent either a remote connection to a remote database server (in client server mode) or a local connection to a local database file (in embedded mode). The functionality in both cases is the same.

PersistenceManager instances are usually obtained for short term use (for example, per web request in a web application). In some cases, however, a single PersistenceManager instance can be used during the entire application run. This may be a good option in a single-user desktop application using an embedded ObjectDB database (for example, a personal organizer). In multi-user applications (using client server mode to connect to the database), short term connections are preferred because every open connection consumes essential server resources.

### Obtaining a Database Connection

Given a PersistenceManagerFactory instance pmf, a PersistenceManager instance can be obtained by:

```
PersistenceManager pm = pmf.getPersistenceManager();
```

If the connection requires username and password other than the PersistenceManagerFactory's username and password (irrelevant when using embedded mode), new values can be specified:

```
PersistenceManager pm = pmf.getPersistenceManager(userName, password);
```

The settings of the PersistenceManagerFactory at the time of this operation are used to initialize the new PersistenceManager instance. The PersistenceManagerFactory's connection url specifies the location of the database. Using an ObjectDB extension, a PersistenceManager instance can also be obtained directly, without a PersistenceManagerFactory instance:

```
PersistenceManager pm = Utilities.getPersistenceManager("local.odb");
```

**Note:** These operations can be used for both opening a database and creating a new one. If a database file is not found at the specified connection url, ObjectDB tries to **create a new, empty database** at that location. This behavior is an ObjectDB extension. JDO does not define a standard method for creating a new database.

### Closing a Database Connection

A typical short term database connection has the following structure:

```
PersistenceManager pm = pmf.getPersistenceManager();
try {
  // Operations on the database should come here.
}
finally {
  if (pm.currentTransaction().isActive())
    pm.currentTransaction().rollback();
  if (!pm.isClosed())
    pm.close();
}
```

The finally block ensures database connection closure (and rolling back an active transaction if any) regardless of whether the operations succeeded or threw an exception.

### Exception Handling

The code fragment above, as with any other code that uses JDO, may throw instances of JDOException (and its subclasses) as exceptions. Therefore, it is important to wrap JDO method calls in a try-catch block:

```
try {
  // The code above, containing database operations, should come here.
}
catch (JDOException x) {
  // Error handling code should come here.
}
```

Because JDOException is a subclass of RuntimeException the compiler does not force a throws declaration if code that uses JDO is not wrapped by a try-catch block (unlike code working with files and I/O, for example, in which handling a checked exception, such as IOException, is forced by the compiler). Therefore, extra caution is required by the developer to make sure that at some level a proper try-catch block exists.

More details on the JDO exception hierarchy can be found at:

   http://java.sun.com/products/jdo/javadocs/.

## 5.3.  javax.jdo.Transaction

The javax.jdo.Transaction interface is used to represent and manage transactions on a database. In JDO, operations that affect the content of the database (store, update, delete) must always be performed within an active transaction. On the other hand, reading from the database does not require an active transaction unless the NontransactionalRead mode is off, as explained below.

### Working with Transactions

The code from the section above is now adjusted to work with a transaction:

```
PersistenceManager pm = pmf.getPersistenceManager();
try {
  pm.currentTransaction().begin();
  // Operations that modify the database should come here.
  pm.currentTransaction().commit();
}
finally {
  if (pm.currentTransaction().isActive())
    pm.currentTransaction().rollback();
  if (!pm.isClosed())
    pm.close();
}
```

Every PersistenceManager instance holds a reference to an attached Transaction instance (i.e. there is a one to one relationship between these two instances). The currentTransaction() method can be called on any PersistenceManager instance, in any context, to return its associate Transaction object. The strong one to one relationship implies that, theoretically, transaction management could be done directly in PersistenceManager. But, the designers of JDO preferred to define a separate interface for managing transactions.

A transaction is started by a call to begin() and ended by a call to commit() or rollback(). All the operations on the database within these boundaries are associated with that transaction and are kept in memory until the transaction is ended. If the transaction is ended with rollback(), all the modifications to the database are discarded. On the other hand, ending the transaction with commit() propagates all the database modifications physically to the database. If for any reason a commit() fails, the transaction is rolled back automatically (including rolling back modifications that have already been propagated to the database prior to the failure) and an exception is thrown. The type of the exception is a JDOException subclass that reflects the specific error that occurred.

A transaction is always expected to be atomic, so a situation in which only some of the changes are applied to the database is not acceptable. Before commit() starts writing to the database it stores the changes in a related temporary file in the same directory as the database file. For instance, a database file named **db.odb**, would have a corresponding temporary file named **.$db.odb$**. After the transaction is completed the temporary file is marked as obsolete (and for efficiency, it is deleted only when the database is closed). If the system crashes during a commit() (e.g. power failure), and before the changes are able to be completely written to the database, the transaction is expected to be completed, using the temporary file, the next time the database is opened. This feature, which is called *auto recovery*, is not supported by the free edition of ObjectDB. To ensure database consistency, it is recommended that write caching be disabled - at least for the database file and the recovery temporary file. Consult your operating system documentation for instructions on how to accomplish this.

It is mainly a matter of style whether or not to use a separate variable to hold the Transaction instance. The following code is equivalent to the code above:

```
PersistenceManager pm = pmf.getPersistenceManager();
Transaction tr = pm.currentTransaction();
try {
  tr.begin();
  // Operations that modify the database should come here.
  tr.commit();
}
finally {
  if (tr.isActive())
    tr.rollback();
  if (!pm.isClosed())
    pm.close();
}
```

## Automatic Lock Management

ObjectDB manages an automatic lock mechanism in order to prevent a database object from being modified by two different users (which are represented by two different PersistenceManager instances) at the same time. In an active transaction, on every retrieval of an object from the database a READ lock on that object is obtained by the PersistenceManager. On every attempt to modify a persistent object, within an active transaction, a WRITE lock is obtained on that object by the PersistenceManager. Multiple READ locks of different PersistenceManager instances on the same object at the same time are allowed, but a WRITE lock is expected to be exclusive. Therefore, any attempt to modify an object that is currently in use by some other PersistenceManager instance (for read or write), as well as any attempt to read an object that is currently being modified by some other PersistenceManager, blocks the thread until the previous lock is released. When a transaction ends, all the locks held by that transaction are automatically released.

To avoid a deadlock, threads do not wait for a lock forever. When a lock request cannot be granted, the requesting thread waits 50 milliseconds and then makes another attempt to obtain the lock. After 40 retries (about 2 seconds) a JDOUserException (a subclass of JDOException) is thrown. These constants (50 milliseconds and 40 retries) can be changed using System properties (before the database is opened) as so:

```
// Maximum 10 tries before throwing an exception:
System.setProperty("com.objectdb.lock.retry.max", "10");

// Wait 100 milliseconds between every two tries:
System.setProperty("com.objectdb.lock.retry.wait", "100");
```

Transactions that use the lock mechanism described above are called, in JDO, *datastore transactions*. This type of transaction is used in ObjectDB by default, but ObjectDB also supports another type of transaction called *optimistic transactions*. When using optimistic

transactions, objects are not locked. Therefore, two transactions may update the same persistent object at the same time, and conflicts are detected at commit time. If a conflict happens, the first committed transaction succeeds while the second fails, throws an exception, and is rolled back automatically.

Optimistic transactions may be a good choice when long running transactions are required. This would avoid the excessively long object locking periods that datastore transactions may cause. On short running transactions, however, or when the probability for lock conflicts is high, datastore transactions are preferred.

## JDO Transaction Flags

Transaction modes can be set at PersistenceManagerFactory level (as shown in <u>section 5.1</u>), or at the PersistenceManager or Transaction levels, using getter and setter methods:

```
// Using the setter methods:
pm.currentTransaction().setOptimistic(true);
pm.currentTransaction().setNontransactionalRead(false);
pm.currentTransaction().setNontransactionalWrite(true);
pm.currentTransaction().setRetainValues(false);
pm.currentTransaction().setRestoreValues(true);

// Using the getter methods:
boolean flag1 = pm.currentTransaction().getOptimistic();
boolean flag2 = pm.currentTransaction().getNontransactionalRead();
boolean flag3 = pm.currentTransaction().getNontransactionalWrite();
boolean flag4 = pm.currentTransaction().getRetainValues();
boolean flag5 = pm.currentTransaction().getRestoreValues();
```

Here is a short explanation of each one of these modes:

**javax.jdo.option.Optimistic**

Indicates whether optimistic locking (optimistic transaction) or pessimistic locking (datastore transaction) is used. The default in ObjectDB is false, indicating that transactions are datastore transactions by default.

**javax.jdo.option.NontransactionalRead**

Indicates whether or not database operations that do not modify the database content (queries, object retrieval, reading) can be done without an active transaction. The default in ObjectDB is true, indicating that only operations that modify the database require an active transaction. A false value enforces the use of an active transaction when reading from the database.

**javax.jdo.option.NontransactionalWrite**

Indicates whether or not persistent objects (i.e. objects that represent database objects) can be modified when no transaction is active. The default in ObjectDB is false, indicating that any

attempt to modify a persistent object without an active transaction is not allowed and will cause a a JDOUserException (a subclass of JDOException) to be thrown. A true value enables modifying persistent objects without an active transaction, but in this case changes are only performed in memory (they are never propagated to the database).

**javax.jdo.option.RetainValues**

Indicates whether or not persistent objects preserve their content after transaction commit. The default in ObjectDB is true, indicating that persistent object fields are the same before and after commit(). A false value causes persistent objects to become hollow on commit, causing all persistent fields to be assigned default values. The content of a hollow object is automatically reread from the database by JDO at the time the application tries to access it (unless NontransactionalRead is false and there is no active transaction - in which case an exception is thrown). Usually the only difference between the two modes is in performance, where the default true value is expected to be slightly more efficient.

**javax.jdo.option.RestoreValues**

Indicates whether or not, on transaction rollback, persistent objects are restored to their original content at the beginning of the transaction. The default in ObjectDB is false, indicating that persistent objects become hollow on rollback(). In this case, their content is reread from the database by JDO only when the application tries to access them again. A true value causes persistent objects to be restored to their original value at the time of transaction rollback. Usually the only difference between the two modes is in performance. If rollback() is not so common, turning this flag off (using the default false value) is expected to be more efficient.

# Chapter 6.  Persistent Objects

This chapter describes how to use ObjectDB to manage database objects:

Shows how to store new objects in the database.

Explains how objects in the database can be identified.

Shows various ways of retrieving objects from the database.

Shows how to modify objects that are already in the database.

Shows how to delete objects from the database.

Discusses the different states of in-memory objects in a JDO application.

## 6.1.  Making Objects Persistent

In JDO applications, every object in memory is either a persistent object, i.e. represents some database content, or a transient object, i.e. not related to any database.

### Storing Objects Explicitly

When a new object is constructed by the new operator, it always starts as a transient object, regardless of its type. Instances of persistent classes can become persistent later, as demonstrated by the following code:

```
pm.currentTransaction().begin();

Person person = new Person("George", "Bush");
Address address = new Address("White House");
person.setAddress(address);
pm.makePersistent(person);

pm.currentTransaction().commit();
```

First, the Person instance is constructed as a simple transient object. Then it becomes persistent by an explicit call to the makePersistent(...) method. When a transient object becomes persistent during an active transaction, its content is automatically stored in the database when the transaction is committed (unless it is being deleted before commit). An attempt to call makePersistent(...) when there is no active transaction throws an exception, because operations that modify the database require an active transaction. An exception is

also thrown if the argument to makePersistent(...) is not an instance of a persistent class (as defined in chapter 3).

