OBJECTSTORE

JAVA API
USER GUIDE

RELEASE 3.0

October 1998

ObjectStore Java API User Guide

ObjectStore Java Interface Release 3.0, October 1998

Object Design, Inc.
Twenty Five Mall Road
Burlington, MA 01803-4194

Contents

Preface

Purpose

The *ObjectStore Java API User Guide* provides information and instructions for using the Java interface to ObjectStore. The Java interface allows you to write Java applications that store and retrieve data in ObjectStore databases.

Audience

This book is for experienced Java programmers who want to write applications that use the Java interface to ObjectStore.

Scope

This book does not provide information for developing ObjectStore applications that use C++ and Java. See *Developing ObjectStore Java Applications That Access C++*.

This book supports Release 3.0 of the Java interface to ObjectStore. See the **README.htm** file in the directory in which you installed ObjectStore for specific ObjectStore release numbers.

How This Book Is Organized

This book is organized as follows:

- Chapter 1, Introducing ObjectStore, on page 1, describes what ObjectStore does, shows the application architecture, and defines some important terms.
- Chapter 2, Example of Using ObjectStore, on page 19, describes the components your application must include to use ObjectStore.
- Chapter 3, Using Sessions to Manage Threads, on page 27, discusses how to initialize threads to use ObjectStore and how to use threads with ObjectStore sessions.
- Chapter 4, Managing Databases, on page 67, provides instructions for creating, opening, closing, garbage collecting, and upgrading databases.

- Chapter 5, Working with Transactions, on page 109, describes how to start and end transactions.
- Chapter 6, Storing, Retrieving, and Updating Objects, on page 127, discusses the steps for storing, retrieving, and updating data.
- Chapter 7, Working with Collections, on page 191, provides information about eachhows how to create collections of objects and run queries over the collections.
- Chapter 8, Automatically Generating Persistence-Capable Classes, on page 235, describes how to use the class file postprocessor to create persistence-capable classes.
- Chapter 9, Manually Generating Persistence-Capable Classes, on page 289, describes how to manually annotate classes you define so they are persistence-capable.
- Chapter 10, Controlling Concurrency, on page 317, describes the APIs you can use to ensure your application's access to data.
- Chapter 11, Using the Notification Facility, on page 341, discusses the system you can set up to notify sessions when a previously defined event has taken place.
- Chapter 12, Miscellaneous Information, on page 359, discusses serialization, **String** literals, storage layout, differences from C++ interface, and Java-supplied persistence-capable classes.
- Chapter 13, Tools Reference, on page 377 provides reference information for the ObjectStore utilities: **osgc**, **osjcfp**, **osjcgen**, **osjcheckdb**, **osjshowdb**, and **osjversion**.
- Appendix, Packaging Your Application for End Users, on page 405, provides instructions for which files you must include when you distribute your application.

Documentation Conventions

This document uses the following conventions:

Convention	*Meaning*
Bold	Bold typeface indicates user input, code fragments, method signatures, file names, and object, field, and method names.
Sans serif	Sans serif typeface is used for system output and system output.
Italic sans serif	Italic sans serif typeface indicates a variable for which you must supply a value. This most often appears in a syntax line or table.
Italic serif	In text, italic serif typeface indicates the first use of an important term.
[ ]	Brackets enclose optional arguments.
{ *a* | *b* | *c* }	Braces enclose two or more items. You can specify only one of the enclosed items. Vertical bars represent OR separators. For example, you can specify *a* or *b* or *c*.
...	Three consecutive periods indicate that you can repeat the immediately previous item. In examples, they also indicate omissions.

Examples in the documentation

Examples in the documentation assume that **COM.odi.*** is imported. This allows specification of, for example,

```
db.open(ObjectStore.READONLY)
```

instead of

```
db.open(COM.odi.ObjectStore.READONLY)
```

Internet Sources of More Information

World Wide Web

Object Design's support organization provides a number of information resources. These are available to you through a web browser such as Internet Explorer or Netscape Navigator. You can obtain information by accessing the Object Design home page with the URL **http://www.objectdesign.com**. Select **Tech Support**. Select **Tech Support Information** for detailed instructions about different methods of obtaining information from support.

Internet gateway

You can obtain such information as frequently asked questions (FAQs) from Object Design's Internet gateway machine as well as from the web. This machine is called **ftp.objectdesign.com** and its Internet address is 198.3.16.26. You can use **ftp** to retrieve the FAQs from there. Use the login name **objectdesignftp** and the password obtained from **patch-info**. This password also changes monthly, but you can automatically receive the updated password by subscribing to **patch-info**. See the ObjectStore 5.1 **README** file for guidelines for using this connection. The FAQs are in the **/support/FAQ** subdirectory. This directory contains a group of subdirectories organized by topic. The **FAQ/FAQ.tar.Z** file is a compressed **tar** version of this hierarchy that you can download.

Automatic email notification

In addition to the preceding methods of obtaining Object Design's latest patch updates (available on the **ftp** server as well as the Object Design Support home page), you can now automatically be notified of updates. To subscribe, send email to **patch-info-request@objectdesign.com** with the keyword **SUBSCRIBE patch-info <***your siteid***>** in the body of your email. This will subscribe you to Object Design's patch information server daemon that automatically provides site access information and notification of other changes to the on-line support services. Your site ID is listed on any shipment from Object Design, or you can contact your Object Design Sales Administrator for the site ID information.

Email discussion list

There is a majordomo discussion list called **osji-discussion**. The purpose of this list is to facilitate discussion about the Java interface to ObjectStorePSE and PSE Pro for Java. For subscription information, see *ObjectStore Java Interface Release Notes*, Description of Discussion List.

Support

Object Design's support organization provides a number of information resources and services. Their home page is at **http://support.objectdesign.com/WWW/Welcome.html**. From the support home page, you can learn about support policies, product discussion groups, and the different ways Object Design can keep you informed about the latest release information — including the Web, **ftp**, and email services.

Training

You can obtain information about training courses from the Object Design Web site (**http://www.objectdesign.com**). From the home page, select **Services** and then **Education**.

If you are in North America, for information about Object Design's educational offerings, call 781.674.5000, Monday through Friday from 8:30 AM to 5:30 PM Eastern Time.

If you are outside North America, call your Object Design sales representative.

Your Comments

Object Design welcomes your comments about ObjectStore documentation. Send feedback to **support@objectdesign.com**. To expedite your message, begin the subject with **Doc:**. For example:

Subject: Doc: Incorrect message on page 76 of reference manual

You can also fax your comments to 781.674.5440.

Chapter 1
Introducing ObjectStore

ObjectStore provides an application programming interface (API) that allows you to persistently store Java objects.

Contents

This chapter discusses the following topics:

What Is ObjectStore?

ObjectStore is an object-oriented database management system. It allows you to

- Manipulate information in the database transparently by creating and modifying persistent Java objects.
- Store and access data in the same format as it exists in the application.
- Describe, store, and query complex data used in sophisticated software applications, as well as data traditionally managed by relational database applications.
- Persistently store data independent of the data type.

The Java interface to ObjectStore Development Client (referred to as ObjectStore) is for Java and C++ applications that require multiuser high-performance persistent storage for large databases with enterprise database features such as failover, on-line backup, fine-grained concurrency, and security.

ObjectStore can work well for distributed databases of virtually unlimited sizes and unlimited numbers of objects. ObjectStore supports the following:

- Applications that interface with databases and servers on local or remote machines
- Java applications and C++ applications that can access the same data
- Multiple concurrent users
- Collections with indexed look-ups and queries
- Databases that consist of multiple segments, which are variable-sized regions of disk space that ObjectStore uses to cluster objects stored in the database
- Operating on multiple databases in a transaction
- Cross-database and cross-segment references
- On-line backup, failover, and archive logging

ObjectStore PSE for Java and ObjectStore PSE Pro for Java are personal storage editions (subsets) of the Java interface to ObjectStore. It is possible to use multiple ObjectStore Java products in the same Java virtual machine (VM). The **COM.odi.product** property allows you to do this.

ObjectStore includes the **COM.odi.odmg** package, which provides an Object Data Management Group (ODMG) binding. This binding includes classes for **Database** and **Transaction** that closely follow the ODMG specification. The package also includes the **COM.odi.odmg.Collection** interface, persistence-capable classes that implement the **Collection** interface, and ODMG exception classes. See the **COM.odi.odmg** package in the *ObjectStore Java API Reference*.

What ObjectStore Does

ObjectStore provides an API that allows a program to

- Start and end sessions to allow threads to use the ObjectStore API.
- Create, open, close, and destroy databases.
- Start, commit, and abort transactions to access data in the database.
- Read and write database roots, which provide starting points for navigating to persistent objects.
- Store objects in a database and retrieve and update those objects.

ObjectStore can recover from an application failure or system crash. If a failure prevents some of the changes in a transaction from being saved to disk, ObjectStore ensures that none of that transaction's changes are saved in the database. When you restart the application, the database is consistent with the way it was before the transaction started.

With ObjectStore's archive logging facility, you can protect against media failure. See *ObjectStore Management*.

Benefits of Using ObjectStore

ObjectStore provides a convenient and complete API for storing and sharing Java objects among users, hosts, and programs. After you define persistence-capable classes (classes whose instances can be stored in a database), writing an ObjectStore application is just like writing any other Java application.

ObjectStore allows you to quickly read or modify portions of your persistent data. You are not required to read in all persistent data when you just want to look at a subset. This reduces start-up and transaction commit times and allows you to run much larger Java applications without increasing the amount of memory or swap space on the system.

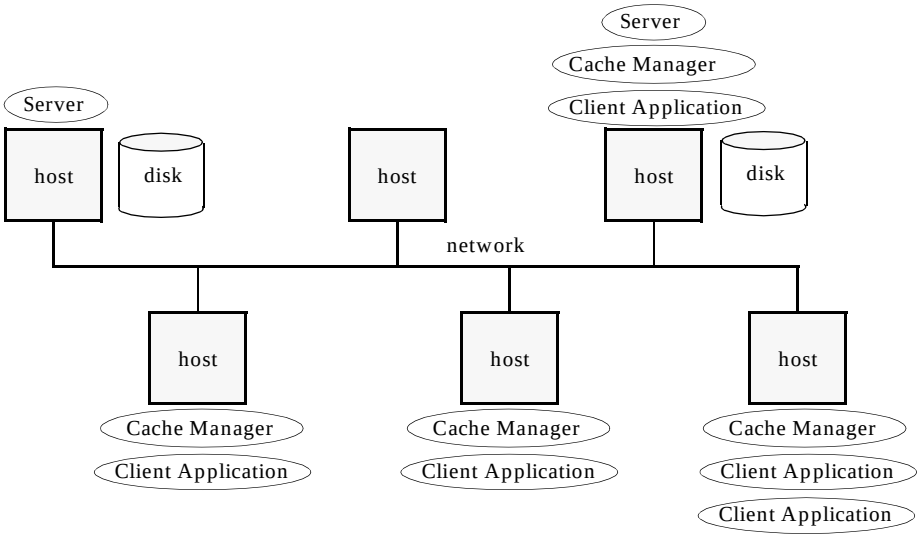
When you access persistent data inside a transaction, ObjectStore ensures that your results are not compromised by other users sharing the data. If something goes wrong, or if you determine that you do not want to keep changes, you can abort the transaction. In that case, ObjectStore restores the database to its state before the transaction started. This makes it straightforward to recover from exceptions or failures.

Description of ObjectStore Process Architecture

See Committing Transactions to Save Modifications on page 153.There are three kinds of processes in the ObjectStore environment:

- The *Server* is the ObjectStore process that controls object storage. The Server can manage databases for multiple client applications, which might be on multiple hosts.
- The *client* is the process in which ObjectStore links the ObjectStore client library into each ObjectStore application. In this way, each ObjectStore application is an ObjectStore client.
- The *Cache Manager* facilitates concurrent access to data by handling callback messages from the Server to client applications.

The process architecture for the Java interface to ObjectStore is the same as for the C++ interface to ObjectStore. The Server and the Cache Manager are the same, regardless of the language interface you use with ObjectStore. See *ObjectStore Management* for detailed information about managing these processes. The following figure shows the process architecture.



In the Java interface to ObjectStore, the client is a single process that has several software components. These components call each other as needed:

- ObjectStore C++ client library, which, among other things, communicates with the Server
- ObjectStore Java client library, which provides the ObjectStore Java API
- Your Java application

The client library and the Cache Manager work together to maintain local copies of data on the client machine. Since local access is a great deal faster than remote access, this improves application performance. ObjectStore ensures that you can never retrieve stale data, so you obtain performance benefits without sacrificing accuracy.

Definitions of ObjectStore Terms

Here are some terms you must be familiar with to use ObjectStore:

Session

A *session* allows the use of the ObjectStore API. ObjectStore uses the abstract **COM.odi.Session** class to represent sessions.

Your application must create a session before it can use any of the ObjectStore API. After a session is created, it is an active session. A session remains active until your application or ObjectStore terminates it. After a session is terminated, it is never used again. You can, however, create a new session.

A session creates a context in which you can create a transaction, access a database, and manipulate persistent objects. A session consists of a set of persistent objects, and a set of ObjectStore API objects, such as a **Transaction**, **Database**s, and **Segment**s. In a single Java VM process,

- PSE allows one session at a time.
- PSE Pro allows multiple concurrent sessions.
- ObjectStore allows one session at a time. It is expected that ObjectStore will allow multiple sessions in a future release.

Separate Java virtual machines can each run their own session at the same time. In addition, if you are using PSE Pro, separate Java virtual machines can each run multiple sessions at the same time. See How Sessions Keep Threads Organized on page 28.

Persistence-Capable

The term *persistence-capable* refers to the capacity of an object to be stored in a database. If you can store an object in a database, the object is persistence-capable. If you can store the instances of a class in a database, the class is a persistence-capable class and the instances are persistence-capable objects.

The definition of a persistence-capable class includes specific annotations required by ObjectStore. After you compile class definitions, you run the ObjectStore class file postprocessor on the compiled classes to add the annotations that make the classes persistence-capable. See Chapter 8, Automatically Generating Persistence-Capable Classes, on page 235. In unusual circumstances, you might choose to manually add the annotations to the Java source file. See Chapter 9, Manually Generating Persistence-Capable Classes, on page 289.

You must explicitly postprocess or manually annotate each class that you want to be persistence-capable. The capacity for an object to be stored in a database is not inherited when you subclass a persistence-capable class.

Some Java-supplied classes are persistence-capable. Other classes are not and cannot be made persistence-capable. A third category of classes can be made persistence-capable, but there are important issues to consider when you do so. Be sure to read Java-Supplied Persistence-Capable Classes on page 360.

Persistent Object

A *persistent object* is a representation of an object that is stored in a database.

After an application retrieves an object from the database, the application works with the persistent object in the Java environment. A persistent object always exists in one of three states:

- Hollow
- Active
- Stale

Methods you call can change the state of a persistent object.

Hollow persistent object

A *hollow persistent object* has the same structure as the object in the database that it represents. A hollow object contains the same fields as the object in the database that the persistent object represents, but the fields of the hollow object have default values.

When your application acquires a reference to an object that has not yet been read in from the database, ObjectStore generates a hollow object as a placeholder for that object. ObjectStore does not actually read in the contents of the object until your application tries to read, write, or invoke a method on the object.

When your application accesses a hollow object, ObjectStore turns it into an active persistent object. ObjectStore retrieves the contents of the object from the database and stores them in the fields of the hollow object, which makes it an *active* persistent object. This process is referred to as initialization.

Obtaining an object from a database root always results in a hollow object. If you get the same root three times, the object it identifies is still hollow. You must access the object to make it active.

After an application accesses the contents of a persistent object, the objects that the persistent object references are hollow objects, unless their contents were previously accessed. For example, suppose you have the following class:

```
class A {
   B b;
}
```

When you obtain a reference to an instance of **A**, ObjectStore creates a hollow **A** object to represent that instance. When you read or update the instance of **A**, ObjectStore turns it into an active object and creates a hollow **B** object to represent the referred to instance of **B**. If you then read or update the instance of **B**, ObjectStore makes that hollow object into an active object and creates hollow objects for any objects referred to by **B**.

Active persistent object

An *active persistent object* starts as an exact copy of the object that it represents in the database. The contents of an active object are available to be read by the application and might be available to be modified. If an active object is updated by the application, it is no longer identical to the object in the database that it represents.

An application can read or update an active persistent object; a persistent object must be active for an application to read or update it. The postprocessor takes care of this for classes that it annotates.

When ObjectStore changes a hollow object to an active object, it initializes the object. In most applications, this happens automatically, because the postprocessor inserts the required calls.

When a persistent object is active, ObjectStore internally flags it as either clean or dirty. An active object is initially marked as clean when its contents are read into memory. At this point, ObjectStore recognizes that the contents of the persistent object match the contents of the object in the database. An active object is dirty when it is a modified version of the stored object that the active object represents. When you modify an object, ObjectStore automatically changes the flag from clean to dirty. The class file postprocessor inserts the code that does this.

For example, suppose you have an instance of a **Person** object where the **age** field has the value **30**. When you read this object, it is in the clean state. If you modify the value of **age**, even if the new value you assign is **30**, the object is then in the dirty state.

Stale persistent object

A *stale persistent object* is no longer valid. Its fields have default values and should not be used. An object becomes stale when an application calls

- **Transaction.commit()** on the transaction in which the object could be read or modified, and the call does not specify a **retain** argument, or it specifies **ObjectStore.RETAIN_STALE.**

  (There is no API that sets a default **retain** value for the **commit()**, **evict()**, or **destroy()** methods. You can only set a default **retain** value for the **abort()** method.)

- **Transaction.abort()** on the transaction in which the persistent object could be read or modified, and the call does not specify a **retain** argument (and the default **retain** value is **ObjectStore.RETAIN_STALE**), or it specifies **ObjectStore.RETAIN_STALE**
- **ObjectStore.evict()** on the object and the call does not specify a **retain** argument, or it specifies **ObjectStore.RETAIN_STALE**
- **ObjectStore.destroy()** on the object

An object can also become stale when the transaction in which it was accessed is aborted because of a deadlock or some other action that caused AbortException to be thrown.

If an application tries to read or update a stale object, ObjectStore throws ObjectException. An application must not invoke any instance method on a stale object.

Persistence-Aware

If the methods of a class can operate on fields of persistent objects, but instances of the class itself are not persistence-capable, the class is *persistence-aware*. Typically, if you want a class to be persistence-aware, you run the postprocessor on it to put in the required annotations. See Chapter 8, Automatically Generating Persistence-Capable Classes, on page 235. Occasionally, you might choose to manually annotate the class to make it persistence-aware. See Chapter 9, Manually Generating Persistence-Capable Classes, on page 289.

When a method accesses fields in a persistent object, ObjectStore checks to ensure that the data has been read from the database. This checking is done by calls that the postprocessor inserts in your code. These are the annotations mentioned in the previous paragraph. The annotations are calls to the **ObjectStore.fetch()** or **ObjectStore.dirty()** method.

Every persistence-capable and persistence-aware class must have these annotations. Persistence-capable classes also include many other annotations.

A class must be persistence-aware only if it directly accesses the fields of a persistence-capable object. This includes access of elements of a persistent array. If your persistence-capable classes have only private fields and do not return arrays that might be persistent, other classes can call methods on the persistence-capable object without having to be persistence-aware.

Primary Object

A *primary object* is a persistence-capable Java object. A persistence-capable object is an object that can be stored in an ObjectStore database. You can use primary objects just as if they were ordinary Java objects. The database correctly records their identities, classes, and field values. Primary objects do not extend the **CPlusPlus** class.

Peer Object

Peer objects provide a way for Java applications to use C++ objects. A peer object acts as a proxy for a particular C++ object. It has no data fields and so it does not hold any state that is represented by the data members of the corresponding C++ object.

However, a peer object provides object identity, which allows you to invoke a method on the corresponding C++ object. You can call Java methods on a peer object to invoke all public methods of the original C++ object. You can think of a peer object as a handle to a C++ object.

Each peer object identifies exactly one C++ object. Multiple peer objects can represent the same C++ object. For example, each element of a C++ array of classes is represented by a peer object.

A peer object is an instance of a Java peer class. All peer objects extend the **CPlusPlus** class. Information about using peer objects is in *Developing ObjectStore Java Applications That Access C++*.

Transient Object

A *transient object* is an object that is not already in a database.

Transitive Persistence

When an application commits a transaction, ObjectStore stores in the database any transient objects that can be reached transitively from any persistent object. This is the process of *transitive persistence*. Transient objects that are referenced by persistent objects become persistent when the transaction commits. For this to work, the transient objects must be persistence-capable.

Annotations

The class file postprocessor annotates classes you define so that they are persistence-capable. This means that the postprocessor makes a copy of your class files, overwrites your original class files or places them in a directory you specify, and adds byte code instructions (*annotations*) that are required for persistence. Complete information about annotations is in Chapter 8, Automatically Generating Persistence-Capable Classes, on page 235.

Occasionally, you might want to manually annotate your code. Information you need to do this is in Chapter 9, Manually Generating Persistence-Capable Classes, on page 289.

Database Roots

A *database root* provides a way to associate a name with an object in a database. Applications use database roots to locate one or more persistent objects for performing queries or navigating to other persistent objects. When you make an object the value of a persistent database root, doing so establishes the object as persistent and makes the objects it refers to available for transitive persistence.

At any given time, a database root is either associated with one database or it is null. You can change the database with which a root is associated. Information about database roots is in Working with Database Roots on page 131.

Prerequisites for Using the ObjectStore Java Interface

To use the ObjectStore Java interface you must

- Be an experienced programmer familiar with the Java language.
- Have available a supported Java platform. The compiler must conform to JavaSoft specifications. The Java VM must be among those supported by ObjectStore. You cannot use a supported compiler with an unsupported VM. See Requirements for Using This Release in the release notes.
- Have installed ObjectStore Server and the C++ interface to ObjectStore Development Client.

You should also be familiar with the information in the *ObjectStore Management* manual, which describes the ObjectStore architecture and provides information about ObjectStore Server parameters, ObjectStore client environment variables, and ObjectStore utilities that you can use.

If you plan to use ObjectStore to operate on both Java and C++ objects, then you must also be an experienced C++ programmer.

Chapter 2
Example of Using ObjectStore

This chapter provides a simple example of a complete ObjectStore program. The code for this example is in the **COM\odi\demo\people** directory provided with ObjectStore.

Contents

This chapter discusses the following topics:

Overview of Required Components

The sample program stores some information about a few people and then retrieves some of the information from the database and displays it. The program shows the components you must include in your application so that it can use ObjectStore. These components are

- Create a session. The example calls the **Session.create()** method to start a nonglobal session. See page 34.
- Join a thread to a session. The example calls the **Session.join()** method to associate this thread with the session. See page 46.
- Create or open a database. The example creates the **person.odb** database and uses the **db** variable to refer to it. See page 68.
- Start and commit transactions as needed. The example uses one transaction to store the objects in the database. It then uses a second transaction to retrieve the stored objects.See page 110.
- Create a database root, which provides a starting point for accessing objects in the database. The example creates a root with the name **"Tim"** and associates it with the **tim** instance of the **Person** class. See page 131.
- Store objects referenced by a root in the database. The example stores **sophie** and **joseph** in the database when the transaction is committed. See page 128.
- Use a database root to retrieve objects from a database and do something with them. The example starts a new transaction, retrieves **tim**, and displays a line of information.See page 137.
- End the session. The example calls the **Session.terminate()** method. This closes the open database and also shuts down ObjectStore. See page 39.

When you write an ObjectStore program, you write it as though classes are persistence-capable. However, a program cannot store objects persistently until you run the ObjectStore-provided class file postprocessor. The postprocessor generates annotated versions of the class files. The annotated version of the class definition is persistence-capable. You run the postprocessor after you compile the program and before you run the program.

Sample Code

```
package COM.odi.demo.people;

// Import the COM.odi package, which contains the API:
import COM.odi.*;

public
class Person {

  // Fields in the Person class:

  String name;
  int age;
  Person children[];

  // Main:

  public static void main(String argv[]) {
   try {
     String dbName = argv[0];

     // The following line starts a nonglobal session and joins this
     // thread to the new session. This allows the thread to use
     // ObjectStore.
     Session.create(null, null).join();

     Database db = createDatabase(dbName);
     readDatabase(db);
     db.close();
   }

   // The following shuts down ObjectStore.
   finally {
       Session.getCurrent().terminate();
   }
  }

  static Database createDatabase(String dbName) {

   // Attempt to open and destroy the database specified on the
   // command line. This ensures that the program creates a
   // new database each time the application is called.

   try {
   Database.open(dbName, ObjectStore.OPEN_UPDATE).destroy();
   } catch (DatabaseNotFoundException e) {
   }

   // Call the Database.create() method to create a new database.

   Database db = Database.create(dbName,
   ObjectStore.ALL_READ | ObjectStore.ALL_WRITE);
   // Start an update transaction:

   Transaction tr = Transaction.begin(ObjectStore.UPDATE);
```

```
 // Create instances of Person:

 Person sophie = new Person("Sophie", 5, null);
 Person joseph = new Person("Joseph", 1, null);
 Person children[] = {sophie, joseph};
 Person tim = new Person("Tim", 35, children);

 // Create a database root and associate it with
 // tim, which is a persistence-capable object.
 // ObjectStore uses a database root as an entry
 // point into a database.

 db.createRoot("Tim", tim);

 // End the transaction. This stores the three person objects,
 // along with the String objects representing their names, and
 // the array of children, in the database.

 tr.commit();

 return db;
}

static void readDatabase(Database db) {

 // Start a read-only transaction:

 Transaction tr = Transaction.begin(ObjectStore.READONLY);

 // Use the "Tim" database root to access objects in the
 // database. Because tim references sophie and joseph,
 // obtaining the "Tim" database root allows the program
 // also to reach sophie and joseph.
 Person tim = (Person)db.getRoot("Tim");
 Person children[] = tim.getChildren();
     System.out.print("Tim is " + tim.getAge() + " and has " +
         children.length + " children named: ");
 for (int i=0; i < children.length; i++) {
   String name = children[i].getName();
   System.out.print(name + " ");
 }
 System.out.println("");
 // End the read-only transaction.
 // This form of the commit method ends the accessibility
 // of the persistent objects and makes the objects stale.
 tr.commit();
}

// Constructor:

public Person(String name, int age, Person children[]) {
 this.name = name; this.age = age; this.children = children;
}

public String getName() {return name;}
public void setName(String name) {this.name = name;}
public int getAge() {return age;}
```

```
    public void setAge(int age) {this.age = age;}
    public Person[] getChildren() {return children;}
    public void setChildren(Person children[]) {
       this.children = children;
    }

    // This class is never used as a persistent hash key, so
    // include the following definition. If you do not, then
    // when you run the postprocessor it is unclear whether or
    // not you intend to use the class as a hash code.
    // Consequently, the postprocessor inserts a hashCode
    // function for you. The following definition avoids this.
    public int hashCode() {
     return super.hashCode();
    }

}
```

Before You Run the Program

Before you can run the sample program, you must

- Add an entry to your **CLASSPATH** environment variable
- Compile the source file
- Run the postprocessor on the **.class** file

Adding An Entry to CLASSPATH

In your **CLASSPATH** environment variable, you already have two entries related to ObjectStore:

- An entry for the **osji.zip** or **osji.jar** file to use ObjectStore
- An entry for the **tools.zip** or **tools.jar** file to use the class file postprocessor and other database tools

Ensure that these zip files are explicitly in your class path. An entry for the directory that contains them is not sufficient.

Another entry is required for you to be able to build and run the program. This entry names the ObjectStore installation directory, and allows ObjectStore to locate the annotated class files when you run the program.