Only the call to makePersistent(...) has to be within an active transaction. Therefore, the following code is legal and equivalent to the code above:

```
Person person = new Person("George", "Bush");
Address address = new Address("The White House");
person.setAddress(address);

pm.currentTransaction().begin();
pm.makePersistent(person);
pm.currentTransaction().commit();
```

Multiple objects can become persistent, either by multiple calls to makePersistent(...), or by passing a collection or array of objects to makePersistentAll(...), which is another method of PersistenceManager. The makePersistentAll(...) method stores all the elements in the specified collection or array in the database, but not the collection or array itself (unlike passing a collection to Utilities.bind(...)).

## Persistence By Reachability

The Address instance in the code above also starts out as a transient object. It, however, becomes persistent, not by an explicit call to makePersistent(...), but rather, because of the *persistence by reachability* rule. According to this JDO rule, every transient object that, at commit time, is reachable from a persistent object (using persistent fields) also becomes persistent, i.e. its content is also stored in the database. This mechanism prevents broken references in the database and simplifies development.

The JDO specification limits the use of makePersistent(...) to instances of user defined persistent classes. However, persistence by reachability is applicable to all persistent types, including system types, such as String and ArrayList (a complete list of persistent types is available at section 3.2). ObjectDB provides an additional storing method, Utilities.bind(...). This method (discussed in the next section) enables the assignment of names to objects in the database and the storing of instances of any persistent type explicitly, including system types.

## Embedded Objects

An embedded object is an object that is stored in the database as part of a containing object. Objects stored explicitly by makePersistent(...) are never embedded. Objects stored as a result of persistence by reachability can be embedded. By default, system types are embedded and user defined types (like the Address instance above) are not. This behavior can be changed as explained in section 4.3

Embedded objects can reduce storage space and improve efficiency, but they also have some limitations. An embedded object cannot have an object ID. An embedded object cannot be

shared by references from multiple objects. In addition, embedded objects of persistent classes are not included in the extents of their classes, so they cannot be iterated or queried directly (an embedded object can only be retrieved using a reference from its containing object).

## 6.2.  Object IDs and Names

Identifying database objects by unique IDs and by names is useful in object retrieval, as shown in the next section (). ObjectDB supports two identifying methods for database objects. The first method, object IDs, is supported by JDO. The second method, object names, is not supported by JDO, but ObjectDB supports it as an extension because of its popularity in object databases. Both methods are only supported for non embedded objects.

### Object IDs

ObjectDB assigns a unique numeric ID to every persistent object when it is first stored in the database. The ID of the first object in the database is always 1, the ID of the second object is 2, and so on. Once assigned, the object ID cannot be modified or replaced. It represents the object as long as the object exists in the database. If an object is deleted from the database, its unique ID is never used again to represent another object. The JDO specification refers to object IDs assigned automatically by the JDO implementation as datastore identity. Application identity, in which the application assigns object IDs to objects, is not supported by ObjectDB.

Accessing the object ID of a persistent object can be done as follows:

```
Object oid = pm.getObjectId(obj);
```

Or by a static method (useful when a PersistenceManager is unavailable):

```
Object oid = JDOHelper.getObjectId(obj);
```

JDO does not define a specific type for object IDs, and different JDO implementations may use different types. Therefore, a JDO portable application should manage Object IDs using java.lang.Object references. You can assume that whatever type is used by a given JDO implementation, it implements the Serializable interface and it has a meaningful toString() method. The toString() method of object IDs in ObjectDB returns the numeric ID of the object as a string. The ability to convert Object IDs to strings, and the ability to convert strings back to object IDs (as shown in the next section), make object IDs very useful for representing objects in web applications (in which parameters are represented by strings).

If getObjectID(...) is called with a transient parameter, it returns null. If it is called with a new persistent object that has not been stored in the database yet, an incomplete object ID is

returned, and its toString() returns a temporary negative number, because a database object ID is not allocated yet. Such an object ID is fixed on commit(). For example:

```
pm.currentTransaction().begin();
Person person = new Person("George", "Bush");
pm.makePersistent(person);
Object oid = pm.getObjectId(person);
String s1 = oid.toString(); // s1 contains a temporary negative number
pm.currentTransaction().commit();
String s2 = oid.toString(); // s2 contains a permanent positive number
```

The above code shows that calling toString() on the same object ID before and after commit() returns different values. Using temporary IDs instead of allocating permanent object IDs immediately on makePersistent(...) improves efficiency because the database only has to be accessed once, as part of the commit().

## Object Names

ObjectDB supports assigning names to specific persistent objects, as shown in the following code:

```
import com.objectdb.Utilities;
    :
    :
  pm.currentTransaction().begin();
  Utilities.bind(pm, person, "me");
  pm.currentTransaction().commit();
```

The Utilities.bind(...) method is very similar to the makePersistent(...) method, and can only be called within an active transaction. When the transaction is committed, the object is stored in the database with the specified name. If the object is already in the database, the new name is assigned to it but its content is only updated if it has changed since the last store. An object name must be unique in the database. The bind(...) method throws a JDOUserException if the name is already in use.

In addition to the naming issue, there is another difference between makePersistent(...) and bind(...). The bind(...) method is not restricted to only storing instances of persistent classes, but can directly store instances of **any** persistent type in the database. A complete list of persistent types is shown in <u>section 3.2</u>.

The Utilities.unbind(...) method removes an object name:

```
import com.objectdb.Utilities;
    :
    :
  pm.currentTransaction().begin();
  Utilities.unbind(pm, "me");
  pm.currentTransaction().commit();
```

Removing a name also requires an active transaction. When the transaction is committed, the name is removed from the database. Notice that unbind(...) is not exactly an inverse operation of bind(...), because the object is not deleted from the database, just its name is removed.

Objects with names are called *root objects* in object databases, because navigation in the database can easily start from them, as explained in the next section.

## 6.3.  Retrieving Persistent Objects

There are various ways to retrieve objects from an ObjectDB database. Every object can be retrieved directly by using its object ID. A root object can also be retrieved by its name. It is also possible to use an Extent instance to iterate over all the objects of a specific class, retrieving them from the database one by one. If more selective retrieval is needed, a Query can be used to retrieve only objects that satisfy some well defined criteria. In addition, after an object is retrieved, all the objects that are reachable from it can be retrieved simply by navigating through the persistent fields (this is part of the *transparent persistence* feature of JDO). No matter which way an object is retrieved from the database, its no-arg constructor is used to construct a memory object to represent it. Therefore, for efficiency sake, it is recommended to avoid time consuming operations in no-arg constructors of persistent classes, and to keep them as simple as possible.

### The Persistent Object Cache

As noted before, persistent objects are memory objects that represent some content in the database. When an object is retrieved from the database, no matter which method is used, ObjectDB has to return a persistent object representing that database object. Every PersistenceManager instance manages a cache of persistent objects that are in use by the application. If the cache already contains a persistent object representing the requested database object, that object is returned. If an object is not found in the cache, a new object is constructed, filled with the content of the database object, added to the cache and finally returned to the application. Persistent objects that have not been modified or deleted during a transaction are held in the cache by weak references. Therefore, when a persistent object is no longer in use by the application the garbage collector can discard it, and it is automatically removed from the cache.

The main role of the persistent object cache is to make sure that, within the same PersistenceManager, a database object is represented by no more than one memory object. Every PersistenceManager instance manages its own cache and its own persistent objects. Therefore, a database object can be represented by different memory objects in different PersistenceManager instances. But retrieving the same database object more than once using the same PersistenceManager instance should always return the same memory object. In

addition to the object cache that is required by JDO, ObjectDB manages another cache containing database pages, in order to improve efficiency. Unlike the object cache, which is always on the client side, the page cache is located on the server side in client-server mode.

## Retrieval by an Object ID

If an object ID instance, oid, is available, an object can be retrieved simply by:

```
Person person = (Person)pm.getObjectById(oid, false);
```

The second argument of getObjectById(...) indicates whether or not the retrieved object has to be verified against the database, in case it is already found in the cache and it is unclear if its content in memory is up to date. If true is specified and the object in database is different, the cached object is refreshed from the database, and only then it is returned.

A more common scenario is starting with a string representation of an Object ID (obtained once by calling toString() on a real object ID instance), rather than an actual object ID instance. This is a classic case in web applications, in which passing string values between web pages is easier than passing real object ID instances. In this case an additional step is needed to build the object ID instance from a string str that represents it:

```
Object oid = pm.newObjectIdInstance(Person.class, str);
Person person = (Person)pm.getObjectById(oid, false);
```

Notice that the newObjectIdInstance(...) method requires not only an object ID string, but also a Class instance. ObjectDB ignores the first argument. Therefore, a null value can also be specified, but in order to keep the application JDO portable, a proper Class should be specified. The class can also be encoded in an object ID string, using:

```
String str = JDOHelper.getObjectId(obj) + '@' + obj.getClass();
```

In this case the object can be retrieved later using str by:

```
String[] s = str.split("@");
Object oid = pm.newObjectIdInstance(Class.forName(s[0]), s[1]);
Person person = (Person)pm.getObjectById(oid, false);
```

## Retrieval by an Object Name

A root object, which is a database object with an assigned name, can be retrieved by using its name. For example, a root with the name "me" can be retrieved with:

```
Person person = (Person)pm.getObjectById("me", false);
```

This is the same getObjectById(...) method that is used to retrieve objects by their object IDs, but when the first argument is a String instance, it is considered an object name. As already noted, object names are supported by ObjectDB but not by JDO.

### Retrieval by an Extent

An Extent enables iteration over all the persistent instances of a specific persistent class (excluding embedded objects). The objects are retrieved one by one:

```
Extent extent = pm.getExtent(Person.class, false);
java.util.Iterator itr = extent.iterator();
try {
  while (itr.hasNext())
  {
    Person person = (Person)itr.next();
    System.out.println(person.getName());
  }
}
finally {
  extent.close(itr); // closes a specified iterator
  //extent.closeAll(); // closes all active iterators
}
```

An Extent instance is obtained by the PersistenceManager's getExtent(...) method. The persistent class for which the Extent is required is specified by the first argument. The second argument specifies whether or not persistent instances of subclasses should also be included (true indicates that subclasses are required).

Iteration over an Extent is similar to iteration over any ordinary Java collection using an Iterator instance. Each call to next() retrieves another instance of the class from the database, until hasNext() returns false, which indicates there are no more instances. In ObjectDB (but not necessarily in other JDO implementations) the iteration order is from the oldest object to a newest, i.e. from a lower object ID to a higher one.

Unlike an iterator over an ordinary Java collection, an iterator over an Extent should be closed at the end of its use. This is mainly important in client server mode in which an active iterator might consume resources in the remote server. The Extent's close(...) method closes a specified iterator. Alternatively, all the active iterators on an Extent can be closed by the Extent's closeAll() method (the commented line in the code above).

### Retrieval by Query

The most advanced method for retrieving objects from the database is to use queries. The official query language of JDO is JDOQL (JDO Query Language). It enables retrieval of objects from the database using simple queries as well using complex, sophisticated queries. The results can also be sorted by specified order expressions. Chapter 7 is devoted to JDOQL queries.

### Retrieval by Access

When an object is retrieved from the database (in whatever manner) all its persistent fields can be accessed freely. This includes fields that reference other persistent objects, which may

or may not have not been loaded from the database yet (and may be seen as null values in the debugger). When an access to another persistent object is made, the object is retrieved automatically by ObjectDB from the database if it is not already present. From the point of view of the developer, it looks like the entire graph of objects is present in memory for his or her use. This illusion is part of the *transparent persistence* feature of JDO, which hides some of the direct interaction with the database, in order to make database programming easier.

For example, after retrieving a Person instance from the database, its address field can be accessed, causing an Address instance to be loaded from the database automatically:

```
Person person = (Person)pm.getObjectById(oid, false);
Address address = person.address; // an Address instance is retrieved
```

To enable this automatic retrieval, special JDO code has to be added to every class in which persistent fields are accessed or modified directly. The addition of this code to persistent classes, along with the addition of the implementation code for the PersistenceCapable interface, is done automatically by the JDO enhancer. Non persistent classes that directly access persistent fields of other classes are referred to as *persistence aware*. Persistence aware classes must also be run through the JDO enhancer, which enhances them to include the code needed for transparent persistence. But, because these classes are not described in the JDO metadata files, the enhancer identifies them as not persistent and does not enhance them with the code that implements the PersistenceCapable interface. Persistence aware classes are discussed in more detail in section 3.3.

### Refresh and Retrieve

The PersistenceManager interface includes some methods for special retrieval operations. The retrieve(...) method is used to load all the fields of a specified persistent object that have not been loaded yet, without waiting for them to be accessed. A collection or array of persistent objects can be passed to retrieveAll(...), for the same purpose. The refresh(...) method is used to reload an object from the database. If the object was modified since it was previously loaded, and the changes were not committed yet, they are discarded. Multiple objects can be refreshed, either by multiple calls to refresh(...), or by a single call to refreshAll(...).

### Non Transactional Persistent Objects

Because NonTransactionalRead is enabled in ObjectDB by default, objects can also be retrieved without an active transaction. Persistent objects outside an active transaction are referred to as *non transactional* persistent objects. By default, these objects are read-only and a JDOUserException is thrown on any attempt to modify them (unless NonTransactionalWrite is enabled as explained in section 5.3).

## 6.4.  Modifying Persistent Objects

Modifying existing database objects using JDO is straightforward, because it is also based on transparent persistence. After retrieving a persistent object from the database (no matter which way), it can simply be modified in memory while a transaction is active. When the transaction is committed, the content of that object in the database is physically updated:

```
pm.currentTransaction().begin();
Person person = (Person)pm.getObjectById(oid, false);
person.address = new Address("Texas"); // persistent field is modified
pm.currentTransaction().commit();
```

If, during a transaction, a transient object becomes reachable from a persistent object (as the new Address instance above), it is also stored in the database as a new persistent object, following the persistence by reachability rule. Of course, if the transaction is ended with rollback() and not with commit(), all the modifications are discarded, and nothing is propagated to the database. It is important to note in the code above that an old Address instance, if there were one, would not be deleted automatically. [Section 6.5](#) discusses persistent object deletion.

When NonTransactionalRead is enabled (the default in ObjectDB), only the **modification** of the object is required to occur within an active transaction. The object itself can be retrieved before the transaction begins:

```
Person person = (Person)pm.getObjectById(oid, false);

pm.currentTransaction().begin();
person.address = new Address("Texas");
pm.currentTransaction().commit();
```

However, if possible, it is better to retrieve an object for modification in the same transaction in which it is modified because this is slightly more efficient.

Notice that any code that modifies persistent fields must be enhanced, either as persistence capable or as persistence aware. Otherwise, modifications cannot be tracked by ObjectDB, and at transaction commit, cannot be propagated to the database.


## 6.5.  Deleting Persistent Objects

In order to delete an object from the database, it must first be retrieved (no matter which way), and then, when a transaction is active, it can be deleted using the deletePersistent(...) method:

```
pm.currentTransaction().begin();
Person person = (Person)pm.getObjectById(oid, false);
pm.deletePersistent(person); // object is marked for deletion
pm.currentTransaction().commit(); // object is physically deleted
```

The object is physically deleted from the database when the transaction is committed. If the transaction is rolled back and not committed, the object is not deleted. Only the call to deletePersistent(...) must be made within an active transaction. The object can be retrieved before the transaction begins (even though it is slightly less efficient).

Multiple objects can be deleted, either by multiple calls to deletePersistent(...), or by passing a collection or an array of objects to deletePersistentAll(...), which is also a method of PersistenceManager.

### Deleting Dependent Objects

When an object is deleted from the database, all the embedded objects that it contains are also deleted automatically. Non embedded objects referred to by the deleted object are not deleted with it, so if they are not needed anymore they have to be deleted explicitly. An elegant way to delete non embedded dependent objects is to implement the InstanceCallback interface, as demonstrated in section 3.4.

Usually, instances of system persistent types are embedded by default, while instances of user defined persistent classes are not (more details are provided in section 4.3 and section 4.4). Therefore, by default, String and Date fields, as well as collection fields (such as ArrayList) are deleted automatically when their containing object is deleted. When possible, it is useful to define dependent objects of user defined classes as embedded. In addition to being simpler to delete, embedded objects are usually more efficient in terms of execution time and database space. However, because of the limitations of embedded objects (as discussed in section 6.1), using them is not always an option.

### Database Garbage Collector

The database garbage collector is a tool that scans an ObjectDB database, locates unreachable objects, and deletes them from the database. Instances of a persistent class with Extent support (excluding embedded objects) are always considered reachable. Root objects (objects with names) are also always reachable. In addition, every object that can be accessed by navigation from a reachable object using persistent fields is also reachable. On every run of the garbage collector, all the unreachable objects are located and deleted.

Unlike the Java memory garbage collector, execution of the database garbage collector requires an explicit call. It can be run in the Database Explorer, as discussed in chapter 9, or by the application:

```
import com.objectdb.Utilities;
    :
    :
  Thread thread = Utilities.startGarbageCollector(pm);
```

Only a single database garbage collector thread can run at any given time. But, the database garbage collector does not require exclusive access to the database and can run while the database is also being used by other processes or threads. The Thread instance returned from Utilities.startGarbageCollector(...) may be used to manipulate the garbage collector thread (i.e. to change priority, to wait until it finishes, and so on).

Even if you do not use the database garbage collector for production, it may be useful for debugging and testing. If the garbage collector locates unreachable database objects, the application can be fixed to delete these objects before they become unreachable. JDO portable applications should not use the garbage collector as it is an ObjectDB extension and probably not supported in other JDO implementations.

Note: The database garbage collector is not supported by the ObjectDB free edition.

## 6.6.  Object States

Objects in a JDO application are divided into persistent objects, which represent database objects, and transient objects, which are ordinary memory objects not related to any database. States of persistent objects are distinguished by several factors. This section contains a brief description of object states in JDO. More details can be found in any book on JDO.

### Examining Object States

The state of an object in a JDO application can be examined using five JDOHelper static methods:

```
boolean b1 = JDOHelper.isPersistent(obj);
boolean b2 = JDOHelper.isTransactional(obj);
boolean b3 = JDOHelper.isNew(obj);
boolean b4 = JDOHelper.isDeleted(obj);
boolean b5 = JDOHelper.isDirty(obj);
```

All of these methods return false if obj is not persistent (i.e. a transient object or null).

If obj is persistent, i.e. isPersistent(obj) returns true, the other methods provide more details:

**JDOHelper.isTransactional(obj)**

Returns true if obj is *transactional*, i.e. if obj is associated with an active transaction. Every object that becomes persistent or retrieved from the database during an active transaction is transactional. Objects retrieved from the database when no transaction is active are non transactional. Transactional objects become non transactional automatically when the transaction is ended. Non transactional objects become transactional if they are accessed or modified when a transaction is active.

**JDOHelper.isNew(obj)**

Returns true if obj is a new persistent object of the current transaction, and not yet stored in the database.

**JDOHelper.isDeleted(obj)**

Returns true if obj has been physically deleted from the database or if obj has been marked for deletion in the current transaction using deletePersistent(...) and it is expected to be deleted at transaction commit.

**JDOHelper.isDirty(obj)**

Returns true if the state or the content of obj has been changed in the current transaction. Particularly, it returns true if isNew(obj) or isDeleted(obj) return true, and also if one of the persistent fields of obj has been modified.

## Object States and Locks

As explained in section 5.3, ObjectDB manages an automatic lock mechanism in order to prevent two or more different PersistenceManager instances from modifying the same object at the same time. When using data store transactions (which is the default in ObjectDB), there is a strong relationship between object states and object locks.

Only persistent transactional objects that are not new are locked. Therefore, a persistent object obj is locked only when the following expression evaluates to true:

```
JDOHelper.isPersistent(obj) &&
JDOHelper.isTransactional(obj) &&
!JDOHelper.isNew(obj)
```

When a JDOHelper.isDirty(obj) is true a WRITE lock is used and when it is false a READ lock is used. When the state of the object is changed (for example a persistent object is modified) an automatic action is taken by ObjectDB to update the object lock (for example from READ to WRITE). Locks are released automatically when the transaction is ended because, at that point, all the objects have become non transactional. If these objects are later accessed in a new transaction, they are then locked again because they become transactional.

## Changing Object States

An object's state can change either implicitly or explicitly. Implicit changes occur when its fields are accessed or modified, or when the transaction associated with it is ended. An objects state is changed explicitly when the object is passed as an argument to a method like makePersistent(...) or deletePersistent(...). The PersistenceManager interface includes other methods for changing object states, which are not discussed in this guide, because in most applications they are rarely used.

Among them:

- evict(...) and evictAll(...)

- makeTransactional(...) and makeTransactionalAll(...)

- makeNonTransactional(...) and makeNonTransactionalAll(...)

- makeTransient(...) and makeTransientAll(...)

The last two methods, for instance, can change the state of one or more persistent objects to transient. A persistent object that becomes transient is not managed by JDO anymore, and it is immediately removed form the persistent object cache. More details about these methods can be found books on JDO.

# Chapter 7.  JDOQL Queries

There are various ways to retrieve objects from an ObjectDB database, as shown in section 6.3. An Extent, for instance, can be used to retrieve all the instances of a specified persistent class. When a more selective retrieval is needed, JDOQL (JDO Query Language) is used. JDOQL for JDO is like SQL for RDBMS. It provides object retrieval from the database according to a specified selection criteria, and iteration over the results in a specified order.

This chapter contains the following sections:

## 7.1.  Introduction to JDOQL

A basic JDOQL query has the following three components:

- A candidate collection containing persistent objects (usually an Extent)
- A candidate class (usually a persistent class)
- A filter, which is a boolean expression in a Java like syntax

The query result is a subset of objects from the candidate collection that contains only the instances of the candidate class satisfying the given filter. In addition to these three main components, queries can include other optional components, such as parameters, variables and import and ordering expressions, which are discussed later in this chapter.

### A First Query

The following query retrieves all the people whose age is 18 or older:

```
Query query = pm.newQuery(Person.class, "this.age >= 18");
Collection result = (Collection)query.execute();
```

Queries are represented by the javax.jdo.Query interface. A Query instance is constructed by one of the PersistenceManager's newQuery(...) methods. For example, in the query above, the candidate class is Person and the filter is "this.age >= 18". When a candidate collection is not specified explicitly, as it is in this query, the entire Extent of the candidate class is used, and

the candidate collection contains all the non embedded instances of the candidate class. In such cases, if an Extent is not managed for the candidate class, the query is not valid.

The execute() method compiles and runs the query. If there is no age field in class Person or if the field exists but its type cannot be compared with an int value, a JDOUserException is thrown. If the query compilation succeeds, the Extent of Person instances in the database is iterated object by object, and the filter is evaluated for every Person instance. The this keyword in the filter represents the iterated object. Only instances for which the evaluation of the filter expression is true are included in the result collection. If an index is defined for the age field (see section 4.5), iteration over the objects in the database would not be required, and the query would execute much faster. The execute() method returns a Collection instance, but its declared return type is Object (for future JDO extensions), so casting is required.

Compilation and execution of queries can also be separated into two commands:

```
Query query = pm.newQuery(Person.class, "this.age >= 18");
query.compile();
Collection result = (Collection)query.execute();
```

The compile() method checks the syntax of the query and prepares it for execution without executing it. In most cases, an explicit call to compile() is not needed, because when execute() is invoked it automatically compiles the query, if it has not been done so already.