For example, on Windows, if you place the ObjectStore distribution in the **c:\odi\osji** directory, you need the following entries:

c:\odi\osji\osji.zip;c:\odi\osji\tools.zip;c:\odi\osji

On UNIX, if you place the ObjectStore distribution in **/usr/local/osji**, you need

/usr/local/osji/osji.zip:/usr/local/osji/tools.zip:/usr/local/osji

Compiling the Program

To compile the program, change to the **COM\odi\demo\people** directory and enter

javac *.java

As output, the **javac** compiler produces the byte code class file **Person.class**.

Running the Postprocessor

You must run the class file postprocessor to make the **Person** class persistence-capable. The postprocessor generates new annotated class files. After you run the postprocessor, your program uses the annotated class files and not the original class files.

Ensure that the **bin** directory that contains the **osjcfp** executable is in your path, as noted in the **README** file in the installation directory and the postprocessor documentation. See Preparing to Run the Postprocessor on page 243.

On Windows, to run the postprocessor, enter

osjcfp -dest . -inplace Person.class

On UNIX, to run the postprocessor, enter

osjcfp -dest . -inplace Person.class

The **-dest** option specifies a destination directory for the annotated files. It is a required option. The **-inplace** option specifes that the postprocessor should overwrite the original class files. When you specify the **-inplace** option, the postprocessor ignores the **-dest** option.

The result from the **osjcfp** command shown above is

- The annotated class file **COM\odi\demo\people\Person.class**
- The **PersonClassInfo.class** file, also in the **people** directory

The **-inplace** option is the best choice for this example. But when you are in an iterative development cycle, it is best not to specify **-inplace**. During development, putting the postprocessed files in a different directory avoids errors.

Running the Program

Run the program as a Java application. Here is a typical command line:

java COM.odi.demo.people.Person person.odb

The argument is the pathname of the database.

The expected output is

```
Tim is 35 and has 2 children named: Sophie Joseph
```

Also, the example application creates or replaces the **person.odb** database in the current directory.

Chapter 3
Using Sessions to Manage Threads

This chapter provides information about how to manage the threads in your application. Sample code that uses threads is in **COM\odi\demo\threads**.

Contents

This chapter discusses the following topics:

How Sessions Keep Threads Organized

For a thread to use ObjectStore, it must be associated with a session. To use threads with ObjectStore, you must create at least one session and you must understand how to work with sessions.

If you try to use the ObjectStore API and you have not created a session, ObjectStore throws ObjectStoreException.

This section discusses the following information about sessions:

What Is a Session?

A session allows the use of the ObjectStore API. ObjectStore uses the abstract **COM.odi.Session** class to represent sessions.

Your application must create a session before it can use any of the ObjectStore API. After a session is created, it is an active session. A session remains active until your application or ObjectStore terminates it. After a session is terminated, it is never used again. You can, however, create a new session.

A session creates a context in which you can create a transaction, access one or more databases, and manipulate persistent objects.

Concurrent sessions

In a single Java VM,

- PSE allows one session at a time.
- PSE Pro allows multiple concurrent sessions.
- ObjectStore allows one session at a time in Release 1.3. It is expected that ObjectStore will allow multiple sessions in a future release.

If you are using ObjectStore, separate Java virtual machines can each run their own session at the same time. If you are using PSE Pro, separate Java virtual machines can each run multiple sessions at the same time. However, the default behavior is that at any one time only one Java VM process can access a database. See Description of Concurrency Rules on page 56.

How Are Threads Related to Sessions?

At any given time, an active session has zero or more associated threads. Any number of threads can join a session. Each thread can belong to only one session at a time.

At any given time, each thread is either joined to a single session or not joined (not associated) with a session. A thread that is not associated with a session can join a session. A thread that is associated with a session can leave the session to end its association with that session. It can rejoin the session at a later time or it can join some other session.

For a thread to use the ObjectStore API, it must be automatically or explicitly associated with a session. All threads that join the same session cooperate with each other. ObjectStore does not prevent cooperating threads from accessing the same object. Consequently, it is your responsibility to identify code segments that must be synchronized. To successfully call the **Session.join()** method to join a session, a thread must not already be associated with any session.

Current thread and current session

The *current thread* is the thread that you are making a call from. The *current session* is the session the current thread belongs to.

What Is the Benefit of a Session?

The benefit of a session is apparent when you want to have more than one session. Two sessions in the same Java process allow you to perform two distinct activities that involve ObjectStore. Each session has a clean, isolated view of the database. If you want to have two or more independent transactions going on at the same time, you can use two or more sessions. Concurrent sessions can be accessing the same database or different databases.

When two sessions are accessing the same object in the database, there are two distinct persistent objects. Each session has its own persistent object, which is a copy of the object in the database. At least initially, these two persistent objects have the same content.

Each session has its own set of persistent objects and API objects. In most circumstances, the threads of session **A** are not allowed to operate on the persistent objects of session **B**. An exception to this rule is described in Multiple Representations of the Same Object on page 53.

Independent threads

A need for many different independent transactions normally arises because you have many Java threads with different things going on in each one. Typically, this happens with a multithreaded application server, in which there are many threads. Each thread serves a different client, so you might want to have many threads. Each thread runs a separate transaction. Each thread is separate from each other thread.

Cooperating threads

On the other hand, there are times when you have multiple threads that are cooperating on some database task, and must operate on the same objects at the same time. In this case, you might want two different Java threads to participate in the same transaction.

Controlling which threads cooperate

Sessions allow you to control which threads cooperate in a transaction and which threads work in independent transactions. A session groups together a set of cooperating threads. Each session has a sequence (in time) of transactions, and a set of associated threads that participate in these transactions.

There is a many-to-one relationship between threads and sessions. That is, any number of threads can belong to one session.

Example of cooperating threads

A common case of cooperating threads arises when you are writing a Java applet. In an applet, there are calls to different parts of your program in different threads. You have to specify for ObjectStore that all these threads are part of the same session. This allows them to operate on the same objects and in the same transactions. A similar situation exists when you use RMI and CORBA servers. That is, there is a control mechanism that calls your methods in different threads.

What Kinds of Sessions Are There?

An active session can be a global session or a nonglobal session. ObjectStore provides two kinds of sessions because, when you only need one session, there are many things ObjectStore can do for you automatically.

Joining threads to sessions

As mentioned earlier, before you can use ObjectStore, you must create a session. For a thread to use ObjectStore, it must join a session. In a global session, an unassociated thread that makes a call to the ObjectStore API automatically joins the session. In a nonglobal session, this happens only when the call implies the session. See Rules for Automatically Joining a Thread to a Session on page 43. Otherwise, you must explicitly add the thread to a session.

Number of sessions

When there is an active global session, it is the only session in the Java VM. With PSE Pro and a future release of ObjectStore, you can have multiple nonglobal sessions or one global session in a Java VM. In the current release of ObjectStore and with PSE, there can be one session in a Java VM. It can be global or nonglobal.

Global sessions

Global sessions make programming easier, because you do not need to know the ObjectStore APIs for associating threads with sessions. All threads that make ObjectStore API calls automatically join the one global session.

The drawback is that you can only have one session. If you ever change your program in the future to use multiple sessions, you might have to go back and put in API calls to associate threads with the appropriate session. If you think you might use multiple sessions in the future, it would probably be a good idea to prepare for that by using a nonglobal session, and explicitly joining the session in each thread.

Note: Automatic joining of threads to a global session is not working in this release. You must explicitly join a thread to a session. See *ObjectStore Java Interface Release Notes*, Known Problems, Threads Are Not Being Automatically Joined to Sessions.

Creating Sessions

When you create a session, you initialize ObjectStore for use by the threads that become associated with that session. There are three ways to create a session:

- Call the **Session.globalCreate()** method to create a global session.
- Call the **Session.create()** method to create a nonglobal session.
- Call the **ObjectStore.initialize(host, properties)** method to create a nonglobal session. This method is maintained for compatibility with previous releases. It might be deprecated in a future release.

Regardless of how you create a session, there are a number of ObjectStore properties you can specify when you create the session. These properties determine how ObjectStore behaves in a variety of situations.

Creating Global Sessions

When the session is a global one and a thread that is not associated with the session calls an ObjectStore API, ObjectStore automatically joins the thread to the session. After you create a global session, you do not need to be concerned about joining threads to the session.

To create a global session, call the **Session.createGlobal()** method. The method signature is

public static Session createGlobal(String host,
java.util.Properties properties)

This method creates and returns a new session and designates the session as a global session. There are no threads joined to this session yet. Any thread, including the thread that creates the session, automatically joins the session the first time the thread uses ObjectStore.

ObjectStore ignores the first parameter; you can specify null. The second parameter specifies null or a property list. See Description of ObjectStore Properties on page 57.

If you try to create a global session when there is already an active session, ObjectStore throws ObjectStoreException.

To obtain a global session, call **Session.getGlobal()**. The method signature is

public static Session getGlobal()

If the global session is active, ObjectStore returns it. Otherwise, ObjectStore returns null.

Creating Nonglobal Sessions

You can create a nonglobal session. The difference between a global and nonglobal session is that in a nonglobal session

- ObjectStore does not automatically join all unassociated threads to the session.
- There can be multiple nonglobal sessions in the same Java VM. (It is expected that ObjectStore will allow multiple sessions in a future release.)

Joining threads to sessions

For ObjectStore to automatically join an unassociated thread to a nonglobal session, the thread must be making an ObjectStore API call that implies a session. See Rules for Automatically Joining a Thread to a Session on page 43. You must explicitly join a thread to a session before that thread can call an ObjectStore API that does not imply a session. See Explicitly Associating Threads with a Session on page 46.

Method signature

The method signature for creating a nonglobal session is

```
public static Session create(String host,
  java.util.Properties properties)
```

This method creates and returns a new session. ObjectStore ignores the **host** argument; specify null. The second argument specifies null or a property list. See Description of ObjectStore Properties on page 57.

ObjectStore does not join the calling thread to the session.

Session name

ObjectStore generates a name for the session and never reuses that name for the lifetime of the process in which the session was created. If you want to specify a particular name, use the following overloading to specify a unique session name:

```
public static Session create(String host,
   java.util.Properties properties,
  String name)
```

ObjectStore uses the session name in debugging messages. The **Session.getName()** method returns the name of the session.

Exception conditions

If you call **Session.create()** when there is an active global session, ObjectStore throws ObjectStoreException. If you are using ObjectStore, and you call **Session.create()** when there is an active nonglobal session, ObjectStore throws FatalApplicationException.

Creating a Nonglobal Session with ObjectStore.initialize()

In previous releases of ObjectStore, the only way to create a session was to call the **ObjectStore.initialize(host, properties)** method. In this release, this method is provided for compatibility with earlier releases. This method might be deprecated in a future release. The method signature is

```
public static boolean initialize(String host,
  java.util.Properties properties)
```

This method creates a new session, joins the calling thread to the session, and returns true. The **host** argument can be null. The second parameter specifies null or a property list. See Description of ObjectStore Properties on page 57.

Comparison with Session.create()

When you use the **Session.create()** method in place of the **ObjectStore.initialize(host, properties)** method, you must also call the **Session.join()** method. While the **ObjectStore.initialize(host, properties)** method starts a session and joins the calling thread to the session, the **Session.create()** method only starts the session. It does not join the calling thread to the session.

False return value

When the three conditions listed below all exist, the **ObjectStore.initialize()** method returns false to indicate that it did not start a new session.

- The calling thread is already associated with a session.
- The associated session is not a global session.
- The **host** parameter in this invocation of the **initialize()** method is exactly the same as the **host** value that was specified when the session was created.

Exception conditions

If the calling thread already belongs to a session, but the host value specified in the **initialize()** method and the host value specified when the session was created are not the same, ObjectStore throws IllegalArgumentException.

If there is already a global session, ObjectStore throws ObjectStoreException.

If there is already an active session, ObjectStore throws FatalApplicationException.

Working with Sessions

After you create a session, you need to know how the session functions with regard to transactions. You also need to know about the operations you can perform on the session. This section discusses

Sessions and Transactions

At any given time, a session has one associated transaction in progress or it does not have any associated transaction. Each transaction is associated with exactly one active session.

When a session is created, there is no associated transaction. While a session is active, an application can start and then commit or abort one transaction at a time per session. Over time, a session is associated with a sequence of transactions.

If there is a transaction in progress when an application or ObjectStore shuts down the session, ObjectStore aborts the transaction as part of the shutdown process.

Within a session, multiple databases can be open at the same time.

All transactions can access the same database. All sessions must have read-only transactions against that database.

See also Description of Concurrency Rules on page 56.

Transaction in progress?

To determine whether or not there is a transaction in progress, call the **Session.inTransaction()** method. The method signature is

public boolean inTransaction()

If there is a transaction associated with the session, this method returns true. If there is no transaction associated with the session, this method returns false. If the session has been terminated, ObjectStore throws ObjectStoreException.

Obtaining associated transaction

To obtain the transaction associated with a session, call the **Session.currentTransaction()** method. The method signature is

public Transaction currentTransaction()

If the session has been terminated, ObjectStore throws ObjectStoreException. If no transaction is associated with the session, ObjectStore throws NoTransactionInProgressException.

Obtaining transaction's session

To obtain the session associated with a transaction, call the **Transaction.getSession()** method. The method signature is

public Session getSession()

Shutting Down Sessions

An application can shut down sessions. One reason you might want to shut down a session is to release the Java objects associated with the session.However, shutting down a session does not release all resources used by the C++ client. Such resources are released only when the application exits.To shut down a session, call the **Session.terminate()** method. The method signature is

public void terminate()

It does not matter whether the session is a global session or a nonglobal session. ObjectStore shuts down the session. If there are no other sessions, no thread can use the ObjectStore API until there is a new active session. The terminated session is never reused. If you are using ObjectStore, you must start a new session before you can use the ObjectStore API again.

Transaction in progress

If the session you shut down has an associated transaction, ObjectStore aborts the transaction. If the session has already been terminated, ObjectStore does nothing. If the session has any associated threads, ObjectStore causes them to leave the session. If the session has an open database, ObjectStore closes it.

ObjectStore. shutdown()

In previous releases, the way to shut down a session was to call the **ObjectStore.shutdown()** method. This method is maintained in this release for compatibility with earlier releases. It might be deprecated in a future release.

If ObjectStore throws FatalException, this shuts down the session.

Obtaining a Session

You can obtain a session with a call to any of the methods listed below:

- **Placement.getSession()**
- **Session.getCurrent()**
- **Session.getGlobal()**
- **Session.of(object)**
- **Session.ofThread(thread)**
- **Transaction.getSession(thread)**

Determining If a Session Is Active

To determine whether or not a session is active, call the **Session.isActive()** method. The method signature is

public boolean isActive()

If the session is active, this method returns true. If the session has been terminated, this method returns false.

Associating Threads with Sessions

To help you associate threads with sessions, this section discusses

There is a bug in the software that prevents threads from automatically being joined to sessions. As a work around, you must explicitly join each thread to a session. See the release notes for details. This bug will be fixed in a future release.

Automatically Joining Threads to a Session

Whether a thread can automatically join a session depends on

- Whether the session is global or nonglobal
- Whether or not the API call that the thread is making implies a session

Global sessions

When there is a global session, an unassociated thread that makes a call to the ObjectStore API automatically joins the global session, if necessary. In the following situations, it might not be necessary to join the thread to the session:

- An unassociated thread calls a method on a transient object and the method requires a persistent object. Since the object is not persistent, the method cannot do anything, and so it does not need to be joined to the session.
- An unassociated thread calls a method that does not operate on persistent objects, for example, calls to **ObjectStore.getAutoOpenMode()** and **ObjectStore.setLazyWriteLocking**.
- An unassociated thread calls a method that has already been executed. The thread might automatically join the session if it executes the method anyway. For example, when an unassociated thread tries to open a database that is already open, ObjectStore joins the thread to the session that the database belongs to, even though the thread doesn't actually do anything.

Nonglobal sessions

In a nonglobal session, ObjectStore automatically joins threads to the session when the call from the thread implies that session. This means that the call specifies an argument that is already associated with that session. This includes the object on which the method is invoked.

After ObjectStore automatically joins a thread to a session,

- The thread is associated with the session until you remove it from the session or the session terminates.
- ObjectStore performs the called method.

Associating a Persistent Object with a Session

How does an object become associated with a session? It happens implicitly. Assume that a thread is already associated with a session. This associated thread successfully calls an ObjectStore API. If there are any objects that result from that call, ObjectStore associates them with the session that the calling thread belongs to.

As a result of explicit and implicit association, a session provides a context for a set of persistent objects, and a set of ObjectStore API objects, such a **Database** objects and a **Transaction** object.

The session defines a namespace. The namespace defines unique names (and consequently identities) for databases, segments, transactions, and persistent objects. While it is possible for threads in different sessions to share objects, doing so is incorrect and usually results in exceptions.

If the thread in which an object was materialized leaves the session, the object remains associated with the session.

Rules for Automatically Joining a Thread to a Session

Because of the associations between objects and a particular session, some API calls imply a session. If there is no global session,

- A call that implies a session allows the calling thread to automatically be joined to the implied session.
- A call that does not imply a session does not allow the calling thread to automatically be joined to a nonglobal session.

If a thread associated with one session makes a call that implies some other session, ObjectStore throws ObjectStoreException.

Examples of Calls That Imply Sessions

A call that implies a session is a call that specifies an argument that is already associated with a session. It can also be a call in which the object on which the method is called is associated with a session. When these calls are in a thread that is not associated with a session, ObjectStore automatically joins the thread to the session with which the argument is already associated. It does not matter whether it is a global or nonglobal session. Some examples of API calls that imply a session follow:

- **Database.close()** — The **Database** argument was associated with a session when it was created.
- **ObjectStore.migrate(object, placement, export)** — The **placement** argument specifies a segment or database, which was associated with a session when it was initialized.
- **ObjectStore.destroy(object)** — The **object** argument designates a persistent object. It was associated with a session the first time it was accessed.

Examples of Calls That Do Not Imply Sessions

A call that does not imply a session is a call that does not specify an argument that is associated with a session. When these calls are in a thread that is not associated with a session, ObjectStore cannot automatically join the thread to a session, if it is a nonglobal session. Consequently, the call fails and ObjectStore throws ObjectStoreException. Some examples of API calls that do not imply a session follow:

- **Database.open(name, openType)** — This is a static method. The **Database** object does not exist yet so the **name** argument is not associated with a session.
- **Transaction.begin(type)** — This is another static method and the **Transaction** object does not exist yet.
- **ObjectStore.majorRelease()** — This is another static method.
- A call that accesses a transient object.
- A call that never accesses a persistent object, for example, **ObjectStore.getAutoOpenType()** and **ObjectStore.setLazyWriteLocking()**.
- Calls to the **ObjectStore.initialize()** and **ObjectStore.shutdown()** methods.

Explicitly Associating Threads with a Session

To explicitly join a thread to a session, you can call the following methods:

- **Session.join()**
- **ObjectStore.initialize(targetThread)**

Session.join()

To explicitly associate a thread with a session, call the **Session.join()** method. The method signature is

```
public void join()
```

This associates the current thread (the thread that contains the call to **join()**) with the session on which the **join()** method is called.

If the session has been terminated, or if the thread making the call is already associated with that session or some other session, ObjectStore throws ObjectStoreException.

To join a thread to a session for a bounded duration of time, try something like this:

```
Session session;
session.create(null, myproperties);
try {
    session.join();
    ...;
    ...;
    ...;
}   finally {
        session.leave();
}
```

ObjectStore.initialize()

In previous releases of ObjectStore, the only way to join a thread to a session was to call the **ObjectStore.initialize(targetThread)** method. In this release, this method is provided for compatibility with earlier releases. This method might be deprecated in a future release. The method signature is

public static boolean initialize(Thread targetThread)

If the specified target thread is already joined to a session, this method joins the current thread to that session and returns true. If the current thread is already joined to the target thread's session, this method returns false.

If the target thread is null, ObjectStore throws IllegalArgumentException. If the target thread is not associated with a session, ObjectStore throws ObjectStoreException, whether or not the session that the target thread was previously associated with is still active.

It does not matter whether or not the target thread has a transaction in progress. You can call **ObjectStore.initialize(targetThread)** at any time.

Working with Threads

After you associate a thread with a session, it is important to understand how to use the thread within the framework of a session. To that end, this section discusses

Cooperating Threads

All threads associated with a particular session cooperate with each other. That is, they

- Share transactions, persistent objects, and locks on ObjectStore data
- View the same state of any databases they access

For example, suppose thread **A** and thread **B** are cooperating threads (that is, they belong to the same session). **A** and **B** are running asynchronously. Each thread is issuing a sequence of operations and these sequences are interleaved in an unpredictable fashion.

For ObjectStore, it is as if these operations are all coming from the same thread. It does not matter which operation comes from **A** and which operation comes from **B**. ObjectStore views the operations as being in a single sequence, because they are issued from cooperating threads.

If **A** or **B** starts a transaction, it does not matter which thread issues the call. The transaction begins for both threads regardless of which thread actually starts the transaction. Any changes performed by **A** or **B** during the transaction are visible to both threads and can be acted on by either thread. Similarly, if **A** commits the transaction, it is just as if **B** commits the transaction.

So **B** must be in a state where it is okay to commit the transaction. **A** and **B** must cooperate.

Noncooperating Threads

Threads that do not belong to the same session cannot share transactions, persistent objects, or locks on data, and cannot view the same state of the database. Threads that belong to different sessions are noncooperating threads. With ObjectStore, a different session belongs to a different process. With PSE Pro, a different session can belong to the same or a different process.

Two or more noncooperating threads can open the same database at the same time and access the same root object. If two or more noncooperating threads access the same object in the database, there are an equivalent number of distinct instances of the persistent object — one for each thread. The identity test, **==**, does not show them to be identical.

Noncooperating threads can experience deadlock. See page 123.

Synchronizing Threads

Your application is responsible for synchronizing activity among cooperating threads when the transaction is committed or aborted. In general, your application must avoid accessing the database while a thread is committing the transaction and until a cooperating thread starts a new transaction. If a transaction is aborted, cooperating threads might need to retry database operations.

Additional information about synchronizing threads is in Multiple Cooperating Threads on page 126.

Removing Threads from Sessions

A thread can leave a session at any time, including while a transaction is in progress. This does not affect the transaction, nor any threads that are still joined to that session. With or without a transaction in progress, it is okay if there are no threads associated with a session. The session does not terminate. A thread can join a session later to finish the transaction. If no thread ever does that, ObjectStore aborts the transaction when the session terminates.

To end the association of a thread with a session, call the **Session.leave()** method. The method signature is

public static void leave()

After you execute this method, the current thread is no longer joined to the session. If the current thread is already not associated with the session on which the method is called, ObjectStore throws ObjectStoreException.

If your application or ObjectStore shuts down a session, ObjectStore causes any associated threads to leave the session before it performs the shut down.

You can also call **ObjectStore.shutdown()** to remove a thread from a session. However, this method also shuts down the session when it is called on the last thread in the session.

If a thread is associated with a session and the thread terminates, it automatically leaves the session.

Threads That Create a Session

There is nothing special about the thread that creates a session. This thread can leave the session and any threads associated with that session can continue operating.

When your application calls the **Session.createGlobal()** or **Session.create()** method, ObjectStore does not associate the thread that calls the method with the newly created session. For that thread to join the new session, it must call the **Session.join()** method.

Other Threads

A thread that does not belong to a session cannot use the ObjectStore API. This rule has a few exceptions. A thread must not be associated with a session when it calls

- **Session.createGlobal()**
- **Session.join()**
- **ObjectStore.initialize()**

A thread need not be associated with a session to successfully call **Session.isActive()**.

If a session has a transaction in progress, a thread that is not associated with that session must not use persistent objects that belong to that session. See Which Threads Can Access Which Persistent Objects? on page 53.

If a session does not have a transaction in progress, any thread, including threads that do not belong to that session, can access persistent objects to the degree they were left visible when the application committed or aborted the transaction. See Ending a Transaction on page 115.

Determining If ObjectStore Is Initialized for the Current Thread

You can use the **Session.getCurrent()** method to determine whether or not ObjectStore is initialized for the current thread. The method signature is

public static Session getCurrent()

This method returns the session with which the current thread is associated. If the current thread is not associated with a session, this method returns null.

Which Threads Can Access Which Persistent Objects?

Each persistent object is associated with exactly one session. Any modification to the state of a persistent object must be done by a thread that cooperates in the session to which the persistent object belongs.

After you terminate a session, the persistent objects and API objects that were associated with it when it was terminated continue to be associated with the terminated session. One exception to this is when you call the **Database.close()** method with a **true** argument. This causes the persistent objects to be retained as transient objects, which are not associated with any session.

The information in this section is provided to help you ensure that threads access the correct objects. This section discusses these topics:

Multiple Representations of the Same Object

When you have multiple sessions, it is possible to have multiple persistent objects that represent the same object in the database. For example, a thread belonging to session **A** accesses object **X**. Then a thread belonging to session **B** accesses object **X**. There are two persistent objects that represent **X**. Each one is a representation of the same object in the database. If you use the **==** operator on session **A**'s **X** and session **B**'s **X**, the result is that they are not identical; they are not the same object. Within a session, ObjectStore preserves object identity.

Example of Multiple Sessions

The following example shows some actions you can and cannot perform when you have multiple sessions. In this example, suppose you have

- **sessionA** and **sessionB**
- **threadA** is associated with **sessionA**
- **threadB** is associated with **sessionB**

In **threadA**, you start a transaction and read the contents of a persistent object called **objectA**. Since **threadA** is associated with **sessionA**, **objectA** belongs to **sessionA**. You commit the transaction with **ObjectStore.RETAIN_UPDATE**.

At this point, in **threadB** you can read or modify **objectA** as long as there is no transaction in progress in **sessionB**. However, any modifications will be discarded when **sessionA** starts a transaction.

Application Responsibility

It is the responsibility of the application to ensure that noncooperating threads act on persistent objects only in the ways allowed when a transaction is not in progress.

If you have a Java static variable that contains a persistent object and there are two separate sessions, you must decide which session owns the static variable. In other words, if there is a Java static variable whose value is a persistent object, that persistent object is associated with one session.

Effects of Committing a Transaction

When a thread commits a transaction, it affects only those persistent objects that belong to the same session that the thread belongs to.

Caution

You must ensure that an object never refers to an object that belongs to a different session. This is crucial because transitive persistence (performed when committing a transaction) must never reach an object that belongs to another session. If it does, ObjectStore throws ObjectStoreException.

Array objects

When a thread commits a transaction, if ObjectStore reaches an object whose class does not implement **IPersistent**, ObjectStore treats the object as a transient object and migrates it to a database. This works correctly for immutable classes such as **Integer** and **String**. For array objects, this can cause unpredictable results, because one session might modify the object while another session is using the old contents.

API Objects and Sessions

Each ObjectStore API object is related to one session. These metaobjects are

- **Database**
- **DatabaseRootEnumeration**
- **DatabaseSegmentEnumeration**
- **Segment**
- **SegmentObjectEnumeration**
- **Transaction**

If you open the same database from two noncooperating transactions, each session has its own **Database** object to represent the database. These Database objects are not identical, that is, **==** returns false.

If you try to use a database in the wrong session, ObjectStore throws ObjectStoreException.

Description of Concurrency Rules

ObjectStore allows multiple readers or *one writer* of an object at any given time. The term one writer implies one session in any process. In a session, two cooperating threads that are both updating an object count as one writer. Two threads from different processes do not cooperate and, therefore, count as two writers.

If you use Multiversion Concurrency Control (MVCC), there can be one writer *and* multiple readers. See Chapter 10, Controlling Concurrency, on page 317.