## The Result Collection

The query result is represented by a java.util.Collection instance and managed like any ordinary Java collection. For example, the number of retrieved objects can be obtained using the size() method:

```
int count = result.size();
```

Iteration over the result collection is done by a java.util.Iterator instance:

```
Iterator itr = result.iterator();
while (itr.hasNext())
  System.out.println(itr.next());
```

Similarly, other methods of java.util.Collection can be used on the result collection. There is, however, one important difference between the result collection and other Java collections. Rather than relying on the garbage collector to cleanup when the collection is no longer used, a result collection has to be closed explicitly. This is especially important in client server mode because the result collection on the client side may hold resources on the server side. The closing is performed using methods of the Query instance:

```
// Close a single result collection obtained from this query:
query.close(result);
```

```
// Close all the result collections obtained from this query:
query.closeAll();
```

The complete code for printing all the people whose age is 18 or older (using the toString() method of Person) might have the following structure:

```
Query query = pm.newQuery(Person.class, "this.age >= 18");
Collection result = (Collection)query.execute();
try {
  Iterator itr = result.iterator();
  while (itr.hasNext())
    System.out.println(itr.next());
}
finally {
  query.close(result);
}
```

The finally block ensures result collection closing regardless of whether the operations succeeded or threw an exception. An exception is thrown on any attempt to use a closed result collection.

## Query Construction

In most queries, the Extent of the candidate class is used as the candidate collection . A Query instance for this type of query can be obtained by one of the following two newQuery(...) forms:

```
Query newQuery(Extent candidates, String filter)
Query newQuery(Class cls, String filter)
```

In the first form, the candidate class is automatically set to the class of the specified candidate Extent, and in the second form, the candidate collection is automatically set to the Extent of the specified candidate class (which also covers subclasses). The first form is slightly more flexible because it enables using an Extent with or without subclass support.

There is another form of newQuery(...) for a case in which the candidate collection is a regular Java collection, rather than an Extent:

```
Query newQuery(Class cls, Collection candidates, String filter)
```

Querying a collection is less common than querying an Extent. It is useful for filtering a collection in memory, and for requerying the result of a previous JDOQL query.

A query can also be executed without a filter. The result of such a query contains all the candidate objects that are instances of the candidate class. This may be useful, for instance, to obtain the number of objects of a specific class (an operation that is not supported directly by

an Extent). The following forms of newQuery(...) are equivalent to the forms described above, but without a filter:

```
Query newQuery(Extent candidates);
Query newQuery(Class cls);
Query newQuery(Class cls, Collection candidates);
```

A Query instance can even be constructed by an empty newQuery() form:

```
Query newQuery();
```

However, a candidate collection must be provided before query execution (explicitly or implicitly by specifying a candidate class), otherwise an execution of the query throws a JDOUserException. Query components that are not specified when invoking one of the PersistenceManager interface newQuery methods can be assigned later using methods in the Query interface itself:

```
void setClass(Class cls)
void setCandidates(Collection pcs)
void setCandidates(Extent pcs)
void setFilter(String filter)
```

## 7.2.  Query Filter Syntax

A query filter is a string containing a boolean expression in a Java like syntax. It has to be valid in the scope of the candidate class. For example, "this.age >= 18" is a valid filter if the candidate class contains a field with the name age and a type comparable to int. This section describes all the elements that can be used in a JDOQL query filter, except for parameters and variables that are discussed later in this chapter.

### Literals

All types of Java literals are supported by JDOQL, as demonstrated by the following table:

| Literal Type | Samples of valid literals in JDOQL |
|---|---|
| int | 2003, -127, 0, 0xFFFF, 07777, ... |
| long | 2003L, -127L, 0L, 0xFFFFL, 07777L, ... |
| float | 3.14F, 0f, 1e2f, -2.f, 5.04e+17f, ... |
| double | 3.14, 0d, 1e2D, -2., 5.04e+17, ... |
| char | 'x', '@', '\n', '\\', '\'', '\uFFFF', ... |
| string | "", " ", "abcd\n1234", ... |
| boolean | true, false |
| reference | null |

As shown in the next section ([section 7.3](#)), parameters could be used instead of constant literals to make queries more generic. Parameter values are provided when the query is executed so that the same query can be executed with different parameter values.

### The 'this' Keyword

During query evaluation, each object in the candidate collection that is an instance of the candidate class is evaluated by the filter. The evaluated object is represented in the filter by the keyword this, whose type is the candidate class. Usually this is used for accessing fields of the candidate object, but it can also be used to reference a candidate object in other operations, such as method calls and comparison expressions.

### Fields

Persistent fields of the candidate class play an important role in query filters. For instance, if a candidate class has a persistent field, verified, of boolean type, the expression "this.verified" is a valid query filter. It selects all the objects with the true value in that field. Other types of persistent fields can participate in queries in combination with comparison expressions, as in "this.age >= 18". The this. prefix can be omitted in Java, therefore, "verified" and "age >= 18" are also valid filter expressions. Field accessing is not limited to the this reference. Persistent fields of any entity that represents a persistent object can be accessed. This includes fields of parameter objects and variable objects (parameters and variables are discussed later in this chapter). The expression this.address.city is also valid if the candidate class contains a persistent field address whose type is a persistent class with a persistent field city. In Java, a NullPointerException is thrown on any attempt to access a field or a method using a null reference. JDOQL behavior is different. An attempt to access a field or invoke a method using a null reference results in a false evaluation for the containing expression but no exception is not thrown. Notice that the candidate object can still be included in the result collection if the failed expression is only a subexpression of the whole filter expression (for example, when the || operator is used).

### Operators

#### Comparison operators

Comparison operators (==, !=, <, >, <=, >=) generate boolean expressions, which are very useful in queries. Comparison operators act in JDOQL as in Java, with a few exceptions:

- Equality operators (==, !=) compare the identity of instances of user persistent classes (as in Java), but use the equals(...) method for system types (String, Date, ArrayList, ...).
- String instances can be compared using all six comparison operators.

Date instances can be compared using all six comparison operators.

Numeric wrapper types (Byte, Short, Character, Integer, Long, Float, Double), as well as BigInteger and BigDecimal can participate in comparisons as if they were primitive numeric types (byte, short, char, int, long, float, double).

**Logical Operators**

Logical operators (&&, ||, !) generate complex boolean expressions out of simpler boolean expressions. These operators function in JDOQL exactly as they do in Java.

**Arithmetic Operators**

In JDOQL, arithmetic operators (+, -, *, /) can be applied to numeric wrapper types (Byte, Character, Short, Integer, Long, Float, Double, and also BigInteger and BigDecimal). The concatenation operator '+' can only be applied to two strings, but not to a string and some other type as in Java. Besides that, their behavior is the same as in Java.

**Bitwise Operators**

Only the ~ operator is supported by JDOQL. Binary bitwise operators (&, |) are not supported.

## Methods

ObjectDB supports using any method that does not modify persistent objects, including instance methods and static methods. Notice, however, that the minimum requirement of JDO implementation includes support for two string methods (startsWith and endsWith), and two collection methods (contains and isEmpty, where isEmpty() also returns true when invoked on a null value). Therefore, using other methods in queries might be less portable. One of the useful string methods that is not supported by JDO 1.0 but supported in ObjectDB (and will be supported by JDO 2.0) is the match(...) method. It provides string comparison using regular expressions (a possible replacement to the like operator of SQL). To use methods of user defined classes in queries, the user code must be available. In embedded mode, user classes are always available. In client-server mode, on the other hand, the classes are usually available on the client side, but not on the server side. Therefore, to enable user defined methods in query execution on the server side, classes have to be added to the classpath of the server as well.

## Casting

Casting is also supported by JDOQL. If a ClassCastException occurs during evaluation the expression is evaluated as false but an exception is not thrown (similarly to handling NullPointerException by JDOQL, as explained above).

**Gathering Elements**

Usually, a query filter contains a combination of elements. For example:

```
Query query = pm.newQuery(Person.class);
query.setFilter("!this.children.isEmpty() && " +
  "this.age - ((Person)this.children.get(0)).age < 25");
Collection result = (Collection)query.execute();
```

This query retrieves all the parents who are older than their older child by not more than 25 years (assuming the child at position 0 is the older child). Support of the get(...) method is an extension of ObjectDB, but all the other elements are standard JDOQL elements. A JDO portable version of this query is shown in section 7.4.

## 7.3.  Query Parameters

Using parameters instead of literals produces more generic queries, which can be executed with different argument values. The role of parameters in JDOQL, however, is for much more than just making more generic queries. The only way to include objects with no literal representation in queries (as Date instances, and instances of user defined classes) is to use parameters.

**Primitive Type Parameters**

The first query in this chapter (in Section 7.1) includes an int literal:

```
Query query = pm.newQuery(Person.class, "this.age >= 18");
Collection result = (Collection)query.execute();
```

The literal, 18, can be replaced by a parameter, age, whose type is int:

```
Query query = pm.newQuery(Person.class, "this.age >= age");
query.declareParameters("int age");
Collection result1 = (Collection)query.execute(new Integer(18));
Collection result2 = (Collection)query.execute(new Integer(21));
```

Parameters are declared by the declareParameters (...) method, with a syntax that is similar to the syntax of parameter declarations in Java. The name of a parameter has to be a valid identifier in Java. Every parameter that is used in the filter has to be declared; otherwise, the compilation of the query fails. Notice that, in the above query, the this. prefix in this.age is required to distinguish the field from the parameter. This code demonstrates the execution of a query with two different arguments. An attempt to use the no-arg execute() method on a query with parameters throws a JDOUserException, because when a query is executed an argument has to be provided for each of its declared parameters. Wrapper values (Boolean, Byte, Short, Character, Integer, Long, Float, Double) are used instead of primitive values because execute(...) only accepts Object arguments.

### Reference Type Parameters

The following query retrieves all the children who were born in the third millennium:

```
Calendar calendar = Calendar.getInstance();
calendar.clear();
calendar.set(2000, 0, 1);
Date date = calendar.getTime();

Query query = pm.newQuery(Person.class, "this.birthDate >= date");
query.declareParameters("java.util.Date date");
Collection result = (Collection)query.execute(date);
```

The first four lines of code prepare a Date instance representing the first day of the year 2000. The query is declared with a parameter date of type Date, and an argument for this parameter is provided when the query is executed. Class Person must have a persistent field with the name birthDate and type Date. Otherwise, the query compilation fails. As explained in section 7.2, the comparison of Date instances is supported by JDOQL. A full class name, java.util.Date, has to be specified in the parameter declaration, unless an import statement is used, as explained in section 7.5.

Date parameters are required to include Date values in queries. String values, on the other hand, can be represented in queries by literals. Double quote characters are specified in Java strings using the \" sequence:

```
Query query = pm.newQuery(Person.class, "this.firstName == \"Steve\"");
Collection result = (Collection)query.execute();
```

Alternatively, the literal can be replaced by a parameter:

```
Query query = pm.newQuery(Person.class, "this.firstName == firstName");
query.declareParameters("String firstName");
Collection result = (Collection)query.execute("Steve");
```

The parameter type can be specified here as String rather than as java.lang.String because classes of java.lang are automatically imported. More details can be found in section 7.5.

Instances of user defined classes can also be used as parameters:

```
Query query = pm.newQuery(Person.class, "this != p1 && this != p2");
query.declareParameters("Person p1, Person p2");
Collection result = (Collection)query.execute(person1, person2);
```

With the above query, all the Person instances are retrieved, except the two Person instances that are specified as arguments. To execute the query, two Person arguments for the two declared parameters must be provided. Also, person1 and person2 must be instances of class Person (or its subclasses), otherwise the query execution throws a JDOUserException.

**Array and Maps of Parameters**

Queries with up to three parameters can be executed using the following Query interface methods:

```
void execute()
void execute(Object p1)
void execute(Object p1, Object p2)
void execute(Object p1, Object p2, Object p3)
```

For example, all the people in a range of ages can be retrieved by:

```
Query query = pm.newQuery(Person.class);
query.declareParameters("int age1, int age2");
query.setFilter("this.age >= age1 && this.age <= age2");
Collection result =
  (Collection)query.execute(new Integer(20), new Integer(60));
```

The same query can be executed using other Query interface methods:

```
void executeWithArray(Object[] parameters)
void executeWithMap(Object[] parameters)
```

Arguments can be passed in an array:

```
Query query = pm.newQuery(Person.class);
query.declareParameters("int age1, int age2");
query.setFilter("this.age >= age1 && this.age <= age2");
Integer[] args = new Integer[] { new Integer(20), new Integer(60) };
Collection result = (Collection)query.executeWithArray(args);
```

Arguments can also be passed in a map (by names rather than by order):

```
Query query = pm.newQuery(Person.class);
query.declareParameters("int age1, int age2");
query.setFilter("this.age >= age1 && this.age <= age2");
Map args = new HashMap();
args.put("age1", new Integer(20));
args.put("age2", new Integer(60));
Collection result = (Collection)query.executeWithMap(args);
```

Arrays and maps of parameters are useful mainly in executing queries with more than three parameters.

## 7.4.  Query Variables

There are two types of variables in JDOQL:

- Bound variables, whose main role is in querying collection fields
- Unbound variables, which serve as a replacement for the JOIN operation of SQL

Support of bound variables is mandatory, but support of unbound variables is optional in JDO. ObjectDB supports both types of variables.

### The contains(...) Method

Bound variables in JDOQL go side by side with the contains(...) method, which is one of the few methods that every JDO implementation should support (see <u>section 7.2</u>). The following query, which does not use variables but does use contains(...), retrieves all the people who live in cities from a specified collection myCities:

```
Query query = pm.newQuery(
  Person.class, "cities.contains(this.address.city)");
query.declareParameters("Collection cities");
Collection result = (Collection)query.execute(myCities);
```

The contains(...) method is used mainly in queries with bound variables, but as shown above, it is also useful in queries without variables. The contains(...) method works in JDOQL as in Java, except that JDOQL returns false when contains(...) is called on a null reference, while Java throws a NullPointerException.

### Bound Variables

A bound variable is an identifier that appears in the query filter as an argument of contains(...), and all its other appearances in the filter (usually at least one additional appearance) are in subexpressions that are ANDed with that contains(...) clause. In ObjectDB the order of the ANDed expressions is not important, but according to JDO, the contains(...) clause must come first (i.e. on the left side), before any other appearance of the variable in the query filter. The following query retrieves people with at least one child living in London:

```
Query query = pm.newQuery(Person.class);
query.declareVariables("Person child");
query.setFilter(
  "this.children.contains(child) && child.address.city == \"London\"");
Collection result = (Collection)query.execute();
```

A variable child whose type is Person is declared by the declareVariables(...) method. Variables (as well as parameters) can be declared before or after setting the query filter, but they must be declared before the query is compiled and executed, otherwise the query compilation fails because of unknown identifiers.

When an AND expression that includes a bound variable is evaluated, all the possible values for that variable are considered. The expression is evaluated as true if there is at least one variable value for which the expression value is true. In the query above, when a candidate Person is evaluated as this, only child values contained in its children collection are considered, because of the contains(...) clause. If there is such a child for whom the

expression on the right side of the && operator is true, the entire AND expression is evaluated to true, and the candidate object is added to the result collection.

### Negative Contains(...)

The next query retrieves all the parents who are older than their older child by not more than 25 years (a similar query, but not JDO portable, is shown in <u>section 7.2</u>):

```
Query query = pm.newQuery(Person.class);
query.declareVariables("Person child");
query.setFilter(
  "this.children.contains(child) && this.age - child.age <= 25"
Collection result = (Collection)query.execute();
```

A single child satisfying the "this.age - child.age <= 25" expression is sufficient to include this in the result collection. On the other hand, to retrieve only parents with **all** children satisfying some condition, a trick has to be used. The following query uses De Morgan's rules to retrieve all the people who are older than every one of their children by not more than 25 years (including people without any children):

```
Query query = pm.newQuery(Person.class);
query.declareVariables("Person child");
query.setFilter(
  "!(this.children.contains(child) && this.age - child.age > 25)"
Collection result = (Collection)query.execute();
```

The expression in parenthesis refers to the people who are older than at least one of their children by more than 25 year. But because of the ! operator, eventually, the result collection contains all the people but them, i.e. all the people who are older than every one of their children by not more than 25 years.

### Nested Bound Variables

In some cases, more than one contains(...) clause is needed. The following query retrieves all the people that have at least one child with a cat:

```
Query query = pm.newQuery(Person.class);
query.declareVariables("Person child; Pet pet;");
query.setFilter(this.children.contains(child) && " +
                          "child.pets.contains(pet) && pet.isCat()");
Collection result = (Collection)query.execute();
```

When more than one variable is declared, a semicolon is used as a separator. A semicolon at the end of the string is optional. Multiple variables are required above because two collection fields are involved, children and pets. Only Person instances, with a combination of child and pet satisfying the AND expression, are included in the result collection. Because child is

dependent on this and pet is dependent on child, the three subexpressions must be ordered as shown above, in order to make the query JDO portable.

### Unbound Variables

Unbound variables are variables that are not constrained by a contains(...). The following query retrieves all the people except the oldest and the youngest:

```
Query query = pm.newQuery(Person.class);
query.declareVariables("Person p1; Person p2");
query.setFilter("this.age > p1.age && this.age < p2.age");
Collection result = (Collection)query.execute();
```

The type of every unbound variable is expected to be a persistent class with Extent support (not necessarily the candidate class as in this example). The variable can have as values any objects in the Extent. The result collection contains all the instances of the candidate collection (excluding non instances of the candidate class), for which there is at least one combination of variables that makes the query filter evaluate to true.

Unbound variables are supported by ObjectDB, but considered optional by JDO. Queries with unbound variables are similar to JOIN queries in SQL because every combination of variables has to be checked for every candidate object. Just like JOIN queries in SQL, queries with unbound variables may become very slow, so caution is needed when using them.

## 7.5.  Import Declarations

Names of classes are used in many components of JDOQL, including the declaration of parameters and variables, when casting and when accessing static methods and fields (supported by ObjectDB as an extension to JDO). As in Java, full class names that include a package name can always be used. The package name can be omitted only for classes in the java.lang package (e.g. String instead of java.lang.String), for classes that are in the same package as the candidate class, and when a proper import declaration is used.

The following query, previously discussed in section 7.3, serves as a good example:

```
Query query = pm.newQuery(Person.class, "this.birthDate >= date");
query.declareParameters("java.util.Date date");
Collection result = (Collection)query.execute(date);
```

The full name of class Date is specified because that class is in neither package java.lang nor the package of the candidate class. Using a short class name (without the package name) causes a query compilation error unless a proper declareImports(...) declaration exists, as shown next:

```
Query query = pm.newQuery(Person.class, "this.birthDate >= date");
query.declareImports("import java.util.Date");
```

```
query.declareParameters("Date date");
Collection result = (Collection)query.execute(date);
```

Multiple import statements can also be declared:

```
query.declareImports(
   "import java.util.*; import directory.pc.Category;");
```

The argument of declareImports(...) is expected to use the Java syntax for import statements. Multiple import statements are separated by semicolons. A semicolon at the end of the string is optional.

Of all the JDOQL components, import declarations are the most rarely used because most of the classes in queries belong to java.lang or to the package of the candidate class and because classes of other packages can be specified by their full name without using an import declaration.

## 7.6.  Ordering the Results

A result collection can be ordered by a query. For example:

```
Query query = pm.newQuery(Person.class, "this.age >= 18");
query.setOrdering("this.age ascending");
Collection result = (Collection)query.execute();
```

As a result of calling setOrdering(...), an ordered collection is returned by the query. Iterating over the result collection using Iterator returns people in order from younger to older. As usual, the this. prefix can be omitted. The ascending keyword specifies ordering from low to high. To order the results in reverse order, the descending keyword can be used:

```
Query query = pm.newQuery(Person.class, "this.age >= 18");
query.setOrdering("this.age descending");
Collection result = (Collection)query.execute();
```

Iteration over the new result collection returns people in order from older to younger. Note that when setOrdering(...) is not used, ObjectDB orders result collections by object IDs from lower to higher, which means that objects will be returned in the order in which they were added to the database (but this behavior is not specified by JDO).

The syntax of ordering expressions is similar to the syntax of query filters, except that variables and parameters cannot be used, and the type of each ordering expression is some comparable type rather than boolean. Valid types include numeric primitive types (byte, short, char, int, long, float, double), numeric wrapper types (Byte, Short, Character, Integer, Long, Float, Double), String, Date, BigInteger and BigDecimal.

Multiple ordering expressions can also be specified:

```
Query query = pm.newQuery(Person.class, "this.age >= 18");
query.setOrdering("age descending, children.size() ascending");
Collection result = (Collection)query.execute();
```

The primary ordering is age and the secondary ordering is children.size() (which is supported by ObjectDB as an extension to JDO). Results are ordered by the primary ordering expression. The secondary ordering expression is only needed when two or more result instances share the same primary ordering evaluation.

# Chapter 8.  ObjectDB Server

An ObjectDB server can manage one or more databases. Databases that are managed by a server can be accessed by multiple processes simultaneously. In addition, the server supports accessing these databases from remote machines by TCP/IP. More details about client server mode vs. embedded database mode are discussed in Section 1.2. Because the JDO API and the format of the database file are the same in both modes, switching between these two modes can be done very easily by simply changing the connection URL string.

This chapter contains the following sections:

## 8.1.  Running an ObjectDB Server

The ObjectDB server is a pure Java application, packaged in the **odbse.jar** file. The jar file contains the entire ObjectDB implementation including support for embedded mode. In order to use client server mode, you have to install the **odbse.jar** file on **both** the client and the server machines. This section explains how to run the server. More details on server configuration and operation are provided later in this chapter.

**The Server as a Java program**

You can run the server as a Java program from the command line, as so:

```
> java -cp <OBJECTDB_HOME>/lib/odbse.jar com.objectdb.Server
```

Or on Windows:

```
> java -cp <OBJECTDB_HOME>\lib\odbse.jar com.objectdb.Server
```

where <OBJECTDB_HOME> represents the installation path of ObjectDB.

If you add **odbse.jar** to the classpath, you can run the server simply by:

```
> java com.objectdb.Server
```

Running the ObjectDB server edition without command line arguments displays a usage message, as shown below:

```
Usage: java com.objectdb.Server [options] start | stop | restart
options include:
  -conf <path>  :  specify a configuration file explicitly
```

```
  -port <port>  :  override configuration TCP port to listen to
```

To start the server, use the **start** command:

```
> java com.objectdb.Server start
```

Running the ObjectDB server requires the specification of an ObjectDB server configuration file. ObjectDB is shipped with a default server configuration file named **default.xml**. It is recommended that you use this file as a template for constructing your own specific configuration file. You can use the –conf command line parameter to specify the location of your server configuration file, as shown below:

You can also specify an explicit path to the configuration file:

```
> java com.objectdb.Server -conf config.xml start
```

If you name your configuration file **server.xml** you do not need to use the –conf command line parameter since ObjectDB will automatically look for a **server/server.xml** file if the –conf parameter is not used. Also, if you do not use the –conf parameter and do not create a **server.xml** file the ObjectDB server will use the **default.xml** file as a last resort. Please note, if you edit the **default.xml** file it is likely that your modifications will be overwritten when you extract a newer version of the ObjectDB server.

The TPC port on which the server is listening for new connections is specified in the configuration file (as explained in [section 8.2](#)), but you can override this setting using the –port command line option to specify another port, as shown below:

```
> java com.objectdb.Server -port 8888 start
```

You can also use standard JVM arguments. For instance, you can increase the maximum allowed memory and instruct Java to run in server mode in order to enlarge the cache and improve server performance:

```
> java -server -Xmx512m com.objectdb.Server start
```

To stop the server you can use the **stop** command:

```
> java com.objectdb.Server stop
```

To stop the server and then immediately start it again, you can use the **restart** command:

```
> java com.objectdb.Server restart
```

While a server is running in the foreground the command window may be blocked. Therefore, you may need a new command window for the **stop** and **restart** commands. The –conf and –port commands can also be used with the **stop** and **restart** commands. However, in this

context, you must use the same port number that was specified during server startup in order for the stop/restart command to have any effect.