See also Handling Deadlocks on page 123.

Granularity of Concurrency

ObjectStore locks data at the page level. ObjectStore acquires a read lock on the object the first time the Java application reads or writes the contents of the object in a transaction. When that happens, the underlying C++ ObjectStore acquires a read lock on all pages used to store the object, as well as additional locks on other internal data structures (the info segment and schema segment). There is also Java-specific metadata that gets locked.

On Solaris and Windows, the size of a page is 4 KB.

Converting Read Locks to Write Locks

When a Java application modifies an object for which it already has a read lock, ObjectStore does not necessarily convert the read lock to a write lock immediately. The ObjectStore **setLazyWriteLocking()** method controls this behavior. If lazy write locking is true (the default) then ObjectStore only acquires write locks when it attempts to write the modified contents of the object to the database. That occurs either at commit time or when and if the application calls **ObjectStore.evict()** on the modified object.

Description of ObjectStore Properties

When you create a session, you can specify a properties argument. This section provides the following information about this argument:

About Property Lists Relevant to ObjectStore

When you create a session, there are two relevant property lists.

- The **java.util.Properties** object that is the second argument to the method that creates the session
- The system property list

Finding the value of a property

To find the value of a property, ObjectStore checks the **java.util.Properties** object. If it provides a value, ObjectStore uses it. If it does not provide a value, ObjectStore checks the system property list.

Passing a property value

When you want to pass a property value to the method that creates a session, you typically put it in the **java.util.Properties** object that is an argument to the method that creates the session.

There is only one system property list for each Java VM. If there are multiple sessions in the same Java VM, they all use the same system property list. For more information about system properties, see **System.getProperty**, in section 20.17.9 of the *Java Language Specification*.

Defining a system property

All ObjectStore property names start with **COM.odi**. You can pass in property information by defining it as a system property. For example:

```
Properties props = System.getProperties();
props.put("COM.odi.useDatabaseLocking", "true");
Session session = Session.create(null,props);
```

There is also a **System.setProperties()** method that resets the **System** property list.

The JDK allows you to specify a system property by including

-D*parameter***=***value*

on the **java** command line before the class name. Each such specification defines one system property. Not all Java virtual machines run this way.

Defining a **Properties** object

If you want to construct your own property list, the type of the property list argument is **java.util.Properties**. For example:

```
Properties props = new Properties();
props.put("COM.odi.useDatabaseLocking", "true");
Session session = Session.create(null,props);
```

Description of COM.odi.applicationName

Set the **COM.odi.applicationName** property to the name of the application for the current client. This allows users of the **LockTimeoutBlocker** class and the **ossvrstat** utility to retrieve information about clients involved in concurrency conflicts. When you set this property, it can provide information about your application to other clients.

Description of COM.odi.cacheSize

The **COM.odi.cacheSize** property specifies the size of the C++ client cache in bytes. The default is 8 MB. If the value is a **String** that starts with 0x or 0X, ObjectStore treats the value as a hexadecimal number. ObjectStore rounds the cache size down to the nearest whole number of pages.

For information about how to determine the optimum cache size, see the book *ObjectStore Management*, Chapter 3, **OS_CACHE_SIZE** environment variable.

Description of COM.odi.disableWeakReferences

The **COM.odi.disableWeakReferences** property defaults to **"false"**. This means that ObjectStore uses the weak reference facility of the JDK. If you set this property to **"true"**, it disables the weak reference facility and ObjectStore does not use it.

When you start the first session in a Java process, the setting of the **COM.odi.disableWeakReferences** property is in effect for the duration of the Java process. If you terminate the session and start another session with a different value for the **COM.odi.disableWeakReferences** property, the new value is ignored.

A weak reference to an object is a reference that does not prevent the object from being garbage collected by the Java VM's garbage collector. ObjectStore uses weak references in its internal object table to hold references to unmodified persistent objects. If your program does not have any references to a persistent object and the reference in the object table is the only reference, the object can be garbage collected. If the persistent object has been modified and the changes have not yet been saved, ObjectStore uses strong references. Strong references do not allow the object to be garbage collected.

The weak reference facility in the JDK 1.1 is implemented in such a way that it prevents unmodified persistent objects from being garbage collected. This is corrected in the JDK 1.2. To use your database with the JDK 1.2, you must upgrade it. See Upgrading Databases for Use with the JDK 1.2 on page 106.

Description of COM.odi.migrateUnexportedStrings

The **COM.odi.migrateUnexportedStrings** property controls what happens when ObjectStore encounters a cross-segment reference to an unexported **String** object. If this property is not set or if it is set to true, ObjectStore creates a new **String** object that has the same value as the referenced object. ObjectStore places this new string in the same segment as the referring object and substitutes this new string for the referenced string. If this property is set to false, ObjectStore throws ObjectNotExportedException if it encounters a reference to a string in another segment and that string is not exported.

Description of COM.odi.ObjectStoreLibrary

COM.odi.ObjectStoreLibrary specifies the name of the native C++ library that contains the ObjectStore schema and native methods for the application.

If you use Java/C++ interoperability, you must specify this property. If you do not use any C++ libraries in your application, you do not need to specify this property. The standard library supports **COM.odi.coll** collections, which are Java peer objects. A custom library is needed for application-specific peer classes.

You must specify a name that is acceptable to **System.Load.Library()** and not an explicit path. The name should follow platform conventions for library names. If you do not specify this property, ObjectStore uses the standard library, which provides for primary Java objects but not for Java peer objects. If your ObjectStore Java application is accessing C++ classes, you want the library that provides for Java peer objects.

For additional information, see *Developing ObjectStore Java Applications That Access C++*, Chapter 4, Building the Application.

Description of COM.odi.password and COM.odi.user

COM.odi.user and **COM.odi.password** allow you to supply a user name and a password when the ObjectStore Server has **Name Password** set for the **Authentication Required** Server parameter.

Description of COM.odi.product

COM.odi.product allows you to run multiple simultaneous sessions against different Object Design Java products in the same Java VM. Each Object Design Java product runs in its own session.

You must separately obtain each Object Design Java product that you want to use. When you purchase ObjectStore, neither PSE nor PSE Pro is included. To use PSE or PSE Pro, you must have a copy of PSE or PSE Pro.

Your **CLASSPATH** environment variable must include an entry for each Object Design Java product that you want to use. For example, if you want to use ObjectStore and PSE Pro, you must have entries for both **osji.zip** and **pro.zip** in your **CLASSPATH**. The order of the entries does not matter.

When you use multiple Object Design Java products, they must be compatible with each other. If they are not, FatalApplicationException is thrown.

You can set the **COM.odi.product** property to one of the following three values (case is not significant):

- **PSE**
- **PSEPro**
- **ObjectStore**

The **COM.odi.product** property applies to one session; it is not global. In other words, each session has a product attribute. After you start a session, you cannot change the value of **COM.odi.product** for that session.

With the **COM.odi.product** property, a single process can run all of the following at the same time:

- One PSE session
- Multiple PSE Pro sessions
- One ObjectStore session

You can create a session without explicitly setting the **COM.odi.product** property. In this case, Object Design software checks the Java system property list. If it finds a value for **COM.odi.product**, it uses that value. If it does not find a value, the default is that the software looks for PSE Pro, then PSE, then ObjectStore, and uses the first product it finds.

There are many ways this feature can be useful. For example, an application can:

- Open a PSE database in one session and an ObjectStore database in another session and use both.
- Copy data from a database created with one product to a database created with another product.

The use of **COM.odi.product** to copy data among databases created with different products requires several steps. For example, to copy data from a PSE database to an ObjectStore database, you must do the following in the PSE session:

1 Open the source database.

2 Read the objects you want to copy.

The **ObjectStore.deepFetch()** method is useful for doing this.

3 Commit the transaction with **ObjectStore.RETAIN_READONY**.

4 Close the source database and specify true to retain the persistent objects as transient objects.

If you do not close the database, the persistent objects remain associated with the session in which you read them. This prevents another session from storing them in another database.

Then, in the ObjectStore session, you must

5 Make the objects reachable from the destination database.

6 Commit the transaction.

ObjectStore uses transitive persistence to store all reachable objects in the destination database.

Here is an example of code that performs these steps.

```
import COM.odi.*;
import COM.odi.util.*;
import java.util.Properties;

class CopyToOSJI implements ObjectStoreConstants {
   public static void main(String[] args) {

   /* Create some data in a PSE database and then read it out. */

      Properties properties = new Properties();
       properties.put("COM.odi.product", "PSE");

      Session.create(null, properties).join();
      Database database = Database.create(
         "pse.odb", ALL_READ | ALL_WRITE);

      Transaction.begin(UPDATE);
      OSVector vector = new OSVector();
      vector.addElement(new Integer(3));
      vector.addElement(new Integer(4));
      database.createRoot("vector", vector);
      Transaction.current().commit();

      Transaction.begin(READONLY);
      vector = (OSVector)database.getRoot("vector");
      ObjectStore.deepFetch(vector);
      Transaction.current().commit(RETAIN_READONLY);

   /* Close the database and specify true
      to retain persistent objects as transient objects. */

      database.close(true);
      Session.leave();

   /* Copy the data to an ObjectStore database. */

      properties.put("COM.odi.product", "ObjectStore");
      Session.create(null, properties).join();
      database = Database.create("osji.odb",
         ALL_READ | ALL_WRITE);

      Transaction.begin(UPDATE);
      database.createRoot("vector", vector);
      Transaction.current().commit();
      database.close();
   }
}
```

Description of COM.odi.stringPoolSize

The **COM.odi.stringPoolSize** property allows you to specify how many newly created strings ObjectStore maintains in the string pool for the current session. The default is **"100"**.

When ObjectStore is about to migrate a string to the database, it first checks the string pool for an identical string. If it finds one, it uses the string that is already stored in the database instead of adding a new identical string to the database. The information about which strings are available to be shared is maintained only for the current transaction. The strings that are available to be shared are maintained in a string pool. ObjectStore resets the string pool to empty at the start of each transaction.

For example, suppose you create two instances of a **Person** object in a transaction. In each instance, the value of the **name** field is **Lee**. If you store both instances in the database in the same transaction, ObjectStore adds only one instance of the string **"Lee"** to the database. This is true even though the Java VM might contain two instances of the string **"Lee"**. When ObjectStore writes the first **"Lee"** string in the database, it notes it in the string pool. Before ObjectStore stores the next instance of **"Lee"** in the database, it checks the string pool to see if an identical instance is already in the database.

Continuing the example, suppose you use two transactions and you store one instance of **Person** in each transaction. The result is that there are two identical **"Lee"** strings in the database. This is because ObjectStore resets the string pool to be empty at the start of each transaction. Consequently, ObjectStore cannot reuse the **"Lee"** string from the previous transaction.

Caution

If you use **ObjectStore.destroy()** to destroy strings explicitly, you might want to turn off string pooling, so that you do not inadvertently destroy a string that is shared by different objects. Alternatively, you can use the persistent GC to reclaim strings when they are no longer referenced. Using the GC is usually preferable to explicitly calling **destroy()**, because it is safer to let the persistent GC collect unreachable strings. Also, this approach is often more efficient and results in less database fragmentation.

Description of COM.odi.trapUnregisteredType

The **COM.odi.trapUnregisteredType** property is useful for troubleshooting ClassCastExceptions. The default is that this property is not set, and it is usually best to use the default.

When ObjectStore encounters an object of a type for which it does not have information (that is, the type is unregistered), it checks the setting of the **COM.odi.trapUnregisteredType** property.

If the property is not set, ObjectStore creates an instance of the **UnregisteredType** class to represent the object of the unknown type. Your application continues to run as long as it does not try to use the **UnregisteredType** object. Often, this can be fine because your application has no need for that particular field. However, if you do try to use the object of the unregistered type, ObjectStore throws ClassCastException.

If **COM.odi.trapUnregisteredType** is set, ObjectStore does not create an **UnregisteredType** object. Instead, it throws FatalApplicationException and provides a message that indicates the name of the unregistered class. For additional information, see Handling Unregistered Types on page 184.

Chapter 4
Managing Databases

You create databases to store your objects. The **Database** class provides the API for creating and managing databases.

The Java interface to ObjectStore supports rawfs databases. For information about rawfs databases, see the book *ObjectStore Management*.

Contents

This chapter discusses the following topics:

Creating a Database

The **Database** class is an abstract class that represents a database. When you create a database,

- There must be an active session or ObjectStore throws ObjectStoreException.
- A transaction must not be in progress, or ObjectStore throws TransactionInProgressException.

Databases are cross-platform compatible. You can create databases on any supported platform and access them from any supported platform.

This section discusses the following topics:

Method Signature for Creating a Database

To create a database, call the static create method on the **Database** class and specify the database name and an access mode. The method signature is

public static Database create(String name, int fileMode)

ObjectStore throws AccessViolationException if the access mode does not provide owner write access.

Example of Creating a Database

For example:

```
import COM.odi.*;
class DbTest {
    void test() {
        Database db = Database.create("objectsrus.odb",
            ObjectStore.OWNER_WRITE);
        ...
    }
}
```

This example creates an instance of **Database** and stores a reference to the instance in the variable named **db**. The **Database.create** method is called with two parameters.

The first parameter specifies the pathname of a file. The path can specify a relative name or a fully qualified name. It must always specify a file that is one of the following:

- Local.
- In a mounted directory.
- In an unmounted remote directory—in this case the file must be identified by a pathname that specifies a remote host.

An ObjectStore Server must be available for the directory that contains the specified file.

The second parameter specifies the access mode for the database.

Terminology note

Database is an abstract class, so ObjectStore actually creates an instance of a subclass that extends **Database**. From your point of view, it does not matter whether ObjectStore creates an instance of **Database** or an instance of a **Database** subclass.

Result of Creating a Database

The result is a database named **"objectsrus.odb"** with an access mode that allows the owner to modify the database. The example stores the reference to the **Database** object in the **db** variable. This means that **db** represents, or is a handle for, the **objectsrus.odb** database.

For each database you create, ObjectStore creates an instance of **Database** to represent your database. Each database is associated with one instance of **Database**. Consequently, you can use the **==** operator to determine whether or not two **Database** objects in the same session represent the same database, for example, the following method returns **true**:

```
boolean checkIdentity(String dbname) {
    Database db = Database.create(dbname,
        ObjectStore.OWNER_WRITE);
    Database dbAgain = Database.open(dbname,
        ObjectStore.UPDATE);
    return (db == dbAgain);
}
```

Specifying a Database Name in Creation Method

When you create or open a database, you can specify or pass a database name that is a relative name, an absolute operating system pathname, or a rawfs pathname. ObjectStore takes into account local network mount points when interpreting pathnames. A pathname can refer to a database on a remote host. However, an ObjectStore Server must be available to the local host of the directory that contains an ObjectStore database.

If you want to refer to a database on a remote host for which there is no local mount point, you can use a Server host prefix. This is the name of the remote host followed by a colon (:), as in **oak:/foo/bar.odb** or **jackhammer:c:\bob\mydb.odb**. On Windows, you can also use UNC pathnames, as in **\\oak\c\foo\bar.odb**.

Also, you can use locator files to allow access to additional hosts. See *ObjectStore Management* for information.

When the Database Already Exists

If you try to create a database that already exists, ObjectStore throws DatabaseAlreadyExistsException. Before you create a database, you might want to check to see if it exists and destroy it if it does. For example, you can insert the following just before you create a database:

```
try {
    Database.open(dbName, ObjectStore.UPDATE).destroy();
}   catch(DatabaseNotFoundException e) {
}
```

Warning

Do this only if you want to destroy and recreate your database. Otherwise, invoke **Database.open()**.

Discussion of Installing Schema upon Database Creation

The **Database.create()** method has an overloading that allows you to install the schema in batch mode rather than incrementally. The default is that ObjectStore performs schema installation as needed. The advantage of batch schema installation is that concurrency conflicts due to schema installation are minimized. The disadvantage is that database creation takes a little longer and the initial database size is larger. See Installing Schema Information in Batch Mode on page 333.

Creating Segments

ObjectStore creates each database with one segment that you can use to store objects. A segment is a variable-sized region of disk space that ObjectStore uses to cluster objects stored in the database. Initially, the size of the segment is about 3 K. As you store additional objects in the segment, ObjectStore increases the size of the segment automatically.

To create additional segments, use the **Database.createSegment()** method. To locate a segment by its ID, use the **Database.getSegment()** method. To retrieve the segment that contains a particular object, call the **Segment.of()** method on that object.

The initial segment in a database is the default segment. To change the default segment, invoke the **Database.setDefaultSegment()** method.

To lock all objects in a segment, see Locking Objects, Segments, and Databases to Ensure Access on page 328.

For additional information about segments, see Managing Databases in Chapter 1 of *ObjectStore Management*.

Storing Objects in a Particular Segment

You can store objects in different segments by changing which segment is the default segment or by explicitly storing an object in a particular segment. Distributing objects can improve performance, but the following tradeoffs must be considered:

- If two objects are in the same segment, the time required to read or update both decreases.
- If two objects are in different segments, the risk of lock contention when you access only one of them is eliminated.

To make a persistent reference from an object in one segment to an object in another segment, the object in the other segment (the referenced object) must be exported. See Implementing Cross-Segment References for Optimum Performance on page 99 for additional information about distributing objects among segments.

Determining If a Database or Segment Is Transient

Sometimes there are Java peer objects that identify C++ objects that have been transiently allocated. ObjectStore stores these C++ objects in the transient database and transient segment. Java primary objects are never in the transient database or transient segment, even if they are transient. ObjectStore creates transient segments as part of some peer object operations. You cannot use ObjectStore to create or manipulate transient segments.

If you try to retrieve the segment or database of a transient primary object, ObjectStore throws ObjectNotPersistentException. To determine whether or not a database or segment is transient, you can do the following:

```
CPlusPlus.getTransientSegment() == segment
CPlusPlus.getTransientDatabase() == database
```

Iterating Through the Segments in a Database

To obtain an enumeration of the segments in a database, call the **Database.getSegments()** method. The signature is

```
public DatabaseSegmentEnumeration getSegments()
```

This method returns a **DatabaseSegmentEnumeration** object. After you have this object, you can use these methods to iterate over the segments in the enumeration:

- **DatabaseSegmentEnumeration.nextElement()**
- **DatabaseSegmentEnumeration.nextSegment()**
- **DatabaseSegmentEnumeration.hasMoreElements()**

If you or another session add a segment to a database after you create an enumeration, the enumeration might or might not include the new segment. If it is important for the enumeration to accurately list all segments, you should recreate the enumeration after you create the segment.

After you create an enumeration, a segment in the enumeration might be destroyed. If you use the enumeration to try to access a destroyed segment, ObjectStore skips the destroyed segment. However, if you retrieve a segment with **DatabaseSegmentEnumeration.nextElement()** or **nextSegment()** and then the segment is destroyed, ObjectStore throws SegmentNotFoundException if you try to use the destroyed segment.

Opening and Closing a Database

A database can be either open or closed. A database must be open before you can store or access objects in that database.When an application opens a database, it does not matter whether

- A transaction is in progress
- The database is already open

When an application closes a database, a transaction cannot be in progress and the database must be open.

This section discusses the following topics:

Opening a Database

When you open a database, it does not matter whether there is a transaction in progress, nor does it matter whether the database is already open.

When you create a database, ObjectStore creates and opens the database. To open an existing database, call the static **Database.open()** method. The method signature is

```
public static Database open(String name, int openMode)
```

For example:

```
Database db = Database.open("myDb.odb",
    ObjectStore.READONLY);
```

The first parameter specifies the pathname of your database. The second parameter indicates the open mode of the database.

Possible Open Modes

ObjectStore provides constants that you can specify for the **openMode** parameter to **Database.open()**. The constants you can specify for **openMode** are

- **ObjectStore.UPDATE** to read and modify a database
- **ObjectStore.READONLY** to read but not modify a database
- **ObjectStore.MVCC** allows you to open the database for MVCC (multiversion concurrency control). This allows you to read the database, but it does not block another session from updating the database.

Incorrect attempts to modify

If you open a database with **ObjectStore.READONLY** or **ObjectStore.MVCC**, and attempt to modify an object, ObjectStore throws UpdateReadOnlyException when you try to commit the transaction.

Example

Suppose you previously created and closed a database that is represented by an instance of a **Database** subclass stored in the **db** variable. You can call the instance **open()** method to open your database this way:

```
db.open(ObjectStore.READONLY);
```

You can use the static class **open()** method this way:

```
db = Database.open("myDb.odb", ObjectStore.READONLY);
```

Typically, both lines cause the same result. However, they might cause different results if a database has been destroyed and recreated.

To recover the database, open it for update. This automatically recovers the database if necessary. Alternatively, you can run the **osjcheckdb** utility with the **-openUpdateForRecovery** option.

To lock all objects in a database, see Locking Objects, Segments, and Databases to Ensure Access on page 328.

Opening the Same Database Multiple Times

Each subsequent opening of a database after the initial open operation returns the same database object. For example:

```
db1 = Database.open("foo", ObjectStore.UPDATE);
db2 = Database.open("foo", ObjectStore.UPDATE);
```

The expression **db1 == db2** returns true. They refer to the same database object. Consequently, a call to **db1.close()** or **db2.close()** closes the same database. No matter how many times you open a database, a single call to the **close()** method closes the database. (This is different in the C++ interface to ObjectStore. In that interface, for example, if you call **open()** four times and **close()** three times all on the same database, the database is still open.)

Closing a Database

To close a database, call the **close()** method on the instance of the **Database** subclass that represents the database, for example,

```
db.close();
```

You cannot close a database when a transaction is in progress. The database you want to close must be open.

Object state after close

When you close a database, all persistent objects that belong to that database become stale or transient. If the last committed transaction that operated on the database retained persistent objects, you can use an overloading of **close()** that allows you to specify what should happen to the retained objects. (For information about retained objects, see Committing Transactions to Save Modifications on page 153.) The method signature is

```
public void close(boolean retainAsTransient)
```

Specify true to make retained objects transient. If you specify false, it is the same as calling the **close()** method without an argument. All access to retained objects ends.

Suppose you close a database and make retained objects transient. In the next transaction, if you reread an object from the database that you retained as a transient object, you then have two separate copies of the same object. One object is transient and one object is persistent. You do not have two references to a single object. When you close a database, all object identity is gone. After you close a database, the database is still associated with the session in which it was closed.

If you do not close

If you do not close a database, ObjectStore closes it when you shut down ObjectStore.

Database identity

Within a session, ObjectStore maintains database identity even after you close a database. For example, consider the following code:

```
import COM.odi.*;

public class Goo {
    public static void main(String[] args) {
        Session session = Session.create(null, null);
        session.join();
        try {
            try {
```

```
            Database db = Database.create("my.odb", 0664);
            db.close();
          } catch (DatabaseAlreadyExistsException e) {
          }
          Database db1 = Database.open("my.odb",
            ObjectStore.READONLY);
          db1.close();
          Database db2 = Database.open("my.odb",
            ObjectStore.READONLY);
          System.out.println(db1 == db2);
        } finally {
          session.terminate();
        }
    }
}
```

If you run a program with the previous code, the system displays **true**.

In general, it is best to leave databases open for the entire session and write your application so that it shuts down ObjectStore before it exits.

However, keeping databases open consumes some ObjectStore Server resources. Also, there is a limit to the number of databases the ObjectStore Server can keep open at one time. This depends on the number of open files that the operating system permits, which varies by platform.

Automatic Opens of a Database

Sometimes an application traverses a reference to a database that has not been explicitly opened. ObjectStore automatically opens the database according to the default open mode. The default open mode is one of the following:

- **ObjectStore.READONLY**
- **ObjectStore.UPDATE**
- **ObjectStore.MVCC**

An application can call the **ObjectStore.setAutoOpenMode()** method to change the default open mode. A call to the **ObjectStore.getAutoOpenMode()** method returns the current open mode. The default autoopen mode is **READONLY**. When you set the autoopen mode, it affects only the current session.

If you know the name of a database that has been automatically opened, you can use **Database.open()** to obtain the already open database. Another way to obtain a handle to an automatically opened database is to call **Database.of()** on an object from the automatically opened database.

If you do not close a database that ObjectStore automatically opens, ObjectStore closes it when you shut down ObjectStore.

Disabling autoopen

You can disable the ability of ObjectStore to automatically open databases. Call the **ObjectStore.setAutoOpenMode()** method and specify the **ObjectStore.DISABLE_AUTO_OPEN** constant. If you do disable automatic opens and your application tries to follow a reference to an unopened database, you receive **DatabaseNotOpenException**.

Objects in Closed Databases

Objects in a closed database are not accessible. However, if you close a database with an argument of true, ObjectStore retains the persistent objects as transient objects.

Moving or Copying a Database

You can use the ObjectStore utilities **oscopy** and **osmv** to copy and move a database. See *ObjectStore Management* for information.

You can move or copy databases among different supported platforms.

Performing Garbage Collection in a Database

The ObjectStore persistent garbage collector (GC) collects unreferenced Java objects and ObjectStore collections in an ObjectStore database. Persistent garbage collection frees storage associated with objects that are unreachable. It does not move remaining objects to make the free space contiguous.

Contents

This section discusses these topics:

Background About the Persistent Garbage Collector

The ObjectStore persistent GC is independent of the Java VM GC. The Java VM GC is strictly a transient object garbage collector. It never operates on objects in the database.

Applications can continue to use a database while persistent GC is in progress. GC locks portions of a segment as needed, as if it were just another application. In this way, the GC minimizes the number of pages that are locked and the duration for which the locks are held. Also, the GC retries operations when it detects lock conflicts.

By default, the GC runs with a transaction priority of zero. Consequently, it is the preferred victim when the Server must terminate a transaction to resolve a deadlock. At a later time, the GC redoes the work that was lost when the transaction was aborted.

The GC uses read and write lock timeouts of short duration. This avoids competition with other processes for locks. If the GC cannot acquire a lock because of a timeout, it retries the operation at a later time.

The GC performs its job in two major phases. In the mark phase, the GC identifies the unreachable objects. In the sweep phase, the GC frees the storage used by the unreachable objects.

A segment is the smallest storage unit that can be garbage collected. You can specify a segment or a database to be garbage collected. It is usually best to avoid destroying strings (or objects) altogether and let the persistent garbage collector take care of destroying such unreachable objects. The persistent garbage collector can typically destroy and reclaim such objects very efficiently, since it can batch such operations and cluster them effectively. If you set up the GC to run when the system is lightly loaded, you can effectively defer the overhead of the destroy operations to a time when your system would otherwise be idle, thus getting greater real throughput from your application when you really need it.

The persistent GC never removes tombstones for exported objects. For unexported objects, the GC treats tombstones the same way that it treats other objects. The GC removes tombstones if they are not referenced. In other words, the GC removes only unreferenced tombstones. This behavior preserves the safe detection of bad references.

API for Collecting Garbage in a Database

To perform garbage collection on a database, call the **Database.GC()** method. This method invokes the **Segment.GC()** method on each the segment in the database. The method signature is

public java.util.Properties GC(java.util.Properties GCproperties)

For the **GCproperties** parameter, specify null or a **Properties** object for the garbage collection operation. The properties are described in the next section as they are the same for **Segment.GC().** If the **GCproperties** parameter is null, ObjectStore uses the default properties as defined in the documentation for **Segment.GC()**. The properties you can specify are the same as the properties for **Segment.GC()**.

API for Collecting Garbage in a Segment

To perform garbage collection on a segment, call the **Segment.GC()** method. The signature is

public java.util.Properties GC(java.util.Properties GCproperties)

A transaction must not be in progress for the current session. The database that contains the segment you want to garbage collect

must not be open for the current session. You cannot perform GC on the transient segment. If you try to, ObjectStore throws SegmentException.However, you must open it to create a **Database** object to represent it. After you close the database you want to garbage collect, you can call **Database.GC()** on the **Database** object that represents your closed database.