### Running the Server on Unix

On Unix you can also use a shell script to run and manage the server. A sample script, **server.sh**, is located in the **bin/sh** subdirectory. Before using that script you have to edit the path to the **odbse.jar** file, and to the JVM. It is recommended that you do not edit this file directly. Rather, you should copy it to the **bin** subdirectory and then edit the new copy. In this way, the edited script will not be deleted when extracting the files of a newer ObjectDB version to the same installation directory.

Consult your operating system documentation about running the server in the background (for instance, using the & character at the end of the command line), or starting the server automatically at boot time and stopping it at shutdown time, or on restarting the server periodically (for instance, using crontab).

### Running the Server on Windows

On Windows you can also run the server using the **server.exe** application, located in the **bin** directory. For this to work, the original structure of ObjectDB directory must be preserved because **server.exe** tries to locate and load the **odbse.jar** and **jdo.jar** files. By default, **server.exe** starts the server using the following command:

```
> java -server -Xms32m -Xmx512m com.objectdb.Server start
```

When running **server.exe**, you can specify arguments for the JVM as well as for the server (excluding the stop and the restart server commands). For example:

```
> server.exe -client -Xmx256m -port 6666
```

The specified arguments override conflicting defaults. Therefore, the above run uses the following command:

```
> java -client -Xms32m -Xmx256m com.objectdb.Server start -port 6666
```

The **server.exe** application is represented, when running, by an icon in the Windows Tray. Right click the icon and use the context popup menu to manage the server (stop, restart and start), and to exit the application.

## 8.2.  Single User Server Configuration

As noted before, to run the ObjectDB server a configuration file is required. If a path to the configuration file is not explicitly specified when starting the server, default paths are checked. A valid configuration file is expected to be located on one of these paths. This

section discusses a simple configuration file with a single registered user. Notice that a single user configuration may also be used by multi user applications (such as web applications) where the same username and password can be shared by all the database connections.

The following is a simple configuration file with one registered user:

```
<objectdb.com>

  <server port="6136" reload="30">
    <data path="$objectdb/server/data" />
    <log path="$objectdb/server/log" />
    <connections max="10" user-max="10" timeout="300" />
  </server>

  <users>
    <user name="admin" password="admin" address="127.0.0.1">
      <dir path="/">
        <permissions access="+" modify="+" create="+" delete="+" />
      </dir>
    </user>
  </users>

</objectdb.com>
```

The server configuration file is in XML format. It always includes an <objectdb.com> root element with two sub elements, <server> and <users>. The <server> sub element specifies general server settings, and the <users> sub element lists all the registered users.

### The <server> element

```
  <server port="6136" reload="30">
    <data path="$objectdb/server/data" />
    <log path="$objectdb/server/log" />
    <connections max="10" user-max="10" timeout="300" />
  </server>
```

All attributes and sub elements shown in the above <server> element are required.

The port attribute specifies the TPC port on which the server will be listening for new connections. Usually it should be set to 6136, which is the default port of ObjectDB. If a port other than the default is specified, it also has to be specified by clients in the url connection string (as explained in section 5.1) when connecting to the database.

The reload attribute specifies how often (in seconds) the server checks the configuration file for changes. For instance, a value of 30 (as specified above) indicates a check every 30 seconds. If a change is found, the new configuration is loaded automatically without having to restart the server. To turn off auto reloading specify a value of "0" for the reload attribute. In this case the server has to be restarted manually to apply configuration changes.

**The &lt;data&gt; element**

The &lt;data&gt; element specifies the data directory containing all the database files that the server is allowed to access and manage. Please note: appropriate file system permissions have to be set on that directory to enable operations by the server process. The location of the data directory is specified in the path attribute using an absolute path. The $objectdb prefix can be used to represent ObjectDB installation directory, as demonstrated in the configuration shown above. The data directory of the ObjectDB server is very similar to the document root directory of a web server. Every file in the data directory or in its subdirectories can be accessed by the server but any file outside this directory cannot be accessed. When connecting to the server, the path specified in the url connection is resolved relative to the data directory. For instance, a url connection string "objectdb://localhost/my/db.odb" refers to a database file **db.odb** in a subdirectory **my** of the data directory.

**The &lt;log&gt; element**

The &lt;log&gt; element specifies a directory to which the server writes its log files. Appropriate file system permissions have to be set on that directory to enable operations by the server process. The location of the log directory is specified in the path attribute using an absolute path. The $objectdb prefix can be used to represent the ObjectDB installation directory, as demonstrated in the configuration shown above.

**The &lt;connections&gt; element**

The &lt;connections&gt; element specifies restrictions on connections to the server. The max attribute specifies the maximum simultaneous connections allowed by the server. The user-max attribute specifies the maximum simultaneous connections allowed for each user. Because only one registered user is defined in the configuration above, there is no point to specifying different values for the max and user-max attributes (if different values are specified the lowest value is applied). A request for a connection that exceeds the maximum is blocked until some open connection is closed or until a timeout specified on the client side is exceeded. The role of the timeout attribute in the server configuration file is different. It specifies the maximum allowed connection inactivity time (in seconds). If a connection is inactive for more than the specified time the server may close it.

**The &lt;users&gt; element**

```
<users>
  <user name="admin" password="admin" address="127.0.0.1">
    <dir path="/">
      <permissions access="+" modify="+" create="+" delete="+" />
    </dir>
  </user>
</users>
```

The <users> element lists all the users who are allowed to connect to the server.

### The <user> element

Every user is represented by a single <user> element. The required name and password attributes specify a username and a password to be used when connecting to the server (see chapter 5). The address attribute is optional. If specified, the user is restricted to connect to the server only from the specified IP address. For instance, the "127.0.0.1" value above (which always represents the local machine) enables connecting to the database server only from the machine on which the server process is running. Multiple IP addresses can also be specified in a comma separated list and using a hyphen (-) to indicate a range. For example, a value "192.18.0.0-192.18.194.255,127.0.0.1" allows connections from any IP address in the range of 192.18.0.0 to 192.18.194.255, as well as from 127.0.0.1.

### The <dir> element

Every <user> element should have one or more <dir> sub elements, indicating which paths under the server data directory (as set by the data element) the user is allowed to access. The required path attribute specifies a path to a directory relative to the data directory root. A permission to access a directory always includes a permission to access the whole tree of subdirectories under that directory. Therefore, a path "/", as specified in the configuration above, indicates that the user can access the entire data directory. By using the <dir> element in a multi user configuration, every user can be assigned a private subdirectory containing his or her private database files (as demonstrated in section 8.3).

### The <permissions> element

The <dir> element itself grants the user permission to view the directory content (using the Explorer). Additional permissions are granted using the <permissions> sub element and its four attributes. The access permission enables opening database files and the modify permission enables modifying their content. In order to create new subdirectories and new database files, a create permission is required. A delete permission is required for deletion of subdirectories and database files. Permissions are granted by "+" values in the proper attributes. Notice that a permission to do an operation in a directory always includes the permission to do the same operation in subdirectories of that directory.

## 8.3.  Multi User Server Configuration

Similar to a web server, which can serve either a single website or multiple websites of one or more users, an ObjectDB server can manage a single database file or multiple database files of any number of users. As long as only one user is involved, a simple configuration as demonstrated in the previous section is sufficient, even for more than just one database file.

This section discusses a more complicated case in which multiple users use a shared server to manage different database files. Providing database server support on a shared server could be useful in a shared web hosting environment and in educational settings (such as universities and colleges).

The following configuration file declares three users:

```
<objectdb.com>

  <server port="6136" update-rate="20">
    <data path="$objectdb/server/data" />
    <log path="$objectdb/server/log" />
    <sessions max="50" user-max="5" timeout="60" />
  </server>

  <users>
    <user name="admin" password="admin" address="192.18.97.71">
      <dir path="/">
        <permissions access="+" modify="+" create="+" delete="+" />
      </dir>
    </user>
    <user name="user1" password="user1">
      <dir path="/home/user1">
        <permissions access="+" modify="+" create="+" delete="+" />
        <quota subdirs="0" files="3" diskspace="500k" />
      </dir>
    </user>
    <user name="user2" password="user2">
      <dir path="/home/user2">
        <permissions access="+" modify="+" create="+" delete="+" />
        <quota subdirs="20" files="100" diskspace="20m" />
      </dir>
    </user>
  </users>

</objectdb.com>
```

The administrator user, admin, is authorized to manage database files and subdirectories anywhere in the data directory. For the sake of security, the administrator is only allowed to connect to the server from a specified IP address. The two other users, user1 and user2, can connect to the server from any IP address, but each one is only authorized to access a specific directory, which is dedicated for his or her use.

**The <quota> element**

A <dir> element can have an optional <quota> sub element, specifying restrictions on the directory content. All three attributes are required. The subdirs attribute specifies how many subdirectories are allowed under that directory (nested subdirectories are also allowed). The files attribute specifies how many database files the directory may contain. Finally, the diskspace attribute specifies maximum disk space for all the files in that directory. A suffix m

indicates megabytes and a suffix k indicates kilobytes. Only one <quota> element is permitted per directory. Therefore, if a directory is represented by multiple <dir> elements (in different <user> elements), no more than one of them can have a <quota> sub element.

## User Inheritance

User inheritance may simplify management of many similar user accounts:

```
<users>
  <user name="admin" password="admin">
    <dir path="/">
      <permissions access="+" modify="+" create="+" delete="+" />
    </dir>
  </user>
  <user name="$standard" address="12.*.*.*">
    <dir path="/home/$user/">
      <permissions access="+" modify="+" create="+" delete="+" />
      <quota subdirs="20" files="20" diskspace="5m" />
    </dir>
    <dir path="/examples">
      <permissions access="+" />
    </dir>
  </user>
  <user name="user1" password="user1" super="$standard" />
  <user name="user2" password="user2" super="$standard" />
</users>
```

A <user> element whose user name starts with a $ character, such as $standard in the configuration above, is abstract and does not represent a real user. Connections to the database cannot be obtained using an abstract user, but an abstract <user> element can serve as a base for other ordinary <user> elements, as demonstrated by user1 and user2 above. Only the name of an abstract <user> element can be specified in the optional super attribute, and that abstract user must be declared in the configuration file before its derived users. One abstract user can inherit from another abstract user. When using the super attribute, all the attributes and the sub elements of the specified abstract user are inherited except the name attribute. The $user variable (in the path attribute of $standard) is evaluated, when inherited, in every user to their name. Therefore, user1 gets access to directory /home/user1 and user2 to directory /home/user2.

Inherited attributes can be overridden, as demonstrated by user3:

```
<user name="user3" password="user3" super="$standard" address="">
  <dir path="/home/$user/">
    <permissions delete="-" />
    <quota subdirs="20" databases="20" diskspace="20m" />
  </dir>
</user>
```

Three attributes are changed. The empty address attribute eliminates the inherited address restriction. The minus value in the delete attribute removes an inherited permission, and a new value in the diskspace attribute increases the disk quota for the user.

## 8.4.  Using Secure Socket Layer (SSL)

ObjectDB supports encryption of client server communications using the Secure Sockets Layer (SSL) protocol. Encryption is especially important when the communication between the client and the server is over an unsecured network, such as the Internet. There are several implementations of SSL available for Java. ObjectDB uses the JSSE implementation, which is integrated in J2SDK as a standard package since version 1.4. For more information on JSSE, see:

http://java.sun.com/j2se/1.4.2/docs/guide/security/jsse/JSSERefGuide.html

### Keystore and Trustore

To use SSL you have to generate at least two files:

- A **Keystore** file that functions as a unique signature of your server. This file contains general details (such as a company name), an RSA private key and its corresponding public key (the SSL protocol is based on the RSA algorithm).
- A **Trustore** file that functions as a certificate that enables the client to validate the server signature. This file is generated from the Keystore file, by omitting the private key (it still contains the general information and the public key).

You can generate these files using the **keytool** utility (located in the J2SDK **bin** directory), as demonstrated in:

http://java.sun.com/j2se/1.4.2/docs/guide/security/jsse/JSSERefGuide.html#CreateKeystore

Using these Keystore and Trustore files, a client can verify during SSL handshaking that it is connected to the real server, and not to another server in the way that is pretending to be the real server (what is known as "a man in the middle attack"). The server, on the other hand, might be less selective and allow connections from any machine, as long as a valid username and password are provided. If an authentication of the client machine by the server is also required, a Keystore file (which might be different from the server Keystore) has to be installed on the client machine, and its corresponding Trustore file has to be installed on the server machine.

### Setting the Server to use SSL

To setup an ObjectDB server for using SSL, you have to put an <ssl> element with a <keystore> sub element inside the <server> element. The keystore file and its password are

specified in the two required attributes of the <keystore> element, where $objectdb represents the ObjectDB installation directory, as shown below:

```
<server port="6136">
    :
  <ssl>
    <keystore path="$objectdb/server/keystore" password="ksp-pwd" />
  </ssl>
    :
</server>
```

When client authentication is also required, the <ssl> element should also contain a <truststore> sub element that serves as a certification of the client signature:

```
<server port="6136">
    :
  <ssl>
    <keystore path="$objectdb/server/keystore" password="ksp-pwd" />
    <truststore path="$objectdb/server/truststore" password="ts-pwd" />
  </ssl>
    :
</server>
```

## Setting the Client to use SSL

To access an ObjectDB server that is set to use SSL, a Truststore file for authentication of the server has to be provided. If the PersistenceManagerFactory is initialized using a property file, two lines should be added to specify the path of the Truststore file and its password:

```
com.objectdb.ssl.truststore.path=$objectdb/lib/truststore
com.objectdb.ssl.truststore.password=ts-pwd
```

Alternatively, the PersistenceManagerFactory's properties can be set at runtime:

```
Properties properties = new Properties();
    :
    :
properties.setProperty(
   "com.objectdb.ssl.truststore.path", "$objectdb/lib/truststore");
properties.setProperty(
   "com.objectdb.ssl.truststore.password", "ts-pwd");

PersistenceManagerFactory pmf =
   JDOHelper.getPersistenceManagerFactory(properties);
```

If client authentication is also required, the client's Keystore can also be specified in a property file:

```
com.objectdb.ssl.keystore.path=$objectdb/lib/keystore
com.objectdb.ssl.keystore.password=ks-pwd
```

or at runtime:

```
Properties properties = new Properties();
  :
  :
properties.setProperty(
  "com.objectdb.ssl.keystore.path", "$objectdb/lib/keystore");
properties.setProperty(
  "com.objectdb.ssl.keystore.password", "ks-pwd");
```

Finally, to access the server using SSL, the connection URL should start with objectdbs:// instead of with objectdb://. Otherwise, the Keystore and the Truststore properties are ignored and an attempt to establish an ordinary non secured connection is performed. Of course, this attempt is expected to fail if the server is set to use SSL.

# Chapter 9.  ObjectDB Explorer

ObjectDB Explorer is a visual tool for managing ObjectDB databases. You can use the Explorer to browse databases, execute JDOQL queries, create new databases and edit the content of existing databases.
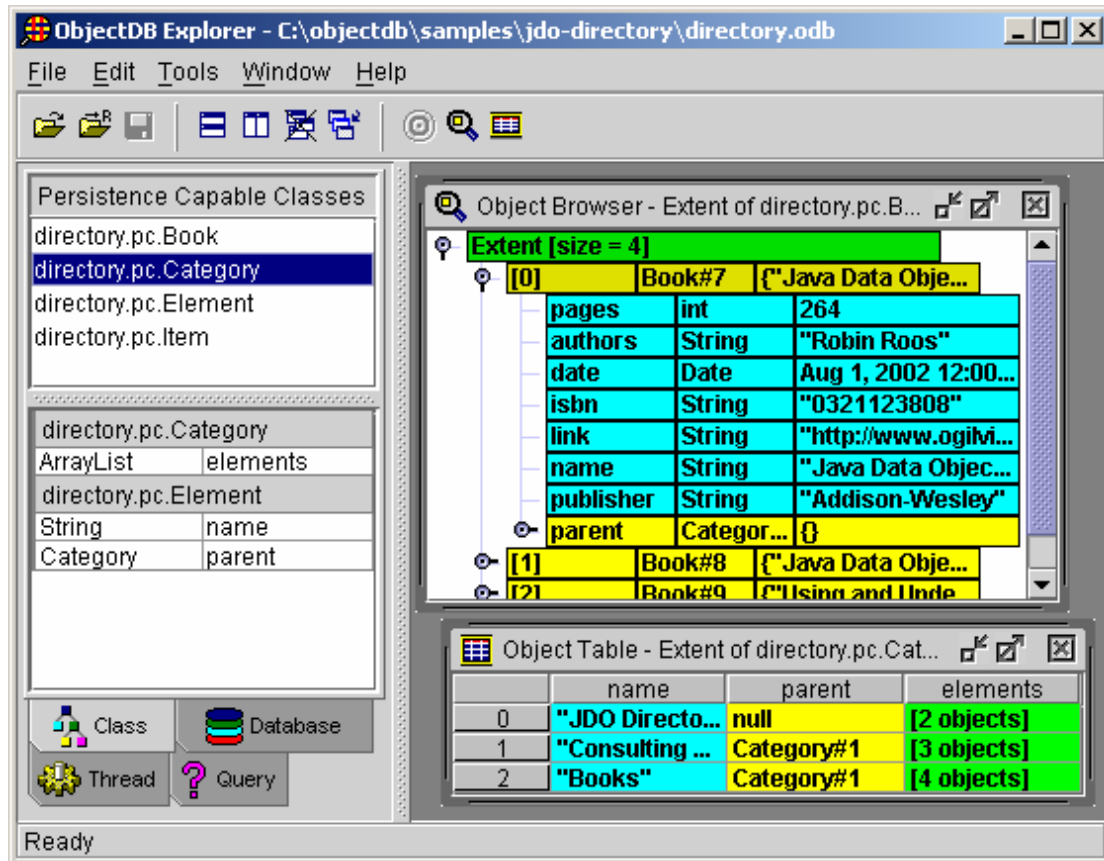


This chapter contains the following sections:

## 9.1.  Running the Explorer

The Explorer is a pure Java GUI application.  It is shipped as an executable jar that will run on any system with a Java 1.3 JRE or later installed. A Windows EXE (explorer.exe) and a Unix shell script (explorer.sh) for starting the Explorer on those respective operating systems are also provided.

**Running the Executable JAR directly**

The Explorer is packaged in the main ObjectDB development jar file (odbfe.jar / odbee.jar / odbse.jar), located in ObjectDB's lib directory. Because this file is an executable jar, on most systems you can double click on it to start the Explorer.

You can also start the Explorer from the command line. For instance, if the installation path of the ObjectDB Free Edition on Windows is **c:\objectdb**, you can start the Explorer using the following command:

```
> java -jar c:\objectdb\lib\odbfe.jar
```

If a database path is specified, the database is opened automatically when the Explorer starts:

```
> java -jar c:\objectdb\lib\odbfe.jar my.odb
```

Some JVM arguments can be useful. For instance, by default Java does not use more than 64MB of memory. If you need to run the Explorer with increased available memory, you can use the **-Xmx** JVM argument:

```
> java -Xmx256m -jar c:\objectdb\lib\odbfe.jar
```

Similar commands (but with different paths) can run the Explorer on other operating systems.

**Starting the Explorer on Windows**

You can start the Explorer on Windows using the explorer.exe application, located in the bin directory. Notice that the original structure of the ObjectDB installation directory must be preserved. Otherwise, the required executable jar would not be found by explorer.exe. By default, running explorer.exe is equivalent to the following command:

```
> java -Xms16m -Xmx512m -jar odbfe.jar
```

In addition, any JVM argument can be specified as an argument to explorer.exe during startup. For instance:

```
> explorer.exe -Xmx256m my.odb
```

Explicitly specified arguments override the default. Therefore, the above command is equivalent to:

```
> java -Xms16m -Xmx256m -jar odbfe.jar my.odb
```

**Starting the Explorer on Unix**

You can start the Explorer on Unix using a shell script. A sample script, explorer.sh, is located in the bin/sh directory. Before using explorer.sh you have to edit the paths that it contains. The following procedure is recommended, copy the file to the **bin** subdirectory, edit

the copy and use it to start Explorer. In this way, the edited script will not be deleted when extracting files of a newer ObjectDB version to the same installation directory.

## 9.2.  Browsing the Database

The ability to explore database objects visually, navigate among them and execute queries is very valuable during development and debugging. Therefore, these are the most commonly used features of the Explorer.

### Opening a Local Database

To open a local database file use the "File | Open Local" menu command (or the equivalent toolbar button) and in the Open dialog box select the desired local database file and click the "Open" button. Recently used local database files can also be opened using the "File | Recent Local Files" menu command. By default, when the Explorer starts its opens the last used local database file automatically. You can change this behavior in "Tools | Options | General".

### Opening a Remote Database

To open a connection to a remote database using client-server mode (supported only by ObjectDB Server Edition) use the "File | Open Remote" menu command (or the equivalent toolbar button). In the Open dialog you have to fill in the host, port, username and password of the remote connection. You also have to specify a path of the database file on the remote server, possibly using the "Browse" button that opens the "Remote File Selection" dialog box.

### Closing the Database

Use the "File | Close" menu command to close a local database file or a connection to a remote database file.

### The Tabbed Windows

Four tabbed windows are displayed on the left side of the Explorer desktop when a database is open. The "Class" window shows all the persistent classes in the database. You can select a class from the list of classes at the top of this window and see all its persistent fields at the bottom. The "Database" window is also split into two sub windows. The top sub window displays general information about the database and the bottom sub window displays a list of root objects that the database contains (root objects are discussed in chapter 6). The "Query" window enables a user to execute JDOQL queries in the Explorer, as discussed below. Finally, the "Thread" window, unlike the other three tabbed windows, is displayed even when no database is open in the Explorer, because it is not related to a specific database. Section 9.4 explains how to work with threads in the Explorer .

**Using Viewer Windows**

The Explorer provides two types of viewer windows for exploring the contents of database objects. The **Table** window displays a collection of objects following the approach of traditional visual database tools. Every row in the table represents a single object, every column represents a persistent field, and the content of a cell is the value of a single field in a single database object. This type of viewer is useful for viewing the data of a simple object model.

In most cases, however, the **Browser** window (which is designed to handle more complex object models) is preferred. The browser window displays objects using a tree. Every database object is represented by a tree node, and the values of its persistent fields are represents by child nodes. The main advantage of using the browser window is in navigation among objects. Because every reference between two database objects is represented by a parent-child relationship in the tree, you can navigate among objects by expanding nodes in the tree, similar to exploring objects in a visual debugger. Notice that the same database object might be accessed using different paths in the tree and may therefore be represented by more than one node. To help identify circles in the graph of objects, a special {R} sign (indicating recursive) is displayed for an object that is already shown in a higher level of the same tree path (i.e. the object is identified as a descendant of itself in the tree).

To open a new viewer window, first select a target element:

- Select a persistent class in the "Class" tabbed window to view all the instances of that class.
- Select a root object in the "Database" tabbed window to view the content of that object.
- Select an object in a currently open table or browser viewer to explore it in a new viewer window.