The **GCproperties** parameter specifies a list of GC properties or null. When you specify null, ObjectStore checks the system properties and uses the default properties, which are suitable for most operations. For more control over GC, you can specify one or more of the following properties. ObjectStore uses the default for any property you do not specify.

- **COM.odi.gc.retries** is an **int** that defaults to **10**. This indicates the number of times the GC tries to resume the sweep phase of garbage collection after it waits for a lock.
- **COM.odi.gc.retryInterval** is an **int** that defaults to **1000**. This value indicates the number of milliseconds the sweep operation waits between sweep attempts for a concurrency conflict to be resolved before it tries to resume the sweep.
- **COM.odi.gc.lockTimeOut** is an **int** that defaults to **1000**. This value indicates the number of milliseconds the sweep operation waits for a lock conflict to be resolved. If it is not resolved in the specified length of time, the GC aborts the current transaction and starts a new transaction. ObjectStore rounds this value up to the nearest second.
- **COM.odi.gc.transactionPriority** is an **int** that defaults to **0**. This is the transaction priority associated with transactions started by the GC. The Server uses this specification when it must determine which transaction must be the victim in a deadlock. This number is intentionally low so that the GC transaction is the deadlock victim of choice.
- **COM.odi.gc.displayGarbage** is an **int** that defaults to **0**. If it is not **0**, objects that are unreachable are not destroyed. Instead, they are displayed. The argument controls the level of detail in the display:

  0 No display

  1 Lists the total number of candidates for garbage collection

2 Reserved

3 Lists the location of each candidate for garbage collection

4 Lists the roots of the object graphs of the candidates for garbage collection

The **GC()** method returns a **Properties** object that contains information about the results of the garbage collection. The properties in this object include

- **COM.odi.gc.reclaimedObjects** is the number of objects that were collected by the GC operation.
- **COM.odi.gc.reachableObjects** is the number of objects that the GC found to be reachable.

Command Line Utility for Collecting Garbage

The command line utility for collecting garbage in a database is osgc. See page 378.

Running osgc on C++ Databases or Segments

You can run the **osgc** utility on C++ databases and segments, but you must observe the following restrictions:

- The database or segment must be self-contained. There cannot be any pointers in another database or segment that point to objects in the database or segment on which you are performing the GC. Similarly, there cannot be any dumped references that point to objects in the segment or database being garbage collected. (A dumped reference is encoded in a **char*** string that is obtained by invoking the **dump()** member function on any ObjectStore reference, for example, **os_reference.dump()**.)

  The **osgc** utility is not aware of such pointers and can erroneously reclaim such objects. If the database or segment contains objects that are the targets of such pointers, you can work around this by doing one of the following:

  - Associate such objects with database roots.
  - Make such objects the targets of protected references. (A protected reference uses **os_reference_protected**.)

The **osgc** utility recognizes as reachable and, so, never collects objects that are associated with database roots and objects that are the targets of protected references.

- The database or segment cannot contain classes that have union members. The **osgc** utility makes no provision for the discriminant functions needed to examine instances of classes that contain unions.
- The database or segment cannot contain instances of **os_reference_local** or **os_reference_protected_local**. Such references can be resolved only in the context of a database, and **osgc** does not make provisions for resolving these references.
- You must not simultaneously run multiple **osgc** processes against the same database or segment. You must synchronize **osgc** processes so that a new **osgc** process on a database or segment is initiated only after the previous **osgc** process on the database or segment has run to completion.
- The database on which you run the **osgc** utility must have been created with ObjectStore 5.0 or a subsequent release. You cannot run the **osgc** utility on a database that was upgraded from an earlier release. However, you can run the **osdump** utility to obtain the contents of a database from an earlier release and then run the **osload** utility to store the contents in a Release 5.0 or later database. See *ObjectStore Management* for information about running these utilities.
- Databases that contain schema information about template instantiations, including information about ObjectStore templated collection types, contain garbage that the osgc utility removes. This is safe and does not affect correct operation.
- You can run the **osgc** utility concurrently with other applications that modify the database.

Schema Evolution: Modifying Class Definitions of Objects in a Database

You can modify the class definitions for objects already stored in a database. This process is called schema evolution, since a database schema is a description of the classes whose instances are stored in a database.

There are primarily two ways to evolve schema:

- Use the **Database.evolveSchema()** API.
- Use serialization with a dump/load utility.

You can always use the schema evolution API, but you should use it when the data you must evolve contains very large object graphs. Also, you must use the API when a database contains instances of **COM.odi.coll** objects.

Use the serialization technique with the sample code provided only when the database you want to evolve fits into heap space. When you use the serialization technique, the database cannot contain ObjectStore collections because they are not serializable.

The topics discussed in this section are

- When is schema evolution required?
- Preparing to use the schema evolution API
- Using the schema evolution API
- Considerations for using serialization to evolve schema
- Steps for using sample code that uses serialization with a dump/load utility
- Sample code

When Is Schema Evolution Required?

If you change your class in the following way, you must evolve the schema:

- Add or remove a persistent instance field.
- Change the type of a persistent instance field (see additional information about indexable fields).
- Change the order of persistent instance fields.

ObjectStore initializes each new instance field with its default value as described in section 4.6.4 of the *Java Language Specification*. The new fields are not initialized with the variable initializer even if the class defines one for the new field.

If you change the type that is associated with a persistent instance field, ObjectStore performs a default initialization, except in cases where both the old and new instance fields are of the following primitive types: **byte**, **short**, **int**, **float**, or **double**. In this case, ObjectStore applies the appropriate narrowing or widening conversion to the old value and assigns that value to the new instance field. Conversions that involve the **long** type cause ObjectStore to perform a default initialization on the new field.

hashCode()

Also, you might need to perform schema evolution if you add or remove the **hashCode()** method. If you use the postprocessor, it determines whether or not to add a **hashCode()** method. If it previously added a **hashCode()** method and now it does not, or if it previously did not add a **hashCode()** method and now it does, schema evolution is required.

Inheritance

You cannot use schema evolution to change the inheritance hierarchy of a class by adding, removing, or changing a superclass.

Allowed changes

You can make the following changes to your class and you are *not* required to evolve the schema:

- Add or remove class or instance methods.
- Add or remove class fields.
- Add or remove transient instance fields.
- Add or remove an implementation of an interface.

Indexable fields

When you make a field indexable, the change might require schema evolution. If a class (including its superclasses) does not contain any indexable fields, schema evolution is required if you make a field in the class indexable. The addition of the indexable field changes the representation of the class.

If a class (including its superclasses) has at least one indexable field, schema evolution is not required if you add indexes to other fields.

Preparing to Use the Schema Evolution API

Before you can use the API to perform schema evolution on a database, you must create a **PersistentTypeSummary** instance that identifies the classes whose definitions have changed. The easiest way to do this is to specify the **-summary** option when you run the postprocessor on the updated class definitions. Be sure to run the postprocessor on all the classes in your application at once or on a correctly grouped batch of classes.

If the database contains collections that use indexes, you must drop the indexes before you perform schema evolution. After you evolve the schema, you can restore the indexes on the collection.

It is always advisable to make a copy of your database before you evolve its schema. This allows you to restore the database if there are errors.

To perform schema evolution, the following conditions must be true:

- The database must not be open for read-only. It can be closed or open-for-update.
- A transaction must not be in progress.
- There must be a **PersistentTypeSummary** instance that identifies the classes whose definitions have changed.

Using the Schema Evolution API

When ObjectStore performs schema evolution, it makes the database inaccessible to any other operation. To evolve the schema for a database, call the **Database.evolveSchema()** method. The signature is

void evolveSchema(String dbName,
String workdbName,
PersistentTypeSummary summary)

The **dbName** parameter specifies the database whose schema you want to evolve. During schema evolution, this database is not available to any other operation.

The **workdbName** parameter specifies the name of a database that ObjectStore uses to hold a checkpoint version of the database while schema evolution progresses. ObjectStore creates this database as part of schema evolution and destroys it when schema evolution is complete. This working database allows ObjectStore to resume schema evolution if it is interrupted. For example, an interruption can be caused by a power failure or a lack of disk space.

The **summary** parameter specifies a **PersistentTypeSummary** object that identifies the classes whose definitions have changed. It is permissible for the summary to include types that have not changed. However, if the summary does not include a type that has changed, that type might not be evolved.

Typically, you create the summary object as follows:

1 When you run the postprocessor on your updated class definitions, specify the **-summary** option.

This creates a class file.

2 Call the no-arguments constructor on this class to create the summary object.

Under unusual circumstances, you might create the **PersistentTypeSummary** object yourself. In this case, you must use its constructor for specifying the persistent classes and included summaries.

ObjectStore can evolve only one database at a time. This means that if there are cross-database references, only one database at a time is unavailable to other operations.

When you perform schema evolution on a particular class, you must provide definitions for all superclasses that are part of the schema for the database being evolved.

Considerations for Using Serialization to Perform Schema Evolution

To evolve a schema, you can

1 Use serialization to dump the contents of a database.

2 Modify your class definitions.

3 Reload the data.

The next two sections provide instructions for using sample code and the sample code that does this. If you use this sample code, you should be aware of the following issues.

java.io.Serializable

To serialize objects into the database, the classes of all the objects stored in the database must implement **java.io.Serializable.** If you have a database that contains objects that do not implement **Serializable**, you can modify the class definitions just to implement **Serializable**, recompile them, and still access the database. This allows you to dump the database to a file before you make the real class modifications.

readObject()

When you modify a class after doing the dump, you must ensure that the **readObject()** method considers the old and new versions of the class to be compatible. The most straightforward way to do this is to create a **static final long** field called **serialVersionUID** in the modified class. This field must have the same value as the serial version UID for the original class. You can obtain the value for the original class with the **serialver** utility, for example:

```
serialver DumpReload$LinkedList
DumpReload$LinkedList: static final long serialVersionUID =
-5286408624542393371L;
```

For simplicity, the sample code includes this field.

Database size

The database whose schema you want to evolve must be small enough to fit into heap space. If it is not, you must customize the code that dumps and loads the database. You would have to organize your data so that you do not have to serialize all the data in the database at one time.

Large numbers of connected objects

The use of **ObjectStore.deepFetch()** is a performance concern for very large object graphs. The current implementation of **deepFetch()** is not careful about bounding stack space. A consequence of this is that it is sometimes impossible to successfully perform the **deepFetch()** operation for very large object graphs.

Steps for Using Sample Schema Evolution Serialization Code

The next section provides a program that takes an argument that causes the program to perform one of three actions:

- Create a database with some data in it, such as instances of **OSHashtable**, **OSVector**, or linked lists.
- Use object serialization to dump data in the database to a file.
- Use object serialization to reload the data from the file.

For example, you can use the sample program to add a new field to the **LinkedList** class. To do so, follow these steps:

1 Place the code in a file called **DumpReload.java**.

2 Set your **CLASSPATH** environment variable to include the directory that contains the **osjcfpout** file and the **DumpReload.java** file.

3 Compile the program with the command

javac DumpReload.java.

4 Run the postprocessor to annotate the **DumpReload** and **LinkedList** classes:

```
osjcfp -dest osjcfpout DumpReload.class \
DumpReload$LinkedList.class
```

5 Create the database:

```
java DumpReload create data.odb
```

6 Use serialization to dump the data:

```
java DumpReload dump data.odb data.out
```

7 Change the **LinkedList** class. Do this by removing the comment flag from the **newField** field in **LinkedList**.

8 Recompile the class:

```
javac DumpReload.java
```

9 Rerun the postprocessor to annotate the **DumpReload** and **LinkedList** classes:

```
osjcfp -dest osjcfpout DumpReload.class \
DumpReload$LinkedList.class
```

10 Use serialization to reload the data:

```
java DumpReload reload data.odb data.out
```

Sample Code for Using Serialization to Perform Schema Evolution

Here is the sample program for using serialization to perform schema evolution.

```
import COM.odi.*;
import COM.odi.util.OSHashtable;
import COM.odi.util.OSVector;

import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;

import java.util.Enumeration;

public class DumpReload {
   static public void main(String argv[]) throws Exception {
      if (argv.length >= 2) {
         if (argv[0].equalsIgnoreCase("create")) {
            createDatabase(argv[1]);
         } else if (argv[0].equalsIgnoreCase("dump")) {
            dumpDatabase(argv[1], argv[2]);
         } else if (argv[0].equalsIgnoreCase("reload")) {
            reloadDatabase(argv[1], argv[2]);
         } else {
            usage();
         }
      } else {
         usage();
      }
   }

   static void usage() {
      System.err.println(
         "Usage: java DumpReload OPERATION ARGS...\n" +
         "Operations:\n" +
         "   create DB\n" +
         "   dump FROMDB TOFILE\n" +
         "   reload TODB FROMFILE\n");
      System.exit(1);
   }

   /* Create a database with 3 roots. Each root contains an
      OSHashtable of OSVectors that contain some Strings. */

   static void createDatabase(String dbName) throws Exception {
      ObjectStore.initialize(null, null);

      try {
         Database.open(dbName, ObjectStore.UPDATE).destroy();
      } catch (DatabaseNotFoundException DNFE) {
      }
```

```
    Database db = Database.create(dbName, 0644);

    Transaction t = Transaction.begin(ObjectStore.UPDATE);
    for (int i = 0; i < 3; i++) {
      OSHashtable ht = new OSHashtable();
      for (int j = 0; j < 5; j++) {
        OSVector vec = new OSVector(5);
        for (int k = 0; k < 5; k++)
          vec.addElement(new LinkedList(i));
        ht.put(new Integer(j), vec);
      }
      db.createRoot("Root" + Integer.toString(i), ht);
    }
    t.commit();
    db.close();
  }

  static void dumpDatabase(String dbName, String dumpName)
    throws Exception {
    ObjectStore.initialize(null, null);

    Database db = Database.open(
            dbName, ObjectStore.READONLY);

    FileOutputStream fos = new FileOutputStream(dumpName);
    ObjectOutputStream out = new ObjectOutputStream(fos);

    Transaction t = Transaction.begin(ObjectStore.READONLY);

    /* Count the roots and write out the count. */
    Enumeration roots = db.getRoots();
    int nRoots = 0;
    while (roots.hasMoreElements()) {
      String rootName= (String) roots.nextElement();
      /* Skip internal OSJI header */
      if (!rootName.equals("_DMA_Database_header")) nRoots++;
    }
    out.writeObject(new Integer(nRoots));

    /* Rescan and write out the data.
       The deepFetch() call is necessary because it obtains the
       contents of all objects that are reachable from root,
       and makes them available for serialization. */
    roots = db.getRoots();
    while (roots.hasMoreElements()) {
      String rootName = (String) roots.nextElement();
      if (!rootName.equals("_DMA_Database_header")) {
        out.writeObject(rootName);
        Object root = db.getRoot(rootName);
        ObjectStore.deepFetch(root);
        out.writeObject(root);
    }

    t.commit();
```

```
        out.close();
    }

    static void reloadDatabase(String dbName, String dumpName)
        throws Exception {
        ObjectStore.initialize(null, null);

        try {
            Database.open(
                dbName, ObjectStore.OPEN_UPDATE).destroy();
        } catch (DatabaseNotFoundException DNFE) {
        }
        Database db = Database.create(dbName, 0644);

        FileInputStream fis = new FileInputStream(dumpName);
        ObjectInputStream in = new ObjectInputStream(fis);

        Transaction t = Transaction.begin(ObjectStore.UPDATE);

        int nRoots = ((Integer) in.readObject()).intValue();
        while (nRoots-- > 0) {
            String rootName = (String) in.readObject();
            Object rootValue = in.readObject();
            System.out.println("Creating " + rootName + " " + rootValue);
            db.createRoot(rootName, rootValue);
        }

        t.commit();
        db.close();
    }

    static
    class LinkedList implements java.io.Serializable {
        private int value;
        private LinkedList next;
        private LinkedList prev;
        //private Object newField;
        static final long serialVersionUID = -5286408624542393371L;

        LinkedList(int value) {
            this.value = value;
            this.next = null;
            this.prev = null;
        }
    }
}
```

Destroying a Database

Destroying a database makes all objects in the database permanently inaccessible. You cannot recover a destroyed database except from backups.

To destroy a database, call the **destroy()** method on the **Database** subclass instance, for example:

```
db.destroy();
```

The database must be open-for-update and a transaction cannot be in progress.

When you destroy a database, all persistent objects that belong to that database become stale.

Obtaining Information About a Database

You can call methods on a database to answer the following questions.

Is a Database Open?

To determine whether or not a database is open, call the **isOpen()** method on the database, for example:

```
db.isOpen();
```

This expression returns true if the database is open. It returns false if the database is closed or if it was destroyed. To determine whether false indicates a closed or destroyed database, try to open the database.

What Kind of Access Is Allowed?

To check what kind of access is allowed for an open database, call the **getOpenMode()** method on the database. The database must be open or ObjectStore throws DatabaseNotOpenException. The method signature is

```
public int getOpenMode()
```

This method returns one of the following constants:

- **ObjectStore.READONLY**
- **ObjectStore.UPDATE**
- **ObjectStore.MVCC**

Here is an example of how you can use this method:

```
void checkUpdate(Database db) {
   if (db.getOpenMode() != ObjectStore.UPDATE)
      throw new Error("The database must be open for update.");
}
```

What Is the Pathname of a Database?

To find out the pathname of a database, call the **getPath()** method on the database, for example:

```
String myString = db.getPath();
```

What Is the Size of a Database?

To obtain the size of a database, call the **getSizeInBytes()** method on the database. The database must be open and a transaction must be in progress, for example:

```
db = Database.open("myDb.odb", ObjectStore.READONLY);
Transaction tr = Transaction.begin(ObjectStore.READONLY);
long dbSize = db.getSizeInBytes();
```

This method does not necessarily return the exact number of bytes that the database uses. The value returned might be the result of your operating system's rounding up to a block size. You should be aware of how your operating system handles operations such as these.

Which Session Is the Database or Segment Associated With?

To obtain the session with which a database or segment is associated, call the **Placement.getSession()** method. The method signature is

public Session Placement.getSession()

Which Objects Are in the Database?

The **osjshowdb** utility displays information about one or more databases. This utility is useful when you want to know how many and what types of objects are in a database. You can use this utility to verify the general contents of the database.

Information about the **osjshowdb** utility is in osjshowdb: Displaying Information About a Database on page 397.

Are There Invalid References in the Database?

The **osjcheckdb** utility or the **Database.check()** method checks the references in a database. This tool scans a database and checks that there are no references to destroyed objects. The most likely cause of dangling references is an incorrectly written program. You can fix dangling references by finding the objects that contain them and overwriting the invalid references with something else, such as a null value. In addition to finding references to destroyed objects, the tool performs various consistency checks on the database.

Information about the **osjcheckdb** utility is in osjcheckdb: Checking References in a Database on page 395.

Implementing Cross-Segment References for Optimum Performance

You can improve application performance by clustering together objects that are expected to be used together. Effective clustering both reduces the number of disk and network transfers the applications require and increases concurrency among applications. You can use a segment to store groups of objects that are accessed as a logical cluster.

ObjectStore allows an object in one segment to refer to an object in another segment in the same database or in a different database. The advantage of cross-segment references is that you can cluster related objects in separate segments and also provide links among the segments. Your database can continue to grow without necessarily slowing down access times.

There is a procedure to follow to implement cross-segment references that results in the best performance. A description of the overhead involved with cross-segment references shows why it is important to follow this procedure.

Procedure for Defining Cross-Segment References

To obtain the greatest benefit from cross-segment references, you must do the following:

1 Create a plan for organizing your objects into clusters and for assigning the clusters to segments. You might want to put one or several logical clusters into a segment. Base your plan on the size of the objects and the patterns of access.

2 In each planned segment, designate a few objects to be the points of reference to the other objects in the segment.

3 Use the **ObjectStore.migrate()** method to store the designated objects in their respective segments.

You do not want these designated objects to become persistent through reachability when a transaction commits. (See transitive persistence on page 15.) If they did, ObjectStore would store them in the same segment as whatever objects pointed to them. The **migrate()** method allows you to specify the segment in which to store an object. Be sure to specify **true**

for the **export** argument to **migrate()**. For an object to be referred to by an object in another segment, the referred-to object must be exported.

4 Ensure that the migrated objects reference the other objects in their clusters.

This allows the other objects to be stored in the same segment as the object that references them. When you commit the transaction, ObjectStore automatically stores all referenced objects in the same segment as the migrated object, unless they were explicitly migrated to another segment.

5 Ensure that no objects outside the cluster's segment try to store a reference to an object that was stored by transitive persistence.

6 There might be some objects that are not part of a cluster, but that must be referenced by objects in multiple segments. Migrate such objects into convenient segments and be sure to export them.

It is most efficient to export objects when explicitly migrating them. Although you can call **ObjectStore.export()** to change an object in the database to be an exported object, doing so can be slow. The time it takes to export an object already in the database is proportional to the number of objects in the segment that contains the object.

You want to minimize the number of exported objects in each segment. A segment that contains large numbers of exported objects has more overhead for the persistent system data needed to describe the exported objects. Also, references to exported objects from within their own segments might take longer because of the increased size of the system data.

You only need to invoke **ObjectStore.migrate()** on objects that are not yet in the database. To change which segment an object is in, see Explicitly Migrating Exported Objects on page 103. You can call the **migrate()** method repeatedly with the same arguments, but there is usually no need to do so.

Exporting Objects

When you export an object, whether you use **migrate()** or **export()**, ObjectStore

- Assigns an export ID to the exported object
- Adds the object to the export table

When you export an object that is already in the database (**export()**), ObjectStore also checks every object already in the segment to determine if it refers to the exported object. If it does, ObjectStore updates the reference from a local reference to a reference through the export table.

If the segment is empty or relatively empty, the actual overhead is small because there are few, if any, objects to check. If there are already many objects in the segment, the overhead of checking each one can be quite large.

Consequently, it is preferable to use the **ObjectStore.migrate()** method and specify **true** for the **export** argument when you store the object in the database. This also ensures that you are storing the object in the intended segment.

Suppose you forget to export a particular object when you store it in the segment. You can fix this by calling the **ObjectStore.export()** method on the object. However, by the time you recognize the need to export an object, there might already be many objects in the segment. The overhead for exporting the object can be quite large.

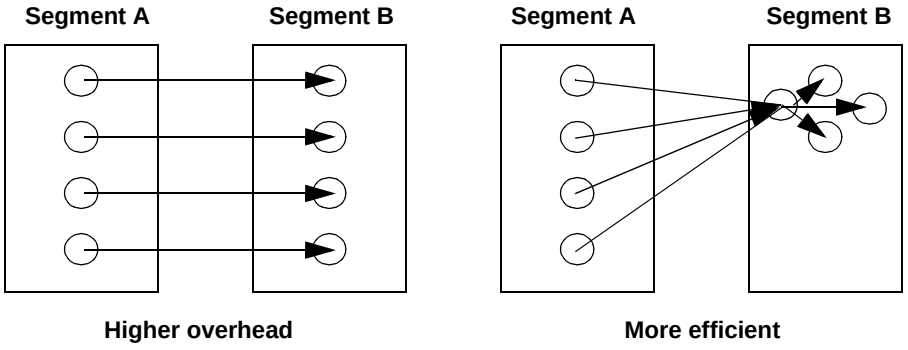
With the exception of ObjectStore collection objects, you cannot export peer objects. If you try to export a peer object that is not an ObjectStore collection object, ObjectStore throws ObjectException.

How Many Exported Objects Are Needed?

You do not want too many objects in a segment to be referred to by objects in other segments. In other words, you want to minimize the number of exported objects in each segment. However, there is no additional cost if you have many objects that refer to the same exported object.

The overhead of having many exported objects in a segment comes from having many entries in the export table. Access to an exported object is through the export table. Having many exported objects is less efficient than having clustered objects because access to the exported objects requires an indirect lookup through the export table. While the lookup is implemented with a highly-tuned hashing mechanism, the overhead for maintaining the export table increases as it grows. Finally, if the exported objects are very volatile, the database locking mechanism can result in additional wait time for users who are creating or accessing exported objects.

For these reasons, it is desirable to design your application so that only a relatively small number of nonvolatile objects must be exported. The figure below shows the optimum way to set up cross-segment references.



Explicitly Migrating Exported Objects

Objects can become persistent because they are reachable from an already persistent object or a database root. ObjectStore places the newly persistent object in the same segment as the object from which it can be reached. Consequently, you must be careful to explicitly migrate those objects that must be exported.

In a transaction, if you create a situation in which an object is referred to by objects in more than one segment, and the object is not exported, ObjectStore throws ObjectNotExportedException when you try to do one of the following:

- Commit the transaction.
- Evict the unexported object.

This situation can happen if an application stores a reference to a nonexported object from one segment into an object in another segment. The check for this does not occur when you store the reference.

Database Operations and Transactions

For each database operation, there are rules about whether it can be performed

- Inside a transaction
- Outside a transaction
- Both inside and outside a transaction

The following table shows which rules apply to which operations. If your application tries to perform a database operation that breaks a rule, you receive a run-time exception.

Database Operation	*Can Be Performed Inside/Outside Transaction?*	*Database Open?*
acquireLock()	Inside	Open
check()	Inside	Open
close()	Outside	Open
create()	Outside	Not applicable
createRoot()	Inside	Open
createSegment()	Inside	Open
destroy()	Outside	Open
destroyRoot()	Inside	Open
GC()	Outside	Open
getDefaultSegment()	Inside	Open
getOpenMode()	Both	Open
getPath()	Both	Open
getRoot()	Inside	Open
getRoots()	Inside	Open
getSegment()	Inside	Open
getSegments()	Inside	Open
getSizeInBytes()	Inside	Open
installTypes	Inside	Open
isOpen()	Both	Open or closed
of()	Inside	Open
open()	Both	Open or closed
setDefaultSegment()	Inside	Open
setRoot()	Inside	Open
show()	Inside	Open

Upgrading Databases for Use with the JDK 1.2

The JDK 1.2, currently in beta testing but expected to be released soon by Sun Microsystems, computes hash codes for **String** types differently than the JDK 1.1. As a result, databases that depend on **String** hash codes force the database to only be usable from the JDK 1.1 or the JDK 1.2, but not both.

ObjectStore marks databases to specify whether the JDK 1.1 or JDK 1.2 was used when the database was created. It then ensures that the same JDK version is used when the database is accessed.

Databases created with releases previous to this one are assumed to have been created with the JDK 1.1. With this release of ObjectStore, you can use only the JDK 1.1 to access these databases unless you upgrade them for use with the JDK 1.2. ObjectStore provides a tool and an API for

- Upgrading databases created with the JDK 1.1 to be accessible with the JDK 1.2.
- Marking databases as already created with the JDK 1.2.

After you upgrade a database, you can no longer use it with the JDK 1.1.

To use the upgrade tool, see osjuphsh: Upgrading String Hash Codes in Databases on page 402. To use the API, see **COM.odi.Upgrade.upgradeDatabaseStringHash()** in the *ObjectStore Java API Reference.*

The upgrade facility also allows you to mark a database as not containing any objects that depend on **String** hash codes. If you mark a database in this way, you can use either the JDK 1.1 or the JDK 1.2 to access the database.

After you upgrade your database, Object Design recommends that you run the garbage collector on the database. Upgrading the database creates some garbage.

If you use the upgrade API to perform the upgrade, watch out for a problem that involves cross-database references from the upgraded database to other databases. If the database being upgraded contains references to other databases, those other databases might need to be opened during the upgrade. If they are

opened, the upgrade API leaves them open when it is done. The upgrade allows the other databases to be opened regardless of whether they have been upgraded because of the limited way in which the other databases are required by the upgrade. However, if the application trys to use those database after performing the upgrade, it receives exceptions if the other database are not already upgraded.

The workaround is to ensure that you upgrade all databases you intend to use before you try to access any upgraded database.

Chapter 5
Working with Transactions

A transaction is a logical unit of work. It is a consistent and reliable portion of the execution of a program. In your code, you place calls to the ObjectStore API to mark the beginnings and ends of transactions. Initial access to a persistent object must always take place inside a transaction. Depending on how the transaction is committed, additional access to persistent objects might be possible.

Either the database is updated with all of a transaction's changes to persistent objects, or the database is not updated at all. If a failure occurs in the middle of a transaction, or you decide to abort the transaction, the contents of the database remain unchanged.

Contents

This chapter discusses the following topics:

Starting a Transaction

ObjectStore provides the **COM.odi.Transaction** class to represent a transaction. You should not make subclasses of this class.

This section discusses the following topics:

Calling the begin() Method

To start a transaction, call the **begin()** method on the **Transaction** class. This returns an instance of **Transaction** and you can assign it to a variable. The method signature is

Method signature

public static Transaction begin(int type)

The transaction type determines whether ObjectStore waits for a database lock. There can be only one write lock on a database. There can be multiple read-only locks on a database.The type of the transaction can be **ObjectStore.UPDATE** or **ObjectStore.READONLY**.

If there is no open database when you start the current transaction, ObjectStore tries to obtain a read lock as soon as the session tries to open a database.

Example

```
Transaction tr = Transaction.begin(ObjectStore.UPDATE);
```

This example returns a **Transaction** object that represents the transaction just started. The result is stored in **tr**. This is an update transaction, which means that the application can modify database contents.

Allowing Objects to Be Modified in a Transaction

To modify persistent objects, you must specify the transaction type to be **ObjectStore.UPDATE**. Also, any database you modify must have been opened for update. Note that even if you open a database for read-only, ObjectStore allows you to start an update transaction. An application does not receive an exception until it tries to modify persistent objects inside the read-only database.

If you try to modify persistent data in a read-only transaction, ObjectStore throws UpdateReadOnlyException.

Difference Between Update and Read-Only Transactions

You can start a transaction for **READONLY** or for **UPDATE**. The only difference between the two types is that when you start a transaction for **READONLY**, ObjectStore performs additional checks during the transaction and when you commit the transaction. These checks ensure that changes are not saved in the database if they were made in a read-only transaction. There is no difference in performance between a read-only transaction and an update transaction.

Working Inside a Transaction

A transaction is associated with the session that is associated with the thread that starts the transaction. A transaction remains active until you explicitly commit it or until it aborts. A session can have only one active transaction. Concurrent transactions must be in separate sessions.

This section discusses the following topics:

Separate transactions that access the same database compete with one another for locks on the objects that they access. This can cause one transaction to wait for another to release its locks. Alternatively, it can cause a transaction deadlock situation in which two or more transactions wait for each other. This forces one transaction to abort.

Two transactions can never update the same object at the same time. However, two transactions can both open the same database for update at the same time and they can concurrently make updates to different parts of the database.

Obtaining the Session Associated with the Current Transaction

The current session is the session that a thread most recently joined. To obtain the session that is associated with the current transaction, call the **Transaction.getSession()** method. The method signature is

public Session Transaction.getSession()

To obtain the transaction that is associated with the current session, call the **Session.currentTransaction()** method. The method signature is

public Transaction Session.currentTransaction()

To determine whether or not there is a transaction in progress for the current session, call the **Transaction.inTransaction()** method on **Transaction**. The method signature is

public static boolean inTransaction()

This method returns true if there is a transaction in progress for the current session. Otherwise, it returns false. It is worth noting that **inTransaction()** return false if the calling thread is not joined to the current session. This can be important if you use an unassociated thread to check whether there is a transaction and then try to close the database based on a false response. The previously unassociated thread would be automatically joined to the session to close the database. If a transaction is actually in progress, ObjectStore throws TransactionInProgressException, which is, of course, unexpected since **inTransaction()** returned false.

Transaction Already in Progress

Nested transactions are not allowed. If you try to start a transaction when a transaction for the current session is already in progress, ObjectStore throws TransactionInProgressException.

Obtaining Transaction Objects

An application can obtain the transaction object for the current thread by calling the static **current()** method on the **Transaction** class. The method signature is

```
public static Transaction current()
```

This method returns the transaction object associated with the current session, for example:

```
Transaction.current().commit()
```

This example commits the current transaction. If no transaction is in progress, **current()** throws NoTransactionInProgressException.

Performing a Transaction Checkpoint

You can use the **Transaction.checkpoint()** method to commit changes but continue working with the same persistent objects. When you call the **checkpoint()** method, you specify whether to retain persistent objects as hollow objects or make all persistent objects stale. This is useful when you are trying to improve concurrency, or when you are at a consistent state and want to save your changes but keep working. See Checkpoint: Committing and Continuing a Transaction on page 325.

Setting a Transaction Priority

When there is a deadlock, the Server uses the transaction priority as one of the criteria to determine which transaction to abort. See Helping Determine the Transaction Victim in a Deadlock on page 332.

Ending a Transaction

When transactions terminate successfully, they commit, and their changes to persistent objects are saved in the database. When transactions terminate unsuccessfully, they abort, and their changes to persistent objects are discarded.

For read-only transactions, there are no advantages to committing them rather than aborting them, nor to aborting them rather than committing them.

This section discusses the following topics:

Committing Transactions

ObjectStore provides the **Transaction.commit()** method for successfully ending a transaction. When an application commits a transaction, ObjectStore

- Saves and commits any changes in the database
- Performs transitive persistence if applicable (see page 15)
- Sets the state of persistent objects that were accessed or referenced in the transaction

Transitive persistence

When ObjectStore commits a transaction, it checks to see if there are any transient objects that are referred to by persistent objects. If there are, and if all referred-to objects are persistence-capable objects, ObjectStore stores the referred-to objects in the segments that contains the referring objects. This is the process of transitive persistence. If at least one referred-to object is not persistence-capable, ObjectStore throws ObjectNotPersistenceCapableException.

Also, a transient referred-to object cannot be referenced from more than one segment. All cross-segment references must be to exported objects, which means you must have explicitly migrated the object to one of the segments and specified that it is exported. If ObjectStore finds an unexported object that is referred to from more than one segment, it throws AbortException.

Making objects stale

To commit a transaction and make the state of persistent objects stale, call the **commit()** method with no argument. The method signature is

public void commit()

For example, **tr.commit();**

Setting object state

To commit a transaction and be flexible about the state of persistent objects after the transaction, call the **commit(retain)** method on the transaction. The values you can specify for **retain** are described in Committing Transactions to Save Modifications on page 153. The method signature is

public void commit(int retain)

The following example commits the transaction and specifies that the contents of the active persistent objects should remain available to be read.

```
tr.commit(ObjectStore.RETAIN_READONLY);
```

What Can Cause a Transaction Commit to Fail?

When ObjectStore tries to commit a transaction, if ObjectStore encounters any of the situations listed below, it causes the transaction commit to fail. When ObjectStore aborts a transaction commit, it throws AbortException.

- A nonexported object is reachable from an object in a different segment.
- A persistent object references an object that is not persistence-capable.
- A persistent object was updated to reference a stale object.
- A deadlock was encountered during commit. This is more likely to happen when lazy write locking is enabled.
- There is a Server error. This can happen when there is not enough disk space to fit everything you stored in the database. It can also happen because of a broken network connection, Server failover, or a disk error, that makes it impossible for the Server to complete the commit. Failover also causes ObjectStore to abort the transaction.

Aborting Transactions

ObjectStore provides the **Transaction.abort()** method for unsuccessfully ending a transaction. An abort can happen explicitly through the **Transaction.abort()** method or implicitly because a session is terminated or there is a system exception. When an application aborts a transaction, ObjectStore

- Ensures that the objects in the database are as they were just before the aborted transaction started
- Sets the state of persistent objects from the transaction
- Returns any transient objects that were made persistent during the transaction to their transient state

Transient objects

Only the state of the database is rolled back. The state of transient objects is not undone automatically. For example, if you created new transient objects during the transaction, they still exist after the transaction aborts. Applications are responsible for undoing the states of transient objects. Any form of output that occurred before the abort cannot be undone.

Open databases

If you opened any databases during the transaction, ObjectStore closes them. Any databases that were open before the aborted transaction was started remain open after the abort operation.

Application failure

If an application fails during a transaction, when you restart the application the database is as it was before the transaction started. If an application fails during a transaction commit, when you restart the application either the database is as it was before the transaction that was being committed *or* the database reflects all the transaction's changes. This depends on how far along in the commit process the application was when it terminated. Either *all or none* of the transaction's changes are in the database.

abort()

To abort a transaction and set the state of persistent objects to the state specified by **Transaction.setDefaultAbortRetain()**, call the **abort()** method. The default state is stale. The method signature is

```
public void abort()
```

For example,

```
tr.abort();
```

abort(retain)

To abort a transaction and specify a particular state for persistent objects after the transaction, call the **abort(retain)** method on the transaction. The values you can specify for **retain** are described in Specifying a Particular State for Persistent Objects on page 171. The method signature is

```
public void abort(int retain)
```

The following example aborts the transaction and specifies that the contents of the active persistent objects should remain available to be read.

```
tr.abort(ObjectStore.RETAIN_READONLY);
```

Handling Automatic Transaction Aborts

ObjectStore sometimes automatically aborts a transaction because

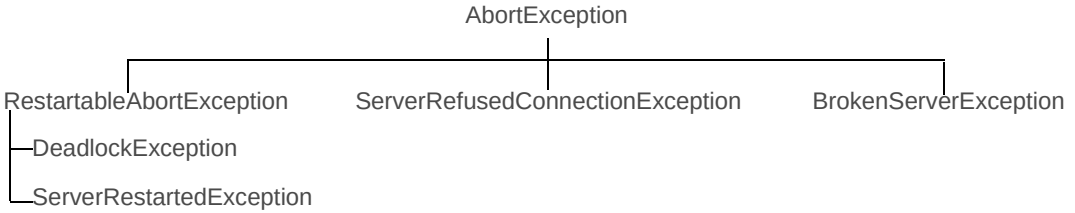
- There is a transaction deadlock.
- The connection to the Server is broken.
- The Server has been restarted.
- The Server refuses the connection.
- An exception occurs during a transaction commit. For example, ObjectStore aborts a transaction if you try to commit it when a persistent object refers to a transient object that is not persistence-capable. This type of automatic abort is discussed in various sections of Chapter 6, Storing, Retrieving, and Updating Objects.

Results of Transaction Abort

When ObjectStore aborts a transaction, it rolls back the persistent state to what it was before the transaction. ObjectStore does not roll back the transient state. Any form of output that occurred before the abort cannot be undone. Therefore, it is generally good practice to perform output outside a transaction.

Description of Transaction Abort Exceptions

When ObjectStore aborts a transaction, it throws an exception that indicates the reason for the abort and whether or not it makes sense to retry the transaction. Here is the hierarchy of transaction abort exceptions:



The superclass of exceptions for transaction abort is AbortException. The superclass of exceptions for which it makes sense to retry the aborted exception is RestartableAbortException. AbortExceptions that do not extend RestartableAbortException

indicate that before the transaction can proceed, some action is probably required to correct the problem that caused the exception.

Exceptions that require intervention

Exceptions for which some action is probably required include

- AbortException — Thrown during transaction commit
- ServerRefusedConnectionException — Thrown when the Server is unavailable when the client tries to connect to it
- BrokenServerException — Thrown when the Server becomes unavailable while the client is connected to it. It indicates that the Server refused to continue the connection.

If your application receives these exceptions, you should check the situation before you retry the transaction. You should not handle these exceptions with retry loops.

Exceptions that indicate retrying

A RestartableAbortException indicates that it makes sense to retry the aborted transaction immediately without taking any other action. This class has two subclasses, which indicate the particular reason for the abort.

- ServerRestartedException indicates that the Server was restarted, for example, when failover occurs. The client was already connected to the Server. The client receives this exception when it next tries to contact the Server. ObjectStore aborts the transaction because it cannot commit it. However, things should be fine for a subsequent transaction. If your application receives this exception, it should roll back transient state before it retries the transaction.
- DeadlockException indicates that transactions were deadlocked. It is possible that the next transaction will also deadlock. However, there is a reasonable chance that the transaction that was allowed to proceed has made progress so that the deadlock is not repeated.

Restarting Aborted Transactions

ObjectStore does not retry transactions that are automatically aborted, even if they extend RestartableAbortException. If you want to retry the transaction when your application receives a RestartableAbortException, you must include code to do this in your

application. An example of this code follows. If you run the same example in separate VMs at the same time, it produces deadlocks.

```
import COM.odi.*;
class DeadlockTest {
 static final int MAX_RETRIES = 10;

 public static void main(String[] args) {
   ObjectStore.initialize(null, null);
   Database db = getDatabase();
   while (true) {
    test(db);
   }
 }

 static void test(Database db) {
   int retries;
   for (retries = 0; retries < MAX_RETRIES; retries++) {
    try {
       Transaction.begin(ObjectStore.UPDATE);
       Integer value = increment(db);
       Transaction.current().commit();
       System.out.println("Value = " + value + ", retries = " + retries);
       break;
    }  catch (RestartableAbortException e) {
    }
   }
   if (retries >= MAX_RETRIES)
    System.out.println("Gave up after " + retries + " retries.");
 }

 static Database getDatabase() {
   try {
    return Database.open("test.odb", ObjectStore.UPDATE);
   }catch (DatabaseNotFoundException e) {
    Database db = Database.create("test.odb", 0664);
    Transaction.begin(ObjectStore.UPDATE);
    db.createRoot("root", null);
    Transaction.current().commit();
    return db;
   }
 }

 static Integer increment(Database db) {
   Integer value = (Integer)db.getRoot("root");
   if (value == null)
    value = new Integer(0);
   else
    value = new Integer(value.intValue() + 1);
   db.setRoot("root", value);
   return value;
 }
}
```

Handling Deadlocks

A simple deadlock occurs when one transaction holds a lock on a page that another transaction is waiting to access, while at the same time this other transaction holds a lock on a page that the first transaction is waiting to access. Neither process can proceed until the other does. There are other, more complicated forms of deadlock that are analogous.

ObjectStore has a deadlock detection facility that breaks deadlocks, when detected, by aborting one of the transactions involved in the deadlock. By aborting one transaction (the victim), ObjectStore causes its locks to be released so other processes can proceed.

ObjectStore throws DeadlockException when it detects a deadlock. This causes ObjectStore to abort the transaction that causes the deadlock. You can change which transaction ObjectStore aborts by changing the setting of the **Deadlock Victim** Server parameter.

ObjectStore does not detect deadlocks when the deadlock is distributed across multiple Servers.

Determining Transaction Boundaries

When determining whether or not to commit a transaction, consider database state, whether or not to combine transactions, and interdependencies among cooperating threads.

Inconsistent Database State

You should not commit a transaction if the database is in a logically inconsistent state. A database is considered to be in an inconsistent state if at that moment a just-started transaction would encounter problems upon viewing the current state of the data.

Consider your database to be something that moves from one consistent state to another. You should commit a transaction only when the state is consistent. When is a database consistent? When the answer to this question is yes: If you start your application at this very moment, is the database completely usable exactly the way it is now?

For example, suppose your database contains information about married couples. Couples refer to one another through a **spouse** field. At a particular moment, suppose a person in the database refers to another person in the database through its **spouse** field, but that spouse does not refer to the first person. At that moment, the database is in an inconsistent state.

Another inconsistency to avoid is retaining references to objects that are not reachable from within the database. Such a situation can cause trouble if the application fails between transactions or if the garbage collector runs.

When the database state is consistent, you might decide not to commit the transaction. However, if you do not commit, you risk losing changes if ObjectStore aborts the transaction. You should always commit changes before you inform a user or some other interface that a particular task was accomplished.

Combining Transactions

The transaction commit operation requires network interaction with the ObjectStore Server. Consequently, you might be able to improve performance by combining several logical transactions into a single transaction.

To do this, skip the call to **commit()** and subsequent call to **Transaction.begin()** for a series of sequential transactions. If many of the same objects are used in the different transactions, the single combined commit operation is more efficient than many small commit operations would be.

However, this strategy means that you risk losing changes if ObjectStore aborts the transaction before you commit it. All the logical transactions up to that point would be rolled back with the current logical transaction. You should always commit changes before you inform a user or some other interface that a particular task was accomplished.

Multiple Cooperating Threads

If your application uses cooperating threads, you must take this into account when determining when to commit transactions. For example, you do not want to create a situation where one thread commits a transaction while a cooperating thread is updating persistent objects. The **commit()** method might make all persistent objects stale for all cooperating threads. If the **commit()** method retains persistent objects, ObjectStore discards any modifications to retained persistent objects at the start of the next transaction. You must coordinate the **Transaction.begin()** and **Transaction.commit()** operations among cooperating threads.

Synchronizing threads is like having a joint checking account. Suppose the amount in the checking account is $100.00. Your partner writes a check for $50.00. Then you try to cash a check for $75.00. This does not work. It does not matter that it was your partner and not you who wrote the check for $50.00. You and your partner have to cooperate.

Performance Considerations

Committing a transaction, even a read-only transaction, has a certain amount of overhead associated with it. If you have a lot of small transactions. You might want to consider combining some of them into larger transactions.

Chapter 6
Storing, Retrieving, and Updating Objects

This chapter provides information about how to store data in a database and also how to read it back and update it. An application can access persistent data only inside a transaction and only when the database is open.

Contents

This chapter discusses the following topics:

Storing Objects

ObjectStore's Java API preserves the automatic storage management semantics of Java. Objects become persistent when they are referenced by other persistent objects. This is called persistence by reachability. The application defines persistent roots and, when it commits a transaction, ObjectStore finds all objects reachable from persistent roots and stores them in the database.

To store objects in a database, do the following:

1 Open the database or create the database in which you want to store the objects. Be sure the database is opened for update. See page 68.

2 Start an update transaction. See page 110.

3 Create a database root or access an existing database root and specify that it refers to one of the objects you want to store. See page 131.

4 Ensure that any other objects you want to store are referenced by the object to which the database root refers.

5 Commit the transaction. This stores the object that the database root refers to and any objects that object references. See page 116.

In general, you should not create a root for each object you want to store in a database. You must create a root to store some object in a database by which all other objects can ultimately be reached.

How Objects Become Persistent

Objects can become persistent in several ways:

- An application assigns a transient object to a database root. ObjectStore immediately migrates the object to the default segment in the database. When the transaction commits, any transient objects that are reachable from the object assigned to the root are also stored in the default segment.
- A transient object is reachable from a persistent object. When the transaction commits, ObjectStore stores the reachable object in the same segment as the persistent object.
- An application invokes the **ObjectStore.migrate()** method on a transient object and specifies a particular segment. When the transaction commits, ObjectStore stores the migrated object in the specified segment. ObjectStore also stores in the specified segment any transient objects that are reachable from the migrated object.

Storing Objects in a Particular Segment

When you want to store objects in a particular segment that is not the default segment, follow these steps:

1 Open a database and start a transaction.

2 If you have not already created the segment in which you want to store the objects, create the segment with the **Database.createSegment()** method.

3 Invoke **ObjectStore.migrate()** to store an object and specify the segment in which you want to store the object.

4 If there are other transient objects that you want to store in the same segment, make those objects reachable from the migrated object.

5 Commit the transaction.

ObjectStore stores all transient objects that are reachable from the migrated object in the same segment as the migrated object.

To allow access to the objects, you must also assign some objects to database roots.

Caution

If you are going to have objects in one segment that refer to objects in another segment, it is crucial that you do some planning before you store any objects. Implementing Cross-Segment References for Optimum Performance on page 99 provides information about the issues to consider. It is important to familiarize yourself with this information because doing so can help you avoid problems.

What Is Reachability?

An object **B** is considered to be reachable from object **A** when **A** contains a reference to **B**. **B** is also reachable from **A** when **A** contains a reference to some object and that object contains a reference to **B**. There are no limits to levels of reachability.

Situations to Avoid

When a transaction commits, you must ensure that objects in different segments do not refer to the same transient object. If this situation exists, ObjectStore cannot determine which segment to store the transient object in. Consequently, ObjectStore throws AbortException**,** aborts the transaction, and informs you that an unexported object is referred to by an object in another segment.

To avoid this situation, you must explicitly migrate transient objects to a database segment when those transient objects are referred to by objects in more than one segment. You must do this before the application commits the transaction and you must export the objects when you migrate them.

When a transaction commits, you must ensure that any transient objects that are reachable from persistent objects are persistence-capable. If one such object is not, ObjectStore throws AbortException, aborts the transaction, and informs you that a reachable object is not persistence-capable.

Storing Java-Supplied Objects

Some Java-supplied classes are persistence-capable. Others are not persistence-capable and cannot be made persistence-capable. A third category of classes can be made persistence-capable, but there are important issues to consider when you do so. Be sure to read Java-Supplied Persistence-Capable Classes on page 360.

Working with Database Roots

A root is a reference to an individual object. You can get by with a single root, but you might find it convenient to have more. In general, you do not want every object in the database to be associated with a root. This is bad for performance. Each root refers to exactly one object. More than one root can refer to the same object. You cannot navigate backwards from the referenced object to the database root.

This section discusses the following topics:

Creating Database Roots

When you create a database root, you give it a name and you assign an object to it. The database root refers to that object and the application can use the root name to access that object. In other words, the object that you assign to a root is the value of that root. The database root and the object assigned to the root are two distinct objects.

You must create a database root inside a transaction. Call the **Database.createRoot()** method on the database in which you want to create the root. The method signature for this instance method on **Database** is

```
public void createRoot(String name, Object object)
```

The name you specify for the root must be unique in the database. If it is not unique, DatabaseRootAlreadyExistsException is thrown. The object that you specify to be referred to by the root can be either transient and persistence-capable, or persistent (including null). If it is not yet persistent, ObjectStore immediately makes it persistent. ObjectStore migrates it to the default segment, which is the segment returned by **Database.getDefaultSegment()**.

More than one root can reference the same object; an object can be associated with more than one root. For example:

```
db.createRoot("Root1", anObject);
db.createRoot("Root2", anObject);
```

Example

For example, suppose you create the variable **db** to be a handle to a database opened for update and an object called **anObject**, and start an update transaction. The following line creates a database root:

```
db.createRoot("MyRootName", anObject);
```

Results

In the database referred to by **db**, this creates a database root named **"MyRootName"** and specifies that it refers to **anObject**. ObjectStore immediately stores **anObject** in the database referred to by **db**. When the transaction commits, ObjectStore stores in the database referred to by **db** any objects that are reachable from **anObject,** if they are not already in the database. If **anObject** or any object it references refers to any transient objects that are not persistence-capable, and you try to commit the transaction, ObjectStore throws ObjectNotPersistenceCapableException.

Retrieving Root Objects

When you retrieve a root object, you obtain a reference to the object that is the value of the root. For example, suppose you assign an **OSVector** object, **myOSVector**, to a root named **"myOSVectorRoot"**. When you get **myOSVectorRoot**, you receive a reference to **myOSVector**. ObjectStore does not fetch the entire vector. Now you can obtain a reference to any object in the vector. For example, you can get a reference to the fifth element in the vector like this:

```
AnObject obj = (AnObject)myOSVector.elementAt(5);
```

The result of this call is that you have a reference to the fifth element in **myOSVector**. ObjectStore has fetched only the contents of the **myOSVector** object, which includes references to the elements in the vector. It has not yet fetched the contents of any vector elements. When you try to access the contents of an element in the vector, ObjectStore fetches the data for that element. When you retrieve an element of an **OSVector** object, ObjectStore does not fetch all the elements of the **OSVector**. It fetches only the particular element you want to read or modify.

Roots with Null Values

It is possible to create a root with a null value. This is useful for creating roots in preparation for assigning objects to them later.

Using Primitive Values as Roots

If you want to store a primitive value as an independent persistent object, such as the value of a root, use an instance of a wrapper class, such as an **Integer**. For example:

```
db.createRoot("foo", new Integer(5));
```

This assigns the value **5** to the root named **foo**.

You cannot directly store primitive values in a database. You can define a primitive value as a field in a persistence-capable object.

Changing the Object Referred To by a Database Root

After you create a database root, you can change the object that it refers to. Inside an update transaction, call the **Database.setRoot()** method on the database that contains the root. You must specify an existing root and you can specify either a transient (but persistence-capable) or a persistent object. If you specify a transient object, ObjectStore immediately stores it in the default segment of the database. The default segment is the segment returned by the **Database.getDefaultSegment()** method. The method signature for changing the object associated with a root is

public void setRoot (String name, Object object)

If ObjectStore cannot find the specified root, DatabaseRootNotFoundException is thrown.

Destroying a Database Root

To destroy a database root, call the **destroyRoot()** method on the database that contains the root that you want to destroy. An update transaction must be in progress. Specify the name of the root. If ObjectStore cannot find the specified root, it throws DatabaseRootNotFoundException. The method signature is

public void destroyRoot (String name)

This has no effect on the referenced object, except that it is no longer accessible from that root. It might still be the value of another root, or it might be pointed to by some other persistent object. If a value of a root is no longer referenced after the root is destroyed, the object becomes unreachable. You can invoke **ObjectStore.destroy()** on it while you still have a reference to it. Alternatively, you can run the persistent garbage collector to remove all unreachable objects. See Performing Garbage Collection in a Database on page 81.

Destroying the Object Referred To by a Database Root

If you want to destroy the object that a database root refers to and you want to continue to use that database root, you must set the root to refer to null or another object before you destroy the object that the root refers to. If you do not do this and you try to use the root, ObjectStore throws ObjectNotFoundException. This is because the root refers to a destroyed object. For example, the correct sequence is something like this:

```
Object object = db.getRoot("username");
db.setRoot("username", null);
ObjectStore.destroy(object);
```

How Many Roots Are Needed in a Database?

It is important to realize that you need not create a root for most objects that you want to store in a database. You only need to create roots for top-level objects that you want to look up by name. You must have at least one root to be able to navigate through a database. Without a root, you have no way of accessing the objects in the database.

Think of a database root as the root of a tree. From the root, you can climb the tree. For many applications, a root is some kind of container, such as an instance of **OSHashtable** or **OSVector**, or an array. After you create one or more database roots, you create other objects that are referred to by fields of the objects that the roots refer to. These objects become persistent when you commit the transaction in which you create them. In a subsequent transaction, you can look up the root objects by name, and navigate from them to any other reachable persistent objects.

Too many roots can cause performance problems. The maximum practical number of roots within a database isapproximately 100. Databases store roots in a vector and ObjectStore uses a linear search to find roots. To avoid long look-up times, restrict the number of database roots.

Troubleshooting **OutOfMemoryError**

If you are storing many large objects, it might appear as though they are never garbage collected. In fact, you might receive the java.lang.OutOfMemoryError. Here is a discussion of why this might happen and what you can do.

When a transaction commits, ObjectStore releases references to all objects except those that are exported. ObjectStore uses JDK weak references to refer to exported objects. When an object is referred to by only weak references, it can be garbage collected. However, performing an explicit Java VM GC does not necessarily cause such weakly referenced objects to be collected and so free their space. Typically, the Java GC frees weakly referenced objects when it needs to grow the VM heap space. So, *eventually* you should see the storage for these objects being reclaimed.