When the target element is selected, you can open a new viewer window using the "Window | Open Table Window" or the "Window | Open Browser Window" menu commands, or by the equivalent context menu commands (right clicking the selected target element) or the equivalent toolbar buttons. When the target element is an object in an open viewer window, the "Window | Open Focus Selection" command switches the current viewer to focus on the selected object. A double click on the target element opens the browser window by default (you can change the default to a table window in "Tools | Options | General | Display").

Additional information about working with viewer windows is provided in later sections. Section 9.3 explains how to use viewer windows to edit and modify the content of a database. Section 9.5 explains how to define views and set other options that affect viewer windows.

**Executing JDOQL queries**

The "Query" tabbed window on the left side of the Explorer desktop enables execution of JDOQL queries. The first step in defining a query is selecting a candidate class. All the instances of the selected class and its subclasses are included in the candidate collection of the query. Optional JDOQL components, such as filter, parameters, variables, ordering and imports (chapter 7 explains all these components), can also be specified in the query form. The "Reset" button clears the content of all the fields.

To execute a query click the "Execute" button. If the query is valid, the default viewer (the browser window by default) is opened with the query results and the size of the result collection and the query execution time are displayed on the query form. If the query compilation fails, an error is displayed on the query form and no viewer window is opened. You can also execute a query using the "Window | Open Table Window" or the "Window | Open Browser Window" menu commands (or using the equivalent toolbar buttons). This opens the specified viewer window type and displays the results.

On the bottom of the query window there are two assistant tab windows. The first tab window implements auto completion support for query definition. You can select elements from the auto completion list when you write the query filter and also when you fill the ordering field in the query form. The second tab window is filled at the moment of query execution with Java code that implements the query. You can use that code to export JDOQL queries that you test in the Explorer into your Java code.

**Refreshing the Cache**

When a database is open in the Explorer using client server mode, it can be accessed simultaneously by other applications. If the database is modified by another process the viewer windows in the Explorer might display content from the cache that does not reflect the real content in the database. In this case you can refresh the cache and the viewer windows using the "File | Refresh Data" menu command.

## 9.3.  Editing the Database

The Explorer is usually used as a viewer of database files, but it can also function as an editor. Therefore, you can create new database files and edit exiting database files using the Explorer.

**Creating a new Database**

The main purpose of the "Open Local Database" dialog box (displayed by "File | Open Local") is to select existing database files, but it can also serve as a basic file manager. Using the toolbar and the context menus of this dialog box (right click the directory tree and the file

list to open the context menus), you can copy, cut, paste, delete and rename directories and files. You can also use this dialog to create new directories and database files. Similarly, in the server edition you can manage a remote server file system using "File | Open Remote | Browse".

## Class Management

The "Class" tabbed window on the left side of the Explorer desktop lists all the persistent classes in the database. For a new, empty database this window is empty. A new persistent class is automatically added to the database when its first instance is stored by the application in the database. To add persistent classes to the database using the Explorer, first create and compile the classes externally in your IDE, and then use the "File | Load New Classes" menu command (or the equivalent context menu command). Notice that JDO metadata for the class (as discussed in chapter 4) must be available, otherwise the load operation will fail. A change to the class schema (for instance, a new persistent field added in the IDE) is applied to the database automatically when the application stores an object with the new schema in the database. To see the change in the Explorer earlier, reload the class using the same "File | Load New Classes" command.

You can also delete or rename a selected persistent class using the "Edit | Delete" and the "Edit | Rename" menu commands, respectively, or using the equivalent context menu commands. A deleted class is not shown in the list of classes anymore (unless it is loaded again), but if it still has instances in the database these objects are not affected. Renaming a class is useful as a complementary operation to renaming the class in the IDE, to avoid losing old persistent instances of the class. Similarly, renaming a selected persistent field in the Explorer is useful in order to avoid losing values of that field if the field has been renamed in the IDE.

## Object Management

To construct new persistent objects and store them in the database, open the "Construct New Persistent Objects" dialog box using one of the following methods:

- Use the "Edit | Construct Objects" menu command.
- Select a class in the "Class" tabbed window and then use the "Edit | Add To" menu command, or the equivalent "Construct Objects" context menu command.
- Select an Extent in a viewer window and then use the "Edit | Add To" menu command, or the equivalent "Add To" context menu command.

In the dialog box you have to specify the number of objects to construct and their type. Additional information is required for specific types. For instance, most system types require

specifying an initial value and arrays require specifying an array length. Click the "OK" button to construct the new persistent objects.

You can also use the Explorer to edit the content of database objects. To edit a persistent field you have to first select it in a viewer window (browser or table). You can start editing a value field by double clicking it, by pressing F2, or, more commonly, by simply typing the new value. Other editing commands are provided in the "Edit" menu and in the content menu. Use the "Edit Multi Line String" command to edit a multi line string. Use the "Set Reference" command to set a reference field to a new object, to an exiting object or to null. You can add objects or references to a persistent collection using the "Add To" command, and you can change the order of elements in an ordered collection using the "Move Element" command. The functionality of the "Delete" command changes according to the context. Deleting a reference field sets the value to null without deleting any referenced persistent object. On the other hand, deleting a persistent object (which is represented by a child of an Extent node in a browser window and by a row in a table window) deletes the object itself from the database. The clipboard commands, "Cut" "Copy" and "Paste", may also be used for setting fields (for instance, for copying a reference from one field to another).

### Root Management

The "Edit" menu and the context menu of the "Database" tabbed window provide commands for managing root objects. To define a new root use the "Create Root" command, and then in the "Create a New Root" dialog box set the name of the root and either construct a new object or select an existing object. To define an existing object as a root you can also right click the object in a viewer window and select the "Create Root" command, and then in the "Create a New Root" dialog box you can pick the selected object in the "Reference" category. You can also rename or delete roots (notice that deleting a root does not delete the object from the database, only its name) using commands in these menus.

### Saving Changes

The Explorer manages an active transaction for every open database file. All editing operations (including loading classes, constructing objects, editing fields, defining roots, etc.) are stored in memory, associated with the active transaction. The "File | Save" menu command (and the equivalent toolbar button) commits the transaction which applies all the changes to the database. The "File | Discard Changes" menu command rolls back the transaction discarding all the changes. After "File | Save" and "File | Discard Changes", the Explorer automatically begins a new transaction for the next editing session. Another useful command, "File | Save As", copies a local database file to a new location, applying all the changes of the current transaction only to the new database file. This command is available only when accessing a local database using embedded mode.

## 9.4.  Tools and Threads

The "Thread" tabbed window on the left side of the Explorer desktop manages special database tasks in the Explorer as background threads.

### Working with Threads

You can start a new thread using commands from the "Tools" menu or from the context menu. A running thread is represented by a line in the "Threads" list. The output of the selected thread is displayed at the bottom of the window. Using the context menu (or the "Tools" and "Edit" menus in the menu bar) you can stop a running thread, delete a thread that is not running, or copy selected lines from the thread output to the clipboard.

### Maintenance Tools

Three maintenance tools are supported by the Explorer. The "Start Garbage Collector" command starts a database garbage collector as a background thread. Objects that are not reachable (by navigation from Extent objects or from roots using persistence reference fields) are deleted from the database. The "Rebuild All Extents" command starts a thread that scans the entire database and rebuilds all the Extents. This tool is useful when changing the value of the "requires-extent" attribute (in the JDO metadata) for a persistent class that already has persistent instances in the database. You can also start this thread for a single class by right clicking the class in the "Class" tabbed window and selecting "Rebuild Class Extent". The "Rebuild All Indexes" command starts a thread that scans the entire database and rebuilds all the indexes. This tool is useful when adding a new index (in the JDO metadata) to a class that already has persistent instances in the database. You can also start this thread for a single class by right clicking the class in the "Class" tabbed window and selecting "Rebuild Class Indexes".

### Server Tools

In the server edition of ObjectDB, the Explorer supports transferring ObjectDB database files from the local machine to the server, and vice versa, using the "Upload to Server" and "Download from Server" commands from the "Tools" menu or from the context menu. In addition, you can use the "Start Database Server" command to start the server as an Explorer thread. The server configuration file is located automatically by the Explorer, but you can specify a configuration path explicitly using "Tools | Options | General".

## 9.5.  Options and Settings

The Explorer settings are organized in a an "Options" dialog box with tabbed pages: "General", "SSL" (in the Server Edition), "Fonts" and "Views". You can open the "Options" dialog box using the "Tools | Options" menu command.

**The "General" Page**

The "General" page contains various options. The "Encoding" combo box should reflect the encoding of strings in the database. ObjectDB stores String instances in the database using the same encoding that they have in memory, which is usually Unicode. If you store String instances in the database that have a different memory encoding you have to set that encoding to manage these strings in the Explorer. This requirement is applied only to the Explorer. In your applications, retrieved String instances always have the same encoding as they had when they were stored, and therefore, there is no need to specify the encoding during retrieval.

The "General" group of check boxes contains the following options:

- Whether or not to open the last locally opened database file when the Explorer starts.
- Transaction type (datastore or optimistic) on the open database (see Section 5.3).
- Whether or not to include instances of subclasses when browsing the Extent of a class.

The "Display" group of check boxes contains the following options:

- Default viewer window (Table or Browser), as discussed in Section 9.2.
- Whether or not to use different colors for open an close nodes in a Browser window.
- Whether to use multi line string editor or single line string editor by default.

When the "Classpath for persistent classes and metadata" field is set, the Explorer uses that path to locate persistent classes and JDO metadata. Setting this field is not mandatory because you can browse and edit ObjectDB database files when class and metadata files are not available. Some features of the Explorer, however, do require setting the classpath. For instance, executing JDOQL queries containing user defined methods can be supported by the Explorer only when the code of these methods is available using the specified classpath.

The "Server Configuration Path" field is displayed only by the Server edition. If not initialized, it is automatically set to the path of the default server configuration file the first time the server is run in the Explorer, but you can always change the path to use a different configuration file.

**The "SSL" Page**

The "SSL" page (which is displayed only in the Server Edition) contains definitions for connecting to an ObjectDB server using SSL. The settings in this page are used only when the "Use SSL" check box in the "Open Remote Database" dialog box (which is shown by the "File | Open Remote" menu command) is checked. You can learn more about using SSL with ObjectDB in section 8.4. Notice that all the keystore and trustore details that are specified in this options page refer to the client side connection. When running an ObjectDB server as an Explorer thread its SSL setting is loaded from the server configuration file as usual.

**The "Fonts" Page**

The "Fonts" page is used to set the appearance of different Explorer components. Select one or more elements on the left side and then use the combo boxes on the right side to choose font face, font size, font style, background color and foreground color. Click the "Reset" button to apply the default settings to the selected elements. To discard all changes and apply the default settings to all the elements, click the "Reset All" button.

**The "Views" Page**

In the "Views" page you can select the persistent fields that are shown in the viewer windows and their order of appearance. This is especially useful when working with classes with a large number of fields where displaying all the fields is problematic (for instance, in a row in the table viewer).

There are three views. The "Table View" determines which fields in each persistent class are displayed as columns in Table viewer windows. The "Browser View" determines which fields are displayed as child nodes when browsing persistent objects using the browser viewer windows. The "Summary" view is also used for the browser viewer. It determines which fields are displayed as a summary of a persistent object in the node that represents the object itself. The "Table View" and the "Browser View" are initialized to contain all the persistent fields, but the "Summary" is initialized as an empty view. Therefore, unless set, persistent objects are presented in the Browser viewer as an empty set of fields (using "{}").

To set a view for a class, first select one of the three supported views and then select the persistent class in the list of classes. You can change the visibility of persistent fields of the class (in the selected view) using the Left and Right arrow buttons to move fields between the "Shown Fields" and the "Hidden Fields" lists. You can change the order of the shown fields by using the Up and Down arrow buttons or the "Field Ordering" combo box. A view for a class can also be set by right clicking one of its instances in the a viewer window (table or browser) and using the "Set View" context menu command.

# Index