Try running with the **-verbosegc** option to the Java VM. If the weakly referenced objects are *never* freed, it is likely to be for one of the following reasons:

- The application is inadvertently retaining references to these exported objects.
- The application is using a commit **retain** option other than **ObjectStore.RETAIN_STALE** or **ObjectStore.RETAIN_HOLLOW**.
- The application has disabled use of weak references when invoking the Java VM (**-DCOM.odi.disableWeakLinks=true**).
- The application is being run with an initial heap size set to the maximum heap size by the **-ms** or **-mx** option. There is a bug in the Java VM (JDK 1.1.4, due to be fixed in some future release) that causes premature java.lang.OutOfMemoryError errors to be signaled in some cases because of an incorrect limit computation in the JDK VM.

If you want the storage for a large object to be reclaimed immediately, wrap the large object in a small dummy object. Export the dummy object instead of the large object. This causes the small object to be retained as a hollow object that is referred to with a weak reference. Upon transaction commit with **ObjectSTore.RETAIN_STALE** or **ObjectStore.RETAIN.HOLLOW**, the large object is available to be garbage collected almost immediately.

Retrieving Persistent Objects

To read the contents of objects in a database, you must first obtain the value of an existing database root. Then the application makes the contents of the referenced object and any object it references accessible.

This section discusses the following topics:

Steps for Retrieving Persistent Objects

Follow these steps to retrieve a persistent object from a database:

1 Open the database.

2 Start a transaction. If you want to modify the object, start an update transaction.

3 Call the **Database.getRoot()** method on the database and specify the name of a previously created root.

ObjectStore returns a reference to the object that the root refers to. This assumes that the object you want is either the object the root refers to or reachable from that object.

4 Access the object just as you would access a transient object.

If you do not plan to run the postprocessor, see Chapter 9, Manually Generating Persistence-Capable Classes, Making Object Contents Accessible on page 294.

Obtaining a Database Root

Call the **Database.getRoot()** method to obtain a database root. When you obtain a database root, it returns a reference to its assigned object. You can use this reference to obtain a reference to any object that the assigned object references.

The signature for the **getRoot()** method is

```
public Object getRoot(String name)
```

Null return

It is possible for the **getRoot()** method to return a null value, which indicates that there is no object associated with the root. It does not mean that the root does not exist. If the root does not exist, ObjectStore throws DatabaseRootNotFoundException.

List of all roots

To obtain a list of the roots in a database, call the **getRoots()** method on the database. The signature of this method is

```
public DatabaseRootEnumeration getRoots()
```

Determining Which Database Contains an Object

You can use the **Database.of()** method to determine the database in which an object is stored. The method signature is

```
public static Database of(Object object)
```

If the specified object has been stored in a database, ObjectStore returns the database in which it is stored. The specified object must be a persistent primary Java object or a Java peer object. If you specify a Java peer object, it can identify a persistent or transient C++ object. If the specified object is a peer object for a transient object, the method returns the transient database.

Determining Whether an Object Has Been Stored

To determine whether an object has already been stored in a database, call the **ObjectStore.isPersistent()** method. The method signature is

```
public static boolean isPersistent(Object object)
```

If the specified object has been stored in a database, ObjectStore returns true. The specified object must not be a stale persistent object. If it is, ObjectStore throws ObjectException.

Iterating Through the Objects in a Segment

To obtain an enumeration of the objects in a segment, call the **Segment.getObjects()** method. This allows you to access any objects that are unreachable, but which have not yet been garbage collected. It also provides an application-independent means for processing all objects within a segment. The method signature is

public SegmentObjectEnumeration getObjects()

This method returns a **SegmentObjectEnumeration** object. After you have this object, you can use the following methods to iterate through the objects in the enumeration:

- **SegmentObjectEnumeration.nextElement()**
- **SegmentObjectEnumeration.hasMoreElements()**

The **Segment.getObjects()** method has an overloading that takes a **java.lang.Class** object as an argument and returns an iterator over all objects of that type in the database. The type can be an interface, class, or array type.

If your session or another session adds an object to a segment after you create an enumeration, the enumeration might or might not include the new object. If it is important for the enumeration to accurately include all objects, you should create the enumeration again.

After you create a **SegmentObjectEnumeration**, objects in the enumeration might get destroyed. When you use the enumeration to iterate through the objects, ObjectStore skips any destroyed objects. However, if you destroy a **COM.odi.coll** dictionary from C++, ObjectStore does not recognize that the dictionary has been destroyed. In this case, the enumeration might return a reference to garbage.

You can use a **SegmentObjectEnumeration** across transactions. If a transaction in which you use the **SegmentObjectEnumeration** aborts, the enumeration becomes stale.

After you create an enumeration of the objects in a segment, other sessions are blocked from destroying that segment until you end your transaction. If you create the enumeration and then destroy the segment, the next call to **nextElement()** or **hasMoreElements()** causes ObjectStore to throw SegmentNotFoundException.

There is a bug that makes the enumeration work incorrectly if you call **Transaction.checkpoint(RETAIN_STALE)**. Doing so causes the next use of the enumeration to throw ObjectException because of stale objects. This will be fixed in a future release.

Locking Objects

You can lock an object if you want to ensure that no other session can access it. This is useful when you want to ensure that a particular operation is not interrupted. See Locking Objects, Segments, and Databases to Ensure Access on page 328.

Using External References to Stored Objects

Outside a database, you might want to represent a reference to an object in a database. This external reference can be in an ASCII file or you might want to transmit it over a serial network connection. External references can be especially useful when you write a distributed application server that processes requests for many clients. This includes client / server applications that are based on Java RMI, ObjectStore ObjectForms, or the Object Management Group's Common Object Request Broker Architecture (CORBA).

ObjectStore provides the **ExternalReference** class to represent external references. To help you use external references, this section discusses

Creating External References

When you have a persistent object for which you want to create an external reference, follow these steps:

1 Create a new **ExternalReference** object by passing the persistent object to the constructor.

2 Obtain the database, segment ID, and location of the persistent object for which you just created an **ExternalReference** object. Use **ExternalReference** methods to do this.

3 In the format of your choice, store the values for the database, segment ID, and location wherever you want them to be.

For example, you can call the **Database.getPath()** method on the database reference and store a string pathname. Alternatively, you can look up the database object in a hash table, obtain an identifier, and store the identifier. Such an identifier is typically shorter than a pathname.

Example

For example:

```
ExternalReference myExRef = new ExternalReference(myPObj);
Database refToDb = myExRef.getDatabase();
int segId = myExRef.getSegmentId();
int loc = myExRef.getLocation();
```

An object represented by an external reference must be a persistent object or null. It can be a Java peer object only if it identifies a persistent C++ object. If you try to create an external reference to a peer object that identifies a transient C++ object, ObjectStore throws DatabaseException.

Requirements

If you want to create an external reference for an object that becomes persistent when the transaction commits, you must migrate the object to the database before you can create the external reference.

A database referred to in a call to an **ExternalReference** constructor or method must exist in the file system at the time of the call and must be open.

One-argument constructor

A transaction must be in progress to use the one-argument external reference constructor, whether or not the specified persistent object is null. The constructor signature is

```
public ExternalReference(Object o)
```

Three-argument constructor

An open transaction is not required for the three-argument constructor. This constructor signature is

public ExternalReference(Database d, int segid, int loc)

Cloning

The **ExternalReference** class implements the **Cloneable** interface. This allows you to call the **clone()** method on an **ExternalReference** object if you want to create another **ExternalReference** object with the same contents.

Using the No-Arguments Constructor

For convenience, the **ExternalReference** class also has a no-arguments constructor that sets the external reference to represent null. When an **ExternalReference** object represents null,

- **ExternalReference.getObject()** and **ExternalReference.getDatabase()** return null
- **ExternalReference.getSegmentId()** and **ExternalReference.getLocation()** return **-1**

When you try to call **ExternalReference.getObject()** or **ExternalReference.toString()** on an external reference object that was created with the no-arguments constructor, the object must contain all these values. If ObjectStore finds one of them but not the others, it throws IllegalArgumentException. You must be in a transaction when you use the no-arguments constructor.

Caution About Creating External References to Nonexported Objects

When you create an external reference to an object that is not exported, you must be aware of these possible situations:

- Your application might garbage collect the object.
- Your application might delete the object and then garbage collect the tombstone.
- ObjectStore reorganization of the segment might move the object.

If one of these situations occurs and you try to use the external reference, you receive incorrect results.

When an object is exported, ObjectStore uses its export ID in the external reference. This prevents the object or its tombstone from being garbage collected. It also allows ObjectStore to find the object if it is moved in a segment reorganization.

Obtaining Objects from External References

To obtain the object that an external reference refers to, you must do the following:

1 Set an **ExternalReference** object with the values of your external reference.

2 Use the **ExternalReference.getObject()** method to obtain the object.

Perform these steps while the database is open and a transaction is in progress.

To perform step 1, you can use the three-argument **ExternalReference** constructor:

```
public ExternalReference(Database d, int segid, int loc);
```

When you use this constructor, a transaction in progress is not required. For example:

```
ExternalReference anExRef = new ExternalReference(
    refToDb, segId, loc);
```

To obtain the corresponding object, call **ExternalReference.getObject()**. For example:

```
Object myPObj = myExRef.getObject();
```

Determining Whether Two External References Refer to the Same Object

Two external references are considered to be equal if they both refer to the same object. In other words, if you call **ExternalReference.getObject()** on each external reference, both calls return identical objects. Call the **ExternalReference.equals()** method to determine whether two external references refer to the same object. The method signature is

```
public boolean equals(Object obj)
```

Reusing External Reference Objects

After you create an **ExternalReference** object, you can reuse it any number of times. The advantage of reusing **ExternalReference** objects is that you avoid the overhead of storage allocation and garbage collection when you use large numbers of external references.

Storing external references

You can use an existing **ExternalReference** object to store an external reference. To do this, set the **ExternalReference** object to contain information about the persistent object for which you want an external reference. A transaction must be in progress. The method signature is

```
public void setObject(Object o)
```

For example, suppose **myExRef** was constructed for **myPObj**. Then you retrieved the database, segment ID, and location values and stored them in some file outside the database. Now you want to do the same thing for another persistent object, **herPObj**:

```
myExRef = ExternalReference.setObject(herPObj);
```

The **myExRef** external reference object now contains information for **herPObj**. You can use the **ExternalReference.get***xxx***()** methods to obtain the values you can store outside the database.

Obtaining externally referenced objects

You can also reuse an external reference object when you want to obtain a persistent object from an externally stored reference. Use the **ExternalReference.set***xxx***()** methods. The method signatures are

- **public void setDatabase(Database d);**
- **public void setSegmentId(int segid);**
- **public void setLocation(int loc);**

For example, suppose you have an external reference to **hisPObj**. You can reuse the **myExRef** object to obtain **hisPObj** by passing the stored values for **hisPObj** to the set methods on **myExRef**:

```
myExRef.setDatabase(refToDb);
myExRef.setSegmentId(segId);
myExRef.setLocation(loc);
hisPObj = myExRef.getObject();
```

Encoding External References as Strings

The **ExternalReference.toString()** and **ExternalReference.fromString()** methods act as a printer and parser, respectively. They allow you to encode an **ExternalReference** as a string, and then parse the string to rebuild an equivalent **ExternalReference**. This is convenient to use, but the strings are relatively long. They include the entire pathname of the database.

toString()

The **toString()** method is straightforward to use. You call the method on an **ExternalReference** object that contains values for a persistent object. The method signature is

```
public String toString()
```

fromString()

To use the **fromString()** method, pass it a string created by a call to **toString()**. This creates and returns a new **ExternalReference** object. The method signature is

```
public static ExternalReference fromString(String string)
```

Large numbers of external references

There might be situations when you print a large number of **ExternalReference** objects and you know that they are all from the same database. In this case it is more efficient to construct your own printed representation by using the **ExternalReference.getSegmentId()** and **ExternalReference.getLocation()** methods.

External References and Transactions

You must be in a transaction when you call these methods and constructors on the **ExternalReference** class:

- **ExternalReference()**
- **ExternalReference(object)**
- **fromString()**
- **getObject()**
- **setObject()** if you are not setting the object to null

You do not need to be in a transaction to call the other **ExternalReference** constructor and methods.

Updating Objects in the Database

To update objects in the database, start at a database root and traverse objects to locate the objects you want to modify. Make your modifications by updating fields or invoking methods on the object, just as you would operate on a transient object. Finally, save your changes by committing the transaction (this ends the transaction) or evicting the modified objects (this allows the transaction to remain active).

Whether you commit a transaction or evict an object, you can specify the state of objects after the operation. To specify the state that makes the most sense for your application, an understanding of the following background information is important:

- Background for Specifying Object State on page 147
- About Object Identity on page 148
- About the Object Table on page 152

Instructions for invoking **commit()** or **evict()** follow this background information.

Background for Specifying Object State

When a Java program accesses an object in an ObjectStore database, there are two copies of the object:

- The copy of the object in the database. This is the copy on the disk. This can be anything that is not a primitive. It can be a wrapper object.
- The copy of the object in your Java program. This is the copy that is referred to as a persistent object.

Normally, you need not be aware of the fact that there are two copies. Your application simply operates on the object in the Java program as if that is the only copy. This is why the documentation refers to this copy as a persistent object. However, the fact that there are two copies becomes apparent if a transaction aborts. In this case, the contents of the object in the database revert to the last committed copy.

About Object Identity

In a session, persistent objects maintain identity. Suppose there is an object in the database that is referred to by two different objects. You can reach the object in the database through two navigation paths. Regardless of which path you use, the resulting persistent object is the same object in the Java VM. In other words, if you have two unrelated objects (**a** and **b**), that refer to a third object (**c**), **a.c == b.c** is true.

In a single session, the Java VM never creates two distinct objects that both represent the same object in the database.

Sample class definitions

For example, suppose you have the following classes:

```
public class City {
    String name;
    int population;
}

public class State {
    City capital;
    String name;
    int population;
}
```

Creating objects

Suppose you also have the following code, which creates some instances of these classes and stores them:

```
City boston = new City("Boston", 1000000);
State massachusetts = new State(
    boston, "Massachusetts", 20000000);
OSHashtable cities = new OSHashtable();
cities.put("Boston", boston);
OSHashtable states = new OSHashtable();
states.put("Massachusetts", massachusetts);
db.createRoot("cities", cities);
db.createRoot("states", states);
```

This creates

- A **City** instance (Boston)
- A **State** instance (Massachusetts) with the Boston **City** instance as its capital
- Two instances of **OSHashtable** — one to hold **City** objects and one to hold **State** objects
- Two database roots — one to refer to each instance of **OSHashtable**

Accessing stored objects

Now you execute the following code to access the stored objects:

```
OSHashtable cities = (OSHashtable) db.getRoot("cities");
OSHashtable states = (OSHashtable) db.getRoot("states");
City boston1 = cities.get("Boston");
State massachusetts = states.get("Massachusetts");
City boston2 = massachusetts.capital;
if (boston1 == boston2)
   System.out.println("same");
else
   System.out.println("not the same");
```

Results

This code prints **"same"**. This is because **boston1** and **boston2**, even though they are located through different paths in the database, are still represented by the same object in the Java VM and therefore they are ==.

If you use **cities** to reach **boston1** and you modify **boston1**, you can then use **states** to access the updated version as **boston2**.

Alternative way to create objects

However, suppose that you change the way you create some instances of the **City** and **State** classes. Instead of doing it in one transaction, use two transactions. Create a new city called **Boston**, store it in the **cities** hash table, and commit the transaction. In a subsequent transaction, define a new state, **massachusetts**, and assign yet another new **Boston** object to its **capital** field. Store **massachusetts** in the **states** hash table and commit the transaction. Here is the code that does this:

```
Transaction tr = Transaction.begin(ObjectStore.UPDATE);
City Boston = new City("Boston", 1000000);
OSHashtable cities = new OSHashtable();
cities.put("Boston", Boston);
db.createRoot("cities", cities);
tr.commit();
```

```
Transaction tr = Transaction.begin(ObjectStore.UPDATE);
City Boston = new City("Boston", 1000000);
State massachusetts = new State(
   Boston, "Massachusetts", 20000000);
OSHashtable states = new OSHashtable();
states.put("Massachusetts", massachusetts);
db.createRoot("states", states);
tr.commit();
```

Results of alternative approach

Now there are two different **Boston** objects in the database. One is accessible through the **cities** root and another one is accessible through the **states** root. Although both objects are instances of **City** and both are named **Boston**, they are two different objects because they were stored as two different objects. When you create objects with separate calls to **new**, the objects can never be the same in the sense of **==**.

If you follow the two different paths through the database to the two different **Boston** objects, you obtain two discrete persistent objects. If you update one **Boston** object, ObjectStore does not update the other **Boston** object. If you execute the same code fragment as you did for the first code sample, the result is **"not the same"**. That is, **(boston1 == boston2)** returns false.

Strings and primitive wrappers

There are additional considerations for **String**s and primitive wrapper classes.

String pooling causes some strings to be the same, even when you create them separately. If you call **new** multiple times to create multiple **String** objects, these separately created objects might be identical. See Description of COM.odi.stringPoolSize. If you explicitly migrate the string to the database, it prevents ObjectStore from using string pooling.

A **String** or primitive wrapper object that you create with a single call to **new** might be represented by more than one persistent object. Usually, this does not matter for **Sting**s and primitive wrapper objects because it is their value and not their identity that matters. If identity does matter, you can explicitly migrate wrapper objects into the database.

Identity across transactions

ObjectStore maintains the identity of referenced objects across transactions within the same session. The following code fragment provides an example of this. It displays **"same"**.

```
public
class Person {
    // Fields in the Person class:
    String name;
    int age;
    Person children[];
    Person father;
    // Constructor:
    public Person(String name, int age,
        Person children[], Person father) {
        this.name = name;
        this.age = age;
        this.children = children;
        this.father = father;
    }

    static void testIdentity() {
    // Omit open database calls

        Transaction tr = Transaction.begin(ObjectStore.UPDATE);
        Person children[] = { null, null };
        Person tim = new Person("Tim", 35, children, null);
        Person sophie = new Person("Sophie", 5, null, tim);
        children[0] = sophie;
        db.createRoot("Tim", tim);
        tr.commit();

        Transaction tr = Transaction.begin(ObjectStore.UPDATE);
        tim = (Person)db.getRoot("Tim");
        Person joseph = new Person("Joseph", 1, null, tim);
        tim.getChildren()[1] = joseph;
        tr.commit();

        Transaction tr = Transaction.begin(ObjectStore.READONLY);
        tim = (Person)db.getRoot("Tim");
        sophie = tim.getChildren()[0];
        joseph = tim.getChildren()[1];
        if (sophie.getFather() == joseph.getFather())
            System.out.println("same");
        else
            System.out.println("not the same");
        tr.commit();
    }
```

About the Object Table

ObjectStore keeps a table of all objects referenced in a transaction. If you refer to the same object in the database twice (perhaps accessing the object through different paths), there is only one copy of the object in your Java program. Remember, if you retrieve the same object through different paths, **==** returns true because ObjectStore preserves object identity.

If the system property **COM.odi.disableWeakReferences** is set to false (the default), the references in the object table are *weak references*, which means that they do not interfere with the Java garbage collector. The Java garbage collector can function in the same way that it normally does. That is, if a Java program does not have any references to a persistent object (the copy in your Java program), other than through the ObjectStore object table, the object can be garbage collected. (The object in the database, of course, is not garbage collected.)

Committing Transactions to Save Modifications

When you commit a transaction, ObjectStore

- Saves and commits any modifications in the database.
- Checks for transient objects that are referred to by persistent objects.

  If there are such objects, ObjectStore stores them in the database if they are persistence-capable. This is called transitive persistence. All reachable persistence-capable objects become persistent through transitive persistence.

  If the modifications contain references to objects that are not persistence-capable, ObjectStore throws AbortException. The **AbortException.getOriginalException()** method returns the object that causes the exception.

- Sets the state of persistent objects after the transaction.

  If objects were stored in the database for the first time during this transaction, the copies of these objects in your Java program are included in the group of persistent objects.

  The default is that persistent objects are stale after the transaction. If you do not want them to be stale, specify a **retain** argument when you invoke **commit()**.

Method signatures

The **commit()** method has two overloadings. The first overloading takes no argument. The method signature is

```
public void commit()
```

The second overloading has an argument that specifies the state of persistent objects after the commit operation. The method signature is

```
public void commit(int retain)
```

Contents

This section discusses the object states in the following topics:

Exceptions

If you try to commit a transaction when an object in one segment refers to an unexported object in another segment, ObjectStore throws AbortException and aborts the transaction.

Incorrect access to persistent objects

If your application commits a transaction while an annotated method is executing, your program might incorrectly access additional persistent objects after the commit. For more information about this and a work around, see Troubleshooting Access to Persistent Objects on page 183.

Synchronization

Synchronizing on a persistent object is another way to retain a reference to that object after a transaction ends. To do this, an application must end a transaction with a retain type other than stale. When objects become stale, ObjectStore does not maintain transient object identity. The synchronized state of an object is not saved persistently.

Making Persistent Objects Stale

When you call the **commit()** method with no argument, ObjectStore makes all persistent objects stale. Stale persistent objects are not accessible and their contents are set to default values. ObjectStore reclaims the entry in the Object Table for the stale object and the object loses its persistent identity.

If your Java program still has references to stale objects, any attempt to use those references (such as by accessing a field or calling a method on the object) causes ObjectStore to throw ObjectException. Therefore, your application must discard any references to persistent objects when it calls this overloading of **commit()**.

Objects available for garbage collection

This overloading of **commit()** also discards any internal ObjectStore references to the copies of the objects in your Java program. When your application makes an object stale, ObjectStore makes any references from the stale object to other objects null. This makes the referenced objects, which can be persistent or transient, available for garbage collection if there are no other references to them from other objects.

Stale persistent objects are not available for Java garbage collection if your Java application has transient references to them.

Accessing objects again

You can reaccess the same objects in the database in subsequent transactions. To do so, look up a database root and traverse to objects from there, or reference them through hollow objects. ObjectStore refetches the contents of the object and creates a new active persistent object. The new object has a new transient identity and the same persistent identity as the object that became stale. For example:

```
Foo foo = myDB.getRoot("A_FOO");
    ExternalReference fooRef = new ExternalReference(foo);
    ObjectStore.evict(foo, ObjectStore.RETAIN_STALE);
    Foo fooTwo = myDB.getRoot("A_FOO");  // refetch from database
    ExternalReference fooRefTwo = new ExternalReference(fooTwo);
    // At this point (foo == fooTwo) returns false,
    // but  (fooRef.equals(fooTwoRef)) returns true.
```

Advantage

The advantage of using **commit()** with no argument is that it wipes your database cache clean, and typically makes all transient copies of persistent data available for Java garbage collection.

Disadvantage

The disadvantage is that any references to these objects that your Java program holds become unusable.

Alternative method

Invoking **commit(ObjectStore.RETAIN_STALE)** is the same as calling **commit()** with no argument.

Making Persistent Objects Hollow

Call **commit(ObjectStore.RETAIN_HOLLOW)** to make persistent objects (the copies of the objects in your Java program) hollow. ObjectStore resets the contents of persistent objects to default values.

References to these objects remain valid; the application can use them in a subsequent transaction. If a hollow object is accessed in a subsequent transaction, ObjectStore refreshes the contents of the object in your Java program with the contents of the corresponding object in the database.

Outside transaction

An application cannot access hollow objects outside a transaction. An attempt to do so causes ObjectStore to throw NoTransactionInProgressException.

Advantage

The advantage of invoking **commit(ObjectStore.RETAIN_HOLLOW)** is that any references to persistent objects that the Java application holds remain valid in subsequent transactions. This means that it is not necessary to renavigate to these objects from a database root.

Garbage collection

Sometimes an application might retain a reference to an object and prevent Java garbage collection that would otherwise occur. It is good practice to avoid retaining references to objects unnecessarily.

Scope

If you commit a transaction with **ObjectStore.RETAIN_HOLLOW**, and then commit a subsequent transaction with no **retain** argument or **ObjectStore.RETAIN_STALE**, this cancels the previous **ObjectStore.RETAIN_HOLLOW** specification. No object references are available in the next transaction. This is true regardless of whether or not they were previously retained.

Retaining Persistent Objects as Readable

Call **commit(ObjectStore.RETAIN_READONLY)** to retain the copies of the objects in your Java program as readable persistent objects. ObjectStore maintains the contents of the persistent objects that the application read in the transaction just committed. The contents of these persistent objects are as they were the last time the objects were read or modified in the transaction just committed.

If there are any hollow objects when you commit the transaction, ObjectStore retains these objects as hollow objects that you can use during the next transaction.

After this transaction and before the next transaction, your application can read the contents of any retained objects whose contents were also retained. The actual contents of the object in the database might be different because some other process modified it. Your application cannot modify these objects. An attempt to do so causes ObjectStore to throw NoTransactionInProgressException. Your application cannot access the contents of hollow retained objects. An attempt to do so causes ObjectStore to throw NoTransactionInProgressException.

Scope

If you commit a transaction with **ObjectStore.RETAIN_READONLY**, the contents of only those persistent objects whose contents were accessed in the transaction just committed are available to you after the transaction. This is because ObjectStore makes all retained objects hollow at the start of the next transaction. Any cached references to persistent objects remain valid. In the new transaction, ObjectStore fetches the contents of a persistent object when your application requires it.

Advantage

The advantage of using **commit(ObjectStore.RETAIN_READONLY)** is that the copies of the persistent objects in your Java program remain accessible after the transaction is over. In the next transaction, any cached references to persistent objects remain valid. ObjectStore copies the object's contents from the database when you access the object.

Disadvantage

The disadvantage of using **commit(ObjectStore.RETAIN_READONLY)** is that it makes more work for the Java garbage collector, because the contents of the copies of the objects in your Java program are not cleared.

Your program might return results that are inconsistent with the current state of the database.

ObjectStore cannot fetch any objects outside a transaction. This makes it difficult to ensure that methods can execute without throwing an exception. However, you can call **ObjectStore.deepFetch()** in the transaction to obtain the contents of all objects you might need. Of course, this increases the risk of the Java VM running out of memory.

Serialization

If you are using Java Remote Method Interface (RMI) or serialization, you can call the **ObjectStore.deepFetch()** method followed by **commit(ObjectStore.RETAIN_READONLY)**. This allows you to perform object serialization outside a transaction.

Retaining collections not allowed

Peer objects, and therefore collection objects, have no data members and so, in Java, peer objects do not appear to be connected to other objects. Even if you explicitly iterate through the elements in a collection and then commit the transaction with **ObjectStore.RETAIN_READONLY**, you cannot access the collection outside a transaction. However, if the collection elements are not themselves peer objects or collections, you can manipulate them outside a transaction, but you cannot use the collection to access them. You must explicitly read them in the transaction, retain them, and then access them directly or through another nonpeer object.

Troubleshooting: NoTransactionInProgressException

Between transactions, you might try to read an object that you thought you retained, and receive NoTransactionInProgressException. Often, the cause of this is that you retained a reference to the object but not the contents of the object.

In a transaction, you might read the contents of an object, but not the contents of objects the first object refers to. For example, during a transaction, suppose you access a vector but not any of the elements in the vector. When you commit the transaction, the contents of vector elements are not available in the transaction and they are not retained. In other words, to be able to read the contents of an object between transactions, you must read that particular object during the previous transaction.

To be able to read objects between transactions, you might want to call **ObjectStore.deepFetch()** on an object. This method fetches

the contents of the specified object, the contents of any objects that object refers to, the contents of any objects those objects refer to, and so on for all reachable objects.

Inside a transaction, ObjectStore automatically fetches the contents of objects as you read the objects. Outside a transaction, if a reference to an object, but not the contents of the object, was retained, ObjectStore throws NoTransactionInProgressException.

Here is another situation in which you would receive the NoTransactionInProgressException:

1 In a transaction, you read object **A**.

2 You commit the transaction with **ObjectStore.RETAIN_READONLY**.

3 You start a new transaction. It does not matter whether or not you access object **A** in this transaction.

4 You commit this transaction with **ObjectStore.RETAIN_STALE** or without a **retain** argument.

5 Outside a transaction, you try to access object **A** and you receive the NoTransactionInProgressException.

You might think that because you retained **A** after a previous transaction, its contents are still available. This is not the case. Since nothing was retained after the second transaction, the contents of **A** are no longer available.

MVCC alternative

Many applications that seem to be suitable for retaining persistent objects as readable can be better implemented with the Multiversion Concurrency Control (MVCC) feature. See page 317.

Retaining Persistent Objects as Writable

Call **commit(ObjectStore.RETAIN_UPDATE)** to retain the copies of the objects in your Java program as readable and writable. ObjectStore maintains the contents of the persistent objects as they are at the end of the transaction.

Sometimes, the contents of an object are not available in a transaction when you expect that they are available. See Troubleshooting: NoTransactionIn ProgressException on page 158.

A specification of **ObjectStore.RETAIN_UPDATE** is exactly like a specification of **ObjectStore.RETAIN_READONLY**, except that if your application accesses the objects after the transaction and before the next transaction, it can modify as well as read the objects. At the beginning of the next transaction, ObjectStore discards any updates made to the persistent objects between transactions. ObjectStore does not modify the contents of any transient-only fields unless you have explicitly defined one of the following methods to modify transient-only fields.

- **IPersistent.clearContents()**
- **IPersistent.preClearContents()**
- **IPersistent.initializeContents()**
- **IPersistent.postInitializeContents()**

The **clearContents()** and **initializeContents()** methods that are generated by the postprocessor do not modify transient-only fields.

Advantage

The advantage of using **commit(ObjectStore.RETAIN_UPDATE)** is that the copies of the objects in your Java program become scratch space between transactions. You can use them to determine what the results might be for a particular scenario.

Disadvantage

The disadvantage is that updates are automatically discarded at the beginning of the next transaction. This can make it difficult to debug applications that use this option indiscriminantly.

Caution About Retaining Nonexported Objects

Suppose you commit a transaction and retain persistent objects as hollow, readable, or writable. Retained objects that are not exported can become stale inside a subsequent transaction. This can happen if you run the schema evolution tool or use the schema evolution API. If you try to access a retained object that has become stale after schema evolution, ObjectStore throws UnexportedObjectsBecameStaleException.

When retained objects are exported, ObjectStore can correctly retain them after schema evolution.

Consequently, you must follow at least one of these rules:

- Do not retain nonexported objects between transactions.
- Ensure than schema evolution does not operate on your database while your application is running.
- Catch the UnexportedObjectsBecameStaleException, abort the transaction, and retry the transaction. This is similar to handling DeadlockException, except that UnexportedObjectsBecameStaleException does not abort the transaction.
- Catch the exception, retrieve objects from roots again, and retry the operation. You do not need to abort the transaction.

Evicting Objects to Save Modifications

You might want to save modifications to an object or change the state of an object without committing a transaction. The **evict()** method allows you to do this.

Method signatures

The **evict()** method has two overloadings. The first overloading takes an object as an argument. The method signature is

public static void evict(Object object)

The second overloading has an additional argument that specifies the state of the evicted object after the eviction. The method signature is

public static void evict(Object object, int retain)

Contents

This section provides the following information about evicting objects:

Description of Eviction Operation

When you evict an object, ObjectStore

- Saves any modifications to the object in the database, but does not commit the changes. If the transaction commits, then any changes are committed. If the transaction aborts, the contents of the object in the database revert to the contents following the last committed transaction in which the object was modified.
- Sets the state of the evicted object after the eviction. This affects the copy of the object that is in your Java program. The default is that the evicted object is stale after the eviction. If you do not want it to be stale, you can specify another state when you invoke **evict()**.

References to other objects

When you evict an object, ObjectStore does not evict objects that the evicted object references.

You might evict an object that has instance variables that are transient strings. ObjectStore migrates such strings to the database and stores them in the same segment as the evicted object. As part of the eviction process, ObjectStore evicts the just-stored string with a specification of **ObjectStore.RETAIN_READONLY**. Consequently, after the eviction, the migrated string remains readable.

If you try to evict an object when that object refers to a transient object that is not persistence-capable, ObjectStore throws ObjectNotPersistenceCapableException and does not perform the eviction. The exception message provides the class of the object that is causing the problem.

If you try to evict an object when the object refers to an unexported object in another segment, ObjectStore throws ObjectNotExportedException and cancels the evict operation.

Caution

If your application evicts one or more objects while an annotated method is executing, your program might incorrectly access persistent objects after the eviction. For more information about this and a work around, see Troubleshooting Access to Persistent Objects on page 183.

Setting the Evicted Object to Be Stale

When you invoke **evict(Object)**, ObjectStore makes the evicted object stale. ObjectStore resets the contents of the copy of the object in your Java program to default values and makes the object inaccessible. Any references to the evicted object are stale, and your application should discard them. The copy of the object in your Java program becomes available for Java garbage collection.

Advantage

The advantage of using the **evict(Object)** method is that the evicted object and all objects it refers to become available for Java garbage collection (provided that they are not referenced by other objects in the Java program).

Disadvantage

The disadvantage is that any references to the evicted object become stale. If you try to use the stale references, ObjectStore throws ObjectException.

The effect on accessibility to the copy of the object in your Java program is therefore similar to the effect of **commit()** and **commit(ObjectStore.RETAIN_STALE)**. However, if the transaction aborts, any changes to the evicted object are discarded.

Alternative method

A call to **evict(Object, ObjectStore.RETAIN_STALE)** is identical to a call to **evict(Object)**.

Setting the Evicted Object to Be Hollow

When you invoke **evict(Object, ObjectStore.RETAIN_HOLLOW),** ObjectStore makes the evicted object hollow. ObjectStore resets the contents of the copy of the object in your Java program to default values. References to the evicted object continue to be valid.

If the application accesses the evicted object in the same transaction, ObjectStore copies the contents of the object from the database to the copy in your Java program. If your application modified the object before evicting it, these modifications are included in the new copy in your Java program.

Advantage

The reason to use the **evict(Object, ObjectStore.RETAIN_HOLLOW)** method is that the object and all objects it refers to become available for Java garbage collection (provided that they are not referenced from other objects in the Java program).

Sometimes an application might retain references to an object and prevent Java garbage collection that would otherwise occur. It is good practice to avoid retaining references to objects unnecessarily.

Setting the Evicted Object to Remain Active

When you invoke **evict(Object, ObjectStore.RETAIN_READONLY),** ObjectStore

- Retains references to the evicted object.
- Retains the contents of the evicted object.
- Saves any changes to the evicted object.
- Internally flags the object as read but not modified. This is because any changes are already saved. If the application later modifies the evicted object in the same transaction, ObjectStore modifies this flag accordingly.

Garbage collection

Any changes that were made before the object was evicted are saved in the database. (Of course, if the transaction aborts, the changes are rolled back.) Therefore, if the evicted object is not referenced by other objects in the Java program, it becomes available for Java garbage collection.

Additional changes

Your application can read or modify the evicted object in the same transaction. If it does, ObjectStore does not have to recopy the contents of the object from the database to your program. When the application commits the transaction or evicts the object again, ObjectStore saves in the database any new changes to the evicted object.

It might seem strange to evict an object with **ObjectStore.RETAIN_READONLY** and yet be able to modify the object after the eviction. The specification of **READONLY** in this context means that, as of this point in time, the evicted object has been read but not modified. The changes have already been saved, but not committed. The contents are still available and can be read or updated.

Advantage

The advantage of using **evict(Object, ObjectStore.RETAIN_READONLY)** is that the updated object becomes available for Java garbage collection.

Summary of Eviction Results for Various Object States

The following table shows the results of an eviction according to the value specified for the **retain** argument.

Results of Eviction	**RETAIN_STALE**	**RETAIN_HOLLOW**	**RETAIN_READONLY**
Object state	Stale	Hollow	Active
References to evicted object	Stale	Remain valid	Remain valid
Candidate for Java garbage collection	Candidate	Can be candidate	Not a candidate

Evicting All Persistent Objects

You can evict all persistent objects with one call to **evictAll()**. The method signature is

public static void evictAll(int retain)

For the **retain** argument, you can specify

- **ObjectStore.RETAIN_STALE**
- **ObjectStore.RETAIN_HOLLOW**
- **ObjectStore.RETAIN_READONLY**

If you specify **RETAIN_STALE** or **RETAIN_HOLLOW**, ObjectStore applies it to all persistent objects that belong to the same session as the active thread. It does it in the same way that it applies it to one object for the **evict(object, retain)** method. If you specify **RETAIN_READONLY**, ObjectStore applies it to all active persistent objects that belong to the same session as the active thread.

Evicting Objects When There Are Cooperating Threads

Before an application evicts an object, it must ensure that no other thread requires that object to be accessible. For example, suppose you have code like this:

```
class C {
    String x;
    String y;

    void function() {
        System.out.println(x);
        ObjectStore.evict(this);
        System.out.println(y);
    }
}
```

Before the first call to **println()**, the object is accessible. After the call to **evict()**, the **y** field is null and the second **println()** call fails. There are more complicated scenarios for this problem, which involve subroutines that call **evict()** and cause problems in the calling functions. This problem can occur in a single thread. If there are multiple cooperating threads, each thread must recognize what the other thread is doing. See Cooperating Threads on page 48.

It is the responsibility of the application to ensure that the object being evicted is not the **this** argument of any method that is currently executing.

Committing Transactions After Evicting Objects

In a transaction, you might evict some objects and specify their state to be hollow or active. If you then commit the transaction and cause the state of persistent objects to be stale, this overrides the hollow or active state set by the eviction. If you commit the transaction and cause the state of persistent objects to be hollow, this overrides an active state set by eviction. For example:

```
Transaction tr = Transaction.begin(ObjectStore.UPDATE);
Trail trail = (Trail) db.getRoot("greenleaf");
GuideBook guideBook = trail.getDescription();
ObjectStore.evict(guideBook, ObjectStore.RETAIN_READONLY);
tr.commit();
```

After the transaction commits, the application cannot use **guideBook**. Committing the transaction without specifying a **retain** argument makes all persistent objects stale. This overrides the **RETAIN_READONLY** specification when **guideBook** was evicted.

Evicting Objects Outside a Transaction

Outside a transaction, eviction of an object has meaning only if you retained objects when you committed the previous transaction. In other words, if you invoke the **commit(retain)** method and specify a value for the **retain** argument other than **RETAIN_STALE**, you can evict retained objects outside a transaction.

If you specified **commit(ObjectStore.RETAIN_STALE),** there are no objects to evict after the transaction commits.

If you invoked **commit()** with any other **retain** value, you can call **evict()** or **evictAll()** with the value of the **retain** argument as **RETAIN_STALE** or **RETAIN_HOLLOW**. If you specify **RETAIN_READONLY**, ObjectStore does nothing.

Outside a transaction, if you make any changes to the objects you evict, ObjectStore discards these changes at the start of the next transaction. They are not saved in the database.

Aborting Transactions to Cancel Changes

If you modify some objects and then decide that you do not want to keep the changes, you can abort the transaction. Aborting a transaction

- Ensures that the objects in the database are as they were just before the aborted transaction started.
- Sets the state of persistent objects from the transaction.

Only the state of the database is rolled back. The state of transient objects is not undone automatically. Applications are responsible for undoing the state of transient objects. Any form of output that occurred before the abort cannot be undone.

This section discusses the following topics:

- Setting Persistent Objects to the Default State on page 171
- Setting the Default Abort Retain State on page 171
- Specifying a Particular State for Persistent Objects on page 171

Caution

If your application aborts a transaction while an annotated method is executing, your program might incorrectly access additional persistent objects after the abort operation. For more information about this and a work around, see Troubleshooting Access to Persistent Objects on page 183.

Setting Persistent Objects to the Default State

To abort a transaction and set the state of persistent objects to the state specified by **Transaction.setDefaultAbortRetain()**, call the **abort()** method. The default state is stale. The method signature is

```
public void abort()
```

For example:

```
tr.abort();
```

Setting the Default Abort Retain State

Call the **setDefaultAbortRetain()** method to set the default state for persistent objects after a transaction is aborted. The method signature is

```
public void setDefaultAbortRetain(int newRetain)
```

The values you can specify for **newRetain** are the same values you can specify when you call **abort()** with a **retain** argument. These values are described in the next section.

Specifying a Particular State for Persistent Objects

To abort a transaction and specify a particular state for persistent objects after the transaction, call the **abort(retain)** method on the transaction. The method signature is

```
public void abort(int retain)
```

The following example aborts the transaction and specifies that the contents of the active persistent objects should remain available to be read:

```
tr.abort(ObjectStore.RETAIN_READONLY);
```

The values you can specify for **retain** are described below. The rules for Java garbage collection of objects retained from aborted transactions are the same as for objects retained from committed transactions. See Committing Transactions to Save Modifications on page 153.

RETAIN_STALE

ObjectStore.RETAIN_STALE resets the contents of all persistent objects to their default values and makes them stale. This is the same as calling **abort()** when **Transaction.setDefaultAbortRetain()** has either not been called or been called with **ObjectStore.RETAIN_STALE** as its argument.

RETAIN_HOLLOW

ObjectStore.RETAIN_HOLLOW resets the contents of all persistent objects to their default values and makes them hollow. In the next transaction, you can use references to persistent objects from this transaction.

RETAIN_READONLY

ObjectStore.RETAIN_READONLY retains the contents of unmodified persistent objects that were read during the aborted transaction. Any objects that were modified become hollow objects, as if **ObjectStore.RETAIN_HOLLOW** had been specified. Objects whose contents were read but not modified in the aborted transaction can be read after the aborted transaction.

If you try to modify a persistent object before the next transaction, ObjectStore throws NoTransactionInProgressException. If you modified any persistent objects during the aborted transaction, ObjectStore discards these modifications and makes these objects hollow as part of the abort operation.

During the next transaction, the contents of persistent objects that were not modified during the aborted transaction are still available.

RETAIN_UPDATE

ObjectStore.RETAIN_UPDATE retains the contents of persistent objects that were read or modified during the aborted transaction. The values that are retained are the last values that the objects contained before the transaction was aborted. Even though the changes to the modified objects are undone with regard to the database, the changes are not undone in the objects in the Java VM.

While you are between transactions, the changes that were aborted are still visible in the Java objects. At the start of the next transaction, ObjectStore discards the modifications and reads in the contents from the database. Objects that were read or modified in the aborted transaction can be modified between the aborted transaction and the next transaction. If you modify any persistent objects during or after the aborted transaction, ObjectStore discards these modifications and makes these object hollow at the start of the next transaction.

During the next transaction, the contents of persistent objects that were not modified during or after the aborted transaction are still available.

Destroying Objects in the Database

You can explicitly destroy any object that you want to be deleted from persistent storage. You can also perform persistent garbage collection, which destroys unreachable objects in the database. See Performing Garbage Collection in a Database on page 81. The discussion of the destroy operation covers the following topics:

- Calling ObjectStore.destroy() on page 173
- Destroying Objects That Refer to Other Objects on page 174
- Destroying Objects That Are Referred to by Other Objects on page 178

Calling ObjectStore.destroy()

To destroy an object, call **ObjectStore.destroy()**. The method signature is

public static void destroy(Object object)

The object you specify must be persistent or the call has no effect. The database that contains the object must be open-for-update and an update transaction must be in progress.

If the destroyed object either implements the **IPersistent** interface or is an array, you cannot access any of its fields after you destroy it.

After you invoke **ObjectStore.destroy()** on a primary Java object, ObjectStore leaves a tombstone. If you try to access the destroyed object, the tombstone causes ObjectStore to throw ObjectNotFoundException.

Destroying Objects That Refer to Other Objects

By default, when you destroy an object, ObjectStore does not destroy objects that the destroyed object references.

There is a hook method, **IPersistent.preDestroyPersistent()**, that you can define. ObjectStore calls this method before actually destroying the specified object. This method is useful when an object has underlying structures that you want to destroy along with the object. The default implementation of this method does nothing.

You can use **preDestroyPersistent()** to propagate the destroy operation to child objects that are referenced by the one being destroyed. If you do this, be careful that the child objects themselves are not referenced by other objects in the database. If an object attempts to use a reference to an explicitly destroyed object, ObjectStore throws ObjectNotFoundException. If you are not certain whether a specific object might be referenced elsewhere, it is better to avoid explicitly destroying the object. Let the persistent garbage collector do the job instead.

OSHashtable and **OSVector**

When you delete a **COM.odi.util.OSHashtable** or **COM.odi.util.OSVector** object, ObjectStore deletes the hash table or vector and its own internal data structures. ObjectStore does not delete the keys or elements that were inserted into the hash table or vector. Doing so might cause problems because other Java objects might refer to those objects.

However, sometimes you want to destroy the objects in a hash table or vector as well as the hash table or vector itself. Suppose you have a class in which one of the instance variables is a **COM.odi.util.OSVector**. You might want to ensure that whenever an instance of this class is destroyed, the **OSVector** and its contents are also destroyed. To do this, you can define a **preDestroyPersistent()** method on your class. Define this method to iterate over the elements in the vector, destroy each one, and then destroy the **COM.odi.util.OSVector**.

Types not destroyed

When you call the **ObjectStore.destroy()** method on an object, it does not destroy fields in the object that are

- **String** types
- Instances of wrapper classes that have been explicitly migrated with the **ObjectStore.migrate()** method

For additional information about ObjectStore treatment of **String** instances, see Description of Special Behavior of String Literals on page 366. For example, if you define a class such as the one below, when you destroy an instance of this class, you should also explicitly destroy **s** and **d**.

```
class C {
   int i;
   String s;
   Double d;
}
```

Advantages of explicit destroy

You should always consider whether or not to have **preDestroyPersistent()** call **ObjectStore.destroy()** on fields that contain **String** types, instance of wrapper classes that have been explicitly migrated, or types that you define. The advantages of explicitly destroying objects are

- ObjectStore replaces large objects or arrays with a four-byte tombstone.
- Space is freed without having to wait for the persistent garbage collector to run.
- Without expanding the size of the database, the length of time between persistent garbage collections can be longer.

Disadvantages of explicit destroy

The disadvantages of explicitly destroying such objects are

- You must write additional code.
- There is the risk of a dangling reference if you are not careful. For example, an unanticipated ObjectException might prevent an object from being destroyed.
- ObjectStore replaces a destroyed object with a tombstone that uses four bytes. This can cause fragmentation. The tombstone can also cause ObjectStore to throw ObjectNotFoundException. For example, suppose you unintentionally destroy an object that is referenced by another object. When you try to dereference the reference to the destroyed object, the tombstone causes ObjectStore to throw ObjectNotFoundException.

If you do not explicitly destroy an unreferenced object, ObjectStore destroys it when you run the persistent garbage collector.

You do not need to have **preDestroyPersistent()** call **ObjectStore.destroy()** on fields that contain primitive types.

Example

For example, suppose you have a persistence-capable class called **MyVector** that has a private field called **contents**. When an instance of **MyVector** is persistent, the **contents** field is also persistent, but a user would not have access to it because it is private. If a user calls **ObjectStore.destroy()** on an instance of **MyVector**, the operation destroys the instance but not the **contents** object.

If you are the programmer implementing the **MyVector** class, you have two choices:

- Provide a **MyVector.***destroy***()** method to call **ObjectStore.destroy(contents).** If you do this, you must ensure that users of **MyVector** understand that they should not call **ObjectStore.destroy()** on an instance of **MyVector** because doing so leaves garbage in the database.
- Provide a **preDestroyPersistent()** method that calls **ObjectStore.destroy(contents)**. This choice ensures that if a user calls **ObjectStore.destroy()** on an instance of **MyVector**, the operation cleans up the private **contents** field.

Here is code that shows the second alternative:

```
public class MyVector {
    private Object[] contents;

    public addElement(Object o) {
        contents[nextElement++] = o;
    }

    public void preDestroyPersistent() {
        if (contents != null)
            ObjectStore.destroy(contents);
    }
}
```

Destroying Objects That Are Referred to by Other Objects

The usual practice is to remove references to a persistent object before you destroy that persistent object. ObjectStore throws ObjectNotFoundException when you try to access a destroyed object. It is up to you to clean up any references to destroyed objects.

If an object retains a reference to a destroyed object, ObjectStore throws ObjectNotFoundException when you try to use that reference. This might occur long after the referenced object was destroyed. To clean up this situation, set the reference in the referring object to null.

String class

A call to **destroy** on a **String** object has different behavior. When you dereference a reference to such a destroyed object, ObjectStore does not throw ObjectNotFoundException. Instead, references to the destroyed object from objects modified in the same transaction as the destroy operation continue to have the value of the destroyed object. References to the destroyed object from objects not modified in the same transaction appear as null values when an object containing such a reference is fetched.

Hash tables

You should avoid having a hash table refer to a destroyed object. It is difficult to remove a reference from a hash table after you destroy the object that it refers to. This is because the search through the hash table for the referring object might cause ObjectStore to try to access the destroyed object. In fact, a search for another object in the hash table might cause ObjectStore to access the destroyed object. The result is that the hash table lookup procedure throws ObjectNotFoundException and the hash table becomes useless. Consequently, you should always remove objects from hash tables before you destroy them.

Default Effects of Various Methods on Object State

The table below summarizes the default effects of various methods on the state of hollow or active persistent objects. You should never try to invoke a method on a stale object. If you do, ObjectStore tries to detect it and throw ObjectException. ObjectStore can throw ObjectException for objects that are instances of classes that implement the **IPersistent** interface.

Unless you manually annotate your classes to make them persistence-capable, you do not write **ObjectStore.fetch()** or **ObjectStore.dirty()** calls in your application. The postprocessor inserts these calls automatically as needed.

The information in this table assumes that you are not specifying a **retain** argument with any of the methods that accept a retain argument.

Method the Application Calls	***Result When Invoked on a Hollow or Active Object***
ObjectStore.fetch()	Active persistent object
ObjectStore.dirty()	Active persistent object
ObjectStore.evict()	Hollow persistent object
ObjectStore.destroy()	Stale persistent object

Method the Application Calls	***Result***
Transaction.commit()	Persistent objects become stale.
Transaction.abort()	Persistent objects become stale.

Transient Fields in Persistence-Capable Classes

This section discusses

See also Creating Persistence-Capable Classes with Transient Fields on page 273.

Behavior of Transient Fields

In a persistence-capable class, a field designated with the **transient** keyword behaves as follows:

- A transient field is never stored in a database.
- A transient field can be initialized in a constructor just like any other field.
- When an object is materialized from a database, a transient field has the value that the constructor gives it.
- By overriding the **postInitializeContents()** method, you can synchronize a transient field for an object when its contents are refreshed from the database.
- When an object becomes hollow or stale, a transient field is not cleared.
- If you assign the value of a persistent object to a transient field, all memory of the reference is lost when the enclosing object is garbage collected.
- If you try to access a transient field outside a transaction, ObjectStore throws NoTransactionInProgressException if the containing object is hollow or ObjectException if the containing object is stale.
- Committing or aborting a transaction has no effect on a transient field.

Preventing fetch() and dirty() Calls on Transient Fields

When you run the postprocessor on a class that has transient fields, you might want to specify the **-noannotatefield** option for the transient fields. This option prevents access to the specified field from causing **fetch()** and **dirty()** calls on the containing object. This is useful for transient fields when you access them outside a transaction. Normally, access to a transient field causes **fetch()** or **dirty()** to be called to allow the **postInitializeContents()** and **preFlushContents()** methods to convert between persistent and transient states.

When you specify the **-noannotatefield** option, follow it with a qualified field name.

Background Information About Access to Transient Fields

Suppose you define class **X** with transient field **Y**. When you try to access **X.Y** outside a transaction, you receive NoTransactionInProgressException. This happens because the postprocessor annotates access to transient fields the same way that it annotates access to persistent fields, that is, with calls to **fetch()** and **dirty()** as needed. If you do not want this behavior, specify the **-noannotatefield** option followed by the qualified name of the transient field when you run the postprocessor.

The default behavior of the postprocessor is to insert calls to **fetch()** and **dirty()**. This is because transient fields need to be initialized in objects that are read from the database, and initialization might be based on persistent state. You can define the **postInitializeContents()** and **preFlushContents()** methods to maintain persistent and transient fields in parallel. The calls to **fetch()** and **dirty()** trigger the calls to these methods.

If the **-noannotatefield** behavior was the default, transient fields would not be initialized and there would be no error notification of this. Since this is not the default, you receive an exception if and when there is a problem.

Avoiding **finalize()** Methods

Object Design strongly recommends that you do not define **java.lang.Object.finalize()** methods in application classes that are persistence-capable. If your persistence-capable class must define a **finalize()** method, you must ensure that the **finalize()** method does not access any persistent objects. This is because the Java garbage collector might call the **finalize()** method outside a transaction or from a thread that does not belong to the session of the object being finalized. Such a situation causes ObjectStore to throw ObjectStoreException and prevents execution of the **finalize()** method.

If your class defines a **finalize()** method, the class file postprocessor inserts annotations at the beginning of the **finalize()** method that change the persistent object to a transient object. This makes it safe to access fields of the finalized object. However, if the object has not been fetched, the fields are in an uninitialized state.

Troubleshooting Access to Persistent Objects

Incorrect program behavior can happen when your program does one of the following while an annotated method is executing:

- Aborts, commits, or checkpoints a transaction
- Evicts one or all objects

The general result is that your program might incorrectly access additional persistent objects after the abort, commit, checkpoint, or eviction. The specific results vary according to the retain setting ObjectStore uses for the operation:

- **ObjectStore.RETAIN_STALE** should cause ObjectStore to throw ObjectException if your program tries to access a stale object. With the optimizations, your program might be able to access stale objects, which should not happen.
- **ObjectStore.RETAIN_HOLLOW**
  ObjectStore.RETAIN_READONLY
  ObjectStore.RETAIN_UPDATE

  These settings might cause your program to retrieve **null** or **0** values in place of correct values. Also, ObjectStore might fail to save some modifications in the database.

The class file postprocessor (**osjcfp**) uses two optimizations that can cause incorrect access to persistent objects. Consequently, there are two options to the postprocessor that allow you to disable these optimizations:

- **-noarrayopt** disables optimization of **fetch()** and **dirty()** calls for array objects in looping constructs. This causes **osjcfp** to make the calls to **fetch()** or **dirty()** in every iteration rather than only in the first loop iteration.
- **-nothisopt** disables optimization of **fetch()** and **dirty()** calls for access to fields relative to **this** in nonstatic member methods. This causes **osjcfp** to insert a **fetch()** or **dirty()** call for each access to a field in **this**.

Specify these options when you recognize the behavior described here.

Handling Unregistered Types

ObjectStore creates objects of type **UnregisteredType** when it must create a persistent object and it cannot find a **ClassInfo** subclass that describes the object it must create. The **ClassInfo** subclass might not be found because of a problem with the **CLASSPATH** or because the **ClassInfo** subclasses are not available for a particular database.

Typically, the postprocessor creates a **ClassInfo** subclass for types you want to store in a database. If you do not run the postprocessor, you must define a **ClassInfo** subclass yourself. ObjectStore uses this **ClassInfo** subclass to register the class the first time ObjectStore encounters the class in your application. If ObjectStore cannot find the **ClassInfo** subclass that describes a type, that type is unregistered.

If your application receives error messages that indicate unregistered types, the information here can help you determine what is happening and what to do about it. This section discusses

How Can There Be Unregistered Types?

How can there be a type in the database with no corresponding **ClassInfo** subclass? This can happen when

- The **CLASSPATH** environment variable has been changed since the object was stored in the database, and the **ClassInfo** subclass is no longer in the **CLASSPATH**.
- The **CLASSPATH** might include the directory or **.zip** or **.jar** file that contains the original class files, but not the directory or **.zip** or **.jar** file that contains the postprocessed class files with their corresponding **ClassInfo** subclasses.
- The database was received from another computing environment, but the corresponding **ClassInfo** class files were not sent along with the database. For example, you might receive a database with a **.zip** or **.jar** file that does not include the postprocessed class files for the types in the database.

Can Applications Work When There Are Types Not Registered?

In some situations, it might not matter to your application that there is an object whose type is unregistered. For example, suppose you are looking up an element in a hash table. One of the elements in the hash table is of an unregistered type, but it is not the element you are looking for. Because ObjectStore creates an **UnregisteredType** object instead of throwing an exception, your application can keep running.

What Does ObjectStore Do About Unregistered Types?

ObjectStore provides the abstract class **UnregisteredType** to represent objects whose types are unregistered. When ObjectStore cannot find the **ClassInfo** subclass for a type that is referenced in your application, it

- Creates an **UnregisteredType** object to represent the type
- Uses the **UnregisteredType** object in place of the hollow object it would have created

You can never read or modify an **UnregisteredType** object. Because of this, it is important for you to understand

- When ObjectStore creates **UnregisteredType** objects
- Whether or not ObjectStore can use **UnregisteredType** objects in a particular situation

With this information, you can determine whether your application can run with objects of unregistered types. Your application can continue to run as long as you do not try to read or modify an **UnregisteredType** object.

When Does ObjectStore Create UnregisteredType Objects?

ObjectStore creates an **UnregisteredType** object when it encounters an object in a database and it determines that the type of that object is not registered. ObjectStore encounters an object in a database when it

- Obtains the value of a database root
- Initializes an object and the value of one of the fields is a class, interface, or array
- Initializes an array and the element type of the array is a class, interface, or array
- Iterates over all objects in a segment

In the above list, *initialize* means to read the contents of the object out of the database and into the persistent Java object. This happens when ObjectStore calls **IPersistent.initializeContents()** and **IPersistent.postInitializeContents()**.

When ObjectStore encounters an object in a database, it determines whether there is already a Java object for the object in the database. If there is, ObjectStore uses that object. If there is not, ObjectStore checks to see if the type of the object is registered.

If the type is not registered, ObjectStore tries to load the **ClassInfo** subclass for the type and register it. ObjectStore uses the regular Java class loading mechanism. Usually, this means that ObjectStore searches your **CLASSPATH**. Depending on the Java implementation you are using, Java class loading can also involve Java **ClassLoader** objects, as described in the *Java Language Specification*.

If ObjectStore cannot load the **ClassInfo** subclass, it cannot register the type and consequently it cannot create a hollow object for the type. In this case, ObjectStore creates a new Java object of type **UnregisteredType** and uses it in place of the hollow object.

ClassInfo background

Typically, the postprocessor defines a **ClassInfo** subclass for your class. If you do not run the postprocessor, you must manually create the **ClassInfo** subclass. The name of the **ClassInfo** subclass is usually the name of the class with **ClassInfo** as a suffix. However, the suffix can also be **Info** or **CI** or another value that you specify with the **-classinfosuffix** postprocessor option.

Can Your Application Run with UnregisteredType Objects?

ObjectStore can use the **UnregisteredType** object if **java.lang.Object** is the type of the field in which the reference is being stored. For example, suppose you have the following class:

```
class Person {
    Pet mypet;
    Object mytrash;
}
```

You also have a database that contains one **Person** object. The value of the **Person.mypet** instance variable is an instance of the **Pet** class. The value of the **Person.mytrash** instance variable is an instance of the **Shoe** class.

Now, suppose that the **Pet** class is an unregistered class. Your application opens the database and tries to read the **Person** object. This means ObjectStore must initialize the **Person** object. When ObjectStore recognizes that the **Pet** class is unregistered, it creates an **UnregisteredType** object. ObjectStore then tries to assign the **mypet** instance variable to the **UnregisteredType** object. The code to do this is something like this:

```
mypet = (Pet)(handle.getClassField(1, XXX));
```

Typically, the postprocessor generates this code, but you can specify it yourself in the **IPersistent.initializeContents()** method. In any case, the call to **handle.getClassField()** returns an **UnregisteredType** object. The cast to **Pet** is required because **Pet** is the type of the **mypet** instance variable. However, this cast does not work. You cannot cast an **UnregisteredType** to **Pet** because **UnregisteredType** is not **Pet** and is not a subclass of **Pet**. The Java VM throws a ClassCastException in the middle of the initialization. The **Person** object is never initialized.

Now suppose that the **Pet** class *is* registered, and that the **Shoe** class, which is the type of the **Person.mytrash** instance variable, is not registered. ObjectStore creates an **UnregisteredType** object and the **handle.getClassField()** method returns it:

```
mytrash = (Object)(handle.getClassField(1, XXX));
```

This time, the cast works correctly because **UnregisteredType** is a subclass of **Object**. The initialization succeeds, and the application continues to run.

Troubleshooting ClassCastExceptions Caused by Unregistered Types

If ObjectStore creates an **UnregisteredType** object and you do not try to do anything with it, your application should work just fine. Now suppose you try to do something with it. Since it exists, it must be in a variable of type **java.lang.Object**. (If it were not, you would have had trouble with it earlier, as in the **Pet** example in the previous section.)

You cannot do very much with objects of type **Object**, so it is likely that the first thing you would do is try to cast the **UnregisteredType** object to some specific type that you expect it to be. However, this does not work. If you try to cast an **UnregisteredType** object to a type other than **java.lang.Object** or **UnregisteredType**, the Java VM throws ClassCastException.

Unfortunately, the ClassCastException does not identify the type that is unregistered. There are two ways that you can determine the name of the type that is not registered:

- Change your program.
- Set the **COM.odi.trapUnregisteredType** property

Changing your program

Somewhere in your program, you have a variable of type **Object** whose value is an object of the **UnregisteredType** class. Modify your program to cast this variable to type **UnregisteredType**. Then invoke the **getTypeName()** method on the **UnregisteredType** object. This returns the name of the type that is unregistered.

The disadvantage of this approach is that you must edit and recompile your code.

Setting the **trapUnregisteredType** property

ObjectStore provides the **COM.odi.trapUnregisteredType** property to help you determine which class is unregistered. The default is that this property is not set, and it is usually best to use the default.

When ObjectStore determines that a type is not registered, it checks the setting of the **COM.odi.trapUnregisteredType** property. If the property is not set, the default, ObjectStore creates an **UnregisteredType** object to represent the unregistered type. If **COM.odi.trapUnregisteredType** is set, ObjectStore throws FatalApplicationException and provides a message that indicates the name of the class that is unregistered.

Advantage

The advantage of the **COM.odi.trapUnregisteredType** property is that it provides the name of the class that is unregistered.

Disadvantage

The disadvantage is that as soon as ObjectStore encounters the first object whose type is unregistered, your application stops running. If the object you want information about is the second object of an unregistered type that ObjectStore would encounter, ObjectStore never reaches that second object. When you set **COM.odi.trapUnregisteredType**, ObjectStore throws FatalApplicationException as soon as it encounters the first object whose type is unregistered.

Troubleshooting the Most Common Problem

There is a common situation in which an **UnregisteredType** object causes a ClassCastException. This is when you try to obtain a database root (**Database.getRoot()**) and the value of the root is an **UnregisteredType** object. For example:

```
Foo foo = (Foo) db.getRoot("foo");
```

If the **Foo** class is unregistered, the Java VM throws a ClassCastException when it comes to the **(Foo)** cast operation. See the previous section for two ways to determine which class is unregistered in this situation.

However, when the value of a root is an unregistered type, it can mean that none of your persistence-capable types is registered. This is often true when an **UnregisteredType** object causes a ClassCastException very early in your program. Your best course of action is likely to be to ensure that your persistence-capable classes are in your **CLASSPATH**, rather than trying to determine which class is not registered.

Chapter 7
Working with Collections

ObjectStore provides a set of persistence-capable utility collections classes in the **COM.odi.util** package. These classes mirror those provided in the upcoming JDK 1.2 release.

ObjectStore includes another package that contains collections classes. The **COM.odi.coll** packages provides the API for the ObjectStore peer collections. Use these collections when you want to access C++ as well as Java. Information about these collections is in the book *Developing ObjectStore Java Applications That Access C++*.

This chapter discusses the following topics:

Description of ObjectStore Utility Collections

ObjectStore provides a number of utility collections interfaces and classes in the **COM.odi.util** package. In addition, ObjectStore provides a query facility in the **COM.odi.util.query** package.

A collection is an object that groups together other objects. It provides a convenient means of storing and manipulating groups of objects, and supports operations for inserting, removing, and retrieving elements.

Collections form the basis of the ObjectStore query facility, which allows you to select those elements of a collection that satisfy a specified condition. However, some collections can be queried, and others cannot. Consequently, before you create a collection and store it in a database, you should consider how you plan to use a collection. When you know what you need, you can select the best persistent collection representation for your application.

To introduce you to the ObjectStore utility collections facility, this section discusses the following topics:

Introduction to COM.odi.util Interfaces and Classes

The **COM.odi.util.Collection** and **COM.odi.util.Map** interfaces provide methods for operating on ObjectStore collections.

- **Collection** provides methods for operating on groups of objects in which the objects might be ordered, might include duplicates, and can be queried. The internal representation of a class that implements **Collection** might be a hash table or a binary tree or some other data structure.
  - The **COM.odi.util.List** interface extends **Collection**. In collections that implement **List**, the elements are ordered and duplicates are allowed.
  - The **COM.odi.util.Set** interface extends **Collection**. In collections that implement **Set**, the elements are not ordered and duplicates are not allowed.
- **Map** provides methods for operating on groups of key/value entries. Each key can map to at most one value. You cannot query collections that implement **Map**.

The ObjectStore utility collections facility provides the persistence-capable **COM.odi.util** classes shown in the following table. Most of these classes implement a **COM.odi.util** interface (many implement other interfaces as well).

Class	*Implements*
OSHashBag	**Collection**
OSHashMap	**Map**
OSHashSet	**Set**
OSHashtable	None
OSTreeMapByteArray	**Map**
OSTreeMapDouble	**Map**
OSTreeMapFloat	**Map**
OSTreeMapInteger	**Map**
OSTreeMapLong	**Map**
OSTreeMapString	**Map**
OSTreeSet	**Set**
OSVector	**Collection**
OSVectorList	**List**

Postprocessing

You do not need to postprocess the classes in the utility collections facility. They are already persistence-capable. If you define a subclass that extends any of these classes and you want the subclass to be persistence-capable, you must either run the postprocessor on the subclass or manually annotate the subclass.

Example

The **query** demo provides an example of using ObjectStore with utility collections. See the **README** file in the **COM/odi/demo/query** directory.

JDK 1.2

The JDK 1.2 collections interfaces specify the behavior of the **hashCode()** method on instances of the **Set**, **Map**, and **List** types. This **hashCode()** specification is based on the contents of the collection; the **hashCode** of a collection changes depending on what elements are added or removed. This means that it is not advisable to store an instance of a set, map, or list class in a hash table, unless the set or list is immutable and will never change.

Future change

After the JDK 1.2 is released, Object Design will modify ObjectStore so that it implements the JDK 1.2 collections interfaces. At that time, ObjectStore will no longer need to provide, and so will not provide, the following interfaces:

- **COM.odi.util.Collection**
- **COM.odi.util.List**
- **COM.odi.util.Map**
- **COM.odi.util.Set**
- **COM.odi.util.Iterator**
- **COM.odi.util.ListIterator**

Therefore, this discussion of transient views of **Map** classes pertains to **OSHashtable** as well.)

Description of OSHashBag

An **OSHashBag** is an unordered collection that allows duplicates. **OSHashBag**s not only keep track of what their elements are, but also of the number of occurrences of each element. As the name implies, a hash table is the internal representation for an **OSHashBag**. **OSHashBag** directly implements the **COM.odi.util.Collection** interface and so you can query instances of **OSHashBag**.

Description of OSHashMap

An **OSHashMap** is also an unordered collection that allows duplicates. Unlike **OSHashBag**, **OSHashMap** associates a key with each value in the map. When you insert a value into an **OSHashMap** , you specify the key along with the value. You can retrieve a value with a given key. The internal representation of an **OSHashMap** is a hash table. **OSHashMap**s do not allow null keys or null values.

Since **OSHashMap** implements the **Map** interface rather than the **Collection** interface, you cannot query **OSHashMap**s. However, you can query the collection views of a map: **Map.keySet(), Map.values()**, and **Map.entries()**. See Querying Collection Views of Map Entries on page 202.

The **OSHashMap.equals()** method performs value (contents) comparisons as described by **Map.equals()** to determine whether two **Map**s are equal. This is the only difference between **OSHashMap** and **OSHashtable**. The **OSHashtable.equals()** method compares the identity of the two objects to determine equality. The **OSHashtable**.**hashcode()** method generates a hash code based upon object identity; it is not based on the contents of the **OSHashtable**. For information about content comparisons and identity comparisons, see OSHashtable and OSVector on page 204.

A call to **OSHashMap.hashCode()** throws UnsupportedOperationException. See Unsupported operations on page 203.

Description of OSHashSet

An **OSHashSet** is an unordered collection that does not allow duplicates. If you try to insert a value into an **OSHashSet** and the set already contains that value, the set remanins unchanged. **OSHashSet** implements the **COM.odi.util.Set** interface. As its name implies, a hash table is the internal representation of an **OSHashSet**. Since **OSHashSet** indirectly implements **COM.odi.util.Collection**, you can query **OSHashSet**s.

OSTreeSets are capable of storing much larger persistent collections than **OSHashSet**s. However, **OSTreeSet**s must be persistent; it is not possible to create a transient instance of an **OSTreeSet**. If your collection is small, an **OSHashSet** is the best choice. If your collection is large, an **OSTreeSet** performs better.

A call to **OSHashSet.hashCode()** throws UnsupportedOperationException. See Unsupported operations on page 203.

Description of OSHashtable

An **OSHashtable** is also an unordered collection that allows duplicates. This class has the same APIs as **java.lang.Hashtable**.

OSHashtable associates a key with each element. When you insert an element into an **OSHashtable** , you specify the key along with the element. You can retrieve an element with a given key. While the internal representation of an **OSHashtable** is a hash table, it is a map-like structure.

Since **OSHashtable** does not implement the **COM.odi.util.Collection** interface, you cannot query **OSHashtable**s. However, you can query the collection views of an **OSHashtable**. See Querying Collection Views of Map Entries on page 202.

The **OSHashtable.equals()** and **OSHashtable.hashCode()** methods perform reference (identity) comparisons and not value (contents) comparisons. This is the only difference between **OSHashtable** and **OSHashMap**. The **OSHashMap** methods perform content comparisons. For information about content comparisons and identity comparisons, see OSHashtable and OSVector on page 204.

By default, an **OSHashtable** allocates room for 50 elements. You can presize an **OSHashtable** to better match what your application really needs. In addition, you can delay allocation of **OSHashtable** substructure, which ObjectStore uses to represent the **OSHashtable**, until elements are actually added to the **OSHashtable**. To do this, specify the **lazy** argument to the **OSHashtable** contructor:

OSHashtable(int intitialBufferSize, int capacityIncrement, boolean lazy)

Description of OSTreeMap*xxx*

OSTreeMap is based on a binary tree representation that is tuned for large persistent collections. **OSTreeMap** is an abstract class with several concrete subclasses. In all **OSTreeMap***xxx* instances, the values are objects. As for the keys, there are separate classes for different types of keys, as shown in the following table:

Class	*Key Type*
OSTreeMapByteArray	**ByteArray**
OSTreeMapDouble	**Double**
OSTreeMapFloat	**Float**
OSTreeMapInteger	**Integer**
OSTreeMapLong	**Long**
OSTreeMapString	**String**

An **OSTreeMap***xxx* is an unordered collection that allows duplicates. Each **OSTreeMap***xxx* associates a key with a value in the map. When you insert a value into an **OSTreeMap***xxx* , you specify the key along with the value. You can retrieve a value with a given key. **OSTreeMap***xxx*s do not allow null keys or null values.

The **OSTreeMap***xxx* classes extend **OSTreeMap**, which implements **Map**. Consequently, you cannot query **OSTreeMap***xxx*s. However, you can query the collection views of a map: **Map.keySet(), Map.values()**, and **Map.entries()**. See Querying Collection Views of Map Entries on page 202.

The **OSTreeMap***xxx* classes are designed for very large persistent aggregations. These classes allow you to iterate over the collection or query the collection without fetching any objects from the database except those that are explicitly returned to you. ObjectStore does not even create hollow objects to represent the elements. **OSTreeMap** collections can only be persistent.

A call to **OSTreeMap.hashCode()** throws UnsupportedOperationException. See Unsupported operations on page 203.

Each **OSTreeMap***xxx* class has a constructor for exported objects.

Description of OSTreeSet

An **OSTreeSet** is an unordered collection that does not allow duplicates. If you try to insert a value into an **OSTreeSet** and the set already contains that value, the set remains unchanged. **OSTreeSet** implements the **COM.odi.util.Set** interface. As its name implies, a balanced tree is the internal representation of an **OSTreeSet**. Since **OSTreeSet** indirectly implements **COM.odi.util.Collection**, you can query **OSTreeSet**s.

The **OSTreeSet** class is designed for very large persistent aggregations. This class allows you to iterate over the collection or query the collection without fetching any objects from the database except those that are explicitly returned to you. ObjectStore does not even create hollow objects to represent the elements. **OSTreeSet** collections can only be persistent.

Object Design recommends that if you are going to query a collection that contains a particularly large number objects, define the collection as an **OSTreeSet** or a subclass of **OSTreeSet**. **OSTreeSet** is the only collections class for which ObjectStore provides the ability to add indexes. Indexes can speed up queries on very large collections. You can, of course, define the ability to add indexes to other types collections that implement **COM.odi.util.Collection**. See Enhancing Query Performance with Indexes on page 222.

The main difference between **OSTreeSet** and **OSHashSet** is the internal representation. For very large collections, **OSTreeSet** is the best choice. However, **OSTreeSet**s can only be persistently allocated. It is not possible to create a transient **OSTreeSet**.

A call to **OSTreeSet.hashCode()** throws UnsupportedOperationException. See Unsupported operations on page 203.

The **OSTreeSet** class has a constructor for creating exported objects.

Description of OSVector

An **OSVector** is a collection that implements a persistent expandable array, as well as **COM.odi.util.Collection**. You can query **OSVector**s.

An **OSVector** associates each element with a numerical position based on insertion order. By default, **OSVector**s allow duplicates. In addition to simple insert (insert into the beginning or end of the collection) and simple remove (removal of the first occurrence of a specified element), you can insert, remove, and retrieve elements based on a specified numerical position, or based on a specified iterator position. An **OSVector** does not have quick lookup by object or key. Consequently, the overhead for an **OSVector** is lower than for utility collections that have quick lookup.

The **OSVector.equals()** and **OSVector.hashCode()** methods perform reference (identity) comparisons and not value (contents) comparisons. This is one difference between **OSVector** and **OSVectorList**. The **OSVectorList** methods perform content comparisons. For information about content comparisons and identity comparisons, see OSHashtable and OSVector on page 204.

By default, an **OSVector** allocates room for 32 elements. You can presize an **OSVector** to better match what your application really needs. In addition, you can delay allocation of **OSVector** substructure, which ObjectStore uses to represent the **OSVector,** until elements are actually added to the **OSVector**. To do this, specify the **lazy** argument to the **OSVector** contructor:

OSVector(int intitialBufferSize, int capacityIncrement, boolean lazy)

Description of OSVectorList

An **OSVectorList** is a collection that implements a persistent expandable array. It implements the **List** interface and functions exactly like an **OSVector**, except in the following way.

An **OSVectorList** does not have quick lookup by object or key. Consequently, the overhead for an **OSVectorList** is lower than for utility collections that have quick lookup.

The **OSVectorList.equals()** and **OSVectorList.hashCode()** methods perform value (contents) comparisons and not reference (identity) comparisons. This makes **OSVectorList** unsuitable for storage in a persistent hash table or any other hash table based collection representation. The **OSVector** methods perform identity comparisons. For information about content comparisons and identity comparisons, see OSHashtable and OSVector on page 204.

A call to **OSVectorList.hashCode()** throws UnsupportedOperationException. See Unsupported operations on page 203.

Advantages of Using ObjectStore Utility Collections

The advantages of using **COM.odi.util** interfaces and classes are as follows:

- The interfaces and classes in **COM.odi.util** are designed to be compatible with the upcoming JDK 1.2 release.
- The classes are persistence-capable.
- There are collection representations that support queries.
- There are classes that are designed for very large aggregations — **OSTreeMap***xxx* and **OSTreeSet**.

Querying Collection Views of Map Entries

The **OSHashMap** and **OSTreeMap***xxx* classes extend **COM.odi.util.Map** and not **COM.odi.util.Collection**, and therefore you cannot use the ObjectStore query facility on them. However, each of the classes that implements **Map** defines the following methods:

- **keySet()** returns a **COM.odi.util.Set** view of the keys contained in the map
- **values()** returns a **COM.odi.util.Collection** view of the values contained in the map
- **entries()** returns a **COM.odi.util.Set** view of the key / value mappings contained in the map

The **OSHashtable** class, although it does not implement **Map**, also defines these methods.

You can use the ObjectStore query facility to query the **Collection** and **Set** views returned by the **keySet()**, **values(),** and **entries()** methods.

Transient views

While **OSHashtable**, **OSHashMap**, and the **OSTreeMap***xxx* subclasses are persistence-capable, the views returned by the **entries()**, **keySet()**, and **values()** methods are not. These are transient views of persistence-capable classes.

Background About Utility Collections and JDK 1.2 Collections

Here is some background information about how the ObjectStore utility collections fit with the JDK 1.2 collections. This discussion assumes that you are familiar with the JDK 1.2 collections API. If you are not, see **http://java.sun.com/products/jdk/1.2/docs/guide/collections/reference.html**.

ObjectStore provides a collections package that parallels the JDK 1.2 **java.util** collections. In addition, ObjectStore includes query and indexing facilities. The new collections implementations are in the **COM.odi.util** package.

The core collections interfaces defined in the JDK 1.2 **java.util** package are:

- **java.util.Collection**
- **java.util.Set**
- **java.util.List**
- **java.util.Map**

In the JDK 1.2, collections classes and behaviors are based on these interfaces. Consequently, you can usually use any representation that is parallel to a particular interface. The **java.util** implementations and their corresponding ObjectStore implementations are shown in the following table:

Interface	*java.util Class*	*ObjectStore Class*
Collection	None	**COM.odi.util.OSHashBag**
Set	**java.util.HashSet**	**COM.odi.util.OSHashSet**
Set	**java.util.ArraySet**	**COM.odi.util.OSTreeSet**
List	**java.util.Vector**	**COM.odi.util.OSVector**
List	**java.util.ArrayList**	**COM.odi.util.OSVectorList**
List	**java.util.LinkedList**	None
Map	**java.util.Hashtable**	**COM.odi.util.OSHashtable**
Map	**java.util.HashMap**	**COM.odi.util.OSHashMap**
Map	**java.util.ArrayMap**	None
Map	**java.util.TreeMap**	**COM.odi.util.OSTreeMap***xxx*

Unsupported operations

In the **COM.odi.util** package, all persistence-capable collections that implement the **Set**, **List**, and **Map** interfaces throw the

UnsupportedOperationException, when the **hashCode()** method is invoked on them. This is because the definition of the computation of **hashCode()** for these interfaces is currently in a state of flux in JDK 1.2 beta 3. When JDK 1.2 collections are finalized, ObjectStore will provide **hashCode()** methods that conform to their JDK 1.2 specificatons. In the interim, you can subclass these representations, and define a suitable overriding **hashCode()** method if your applications needs it.

OSHashtable and **OSVector**

COM.odi.util.OSHashtable and **COM.odi.util.OSVector** have been updated to be parallel to most of the JDK 1.2 specifications. They do not quite meet the description of the JDK 1.2 behavior for **equals()** and **hashCode()**. The JDK 1.2 changed this behavior in an incompatible way for these two classes.

The JDK 1.2 **List**, **Set**, and **Map** interfaces mandate an **equals()** method that does value comparison and not reference comparison. That is, two **Set**s are equal if they have the same elements, two **List**s are equal if they have the same elements in the same order, and two **Map**s are equal if they have the same key / value pairs.

This places corresponding constraints on the **hashCode()** method, since **(a.equals(b)) => (a.hashCode()==b.hashCode())**. The ObjectStore **OSHashtable** and **OSVector** classes, however, implement persistent (unchanging) **hashCode**s, and rely on **Object.equals()**. The JDK definition for **hashCode** means that classes that meet the JDK 1.2 specification should not be stored in hash tables, because their **hashCode**s change when elements are added or removed. So for these two classes, ObjectStore retains the old identity-based definitions, rather than moving to the new content-based definitions of **equals()** and **hashCode()**.

Collection interface

There are no concrete implementations of the **Collection** interface in the JDK 1.2. **Collection** is essentially a **Bag**, that is, a **Set** that might contain duplicates. ObjectStore includes the **COM.odi.util.OSHashBag** and **COM.odi.util.OSVector** classes to implement **Collection**.

How to Choose a Collections Alternative

Your choice of how to implement collections depends on

- The amount of data to be stored in the collection. The numbers below provide a rough guideline for determining whether a collection is of small, medium, or large size. Use them as a starting point when you decide which collection type is best for your application.
  - Small collections have fewer than 100 elements.
  - Medium collections contain as many as 10,000 elements
  - Large collections have more than 10,000 elements.
- Your familiarity with a third-party library. You might want to use a particular library just because you already know how to use it.
- Features required by your application.

  If you want to access your collection from C++, use the collections in the **COM.odi.coll** package. See *Developing ObjectStore Java Applications That Access C++*.
- Importance of compatibility with the JDK 1.2 collection interfaces. When the JDK 1.2 is released, it will provide standard interfaces for collection representations. In a future release, ObjectStore will no longer provide the **Collection**, **List**, **Map**, and **Set** interfaces in **COM.odi.util** and will instead implement the JDK 1.2 interfaces.
- Importance of compatibility between OSJI and PSE/PSE Pro.

For applications that already use the **COM.odi.coll** collections, the **COM.odi.util.OSTreeMap***xxx* collections are comparable to the **COM.odi.coll.Dictionary**_*xxx* classes, while the **COM.odi.util.OSTreeSet** class is comparable to the **COM.odi.coll.Set** class. These classes are comparable in that their performance should be about the same.

To help you choose the right persistent collection representation for your application, the following table compares the behavior of the utility collections in **COM.odi.util**.

Collection Class	*Ordered/ Unordered*	*Duplicates/ No duplicates*	*Quick Lookup*	*Comparison Operations*	*Queries Allowed*	*Collection Size*
OSHashBag	Unordered	Duplicate values allowed	Object lookup	Identity-based	Can query	Medium
OSHashMap	Unordered	Duplicate values allowed No duplicate keys	Key lookup	Content-based	No queries	Medium
OSHashSet	Unordered	No duplicates	Object lookup	Content-based	Can query	Medium
OSHashtable	Unordered	Duplicate values allowed No duplicate keys	Key lookup	Identity-based	No queries	Medium
OSTreeMap*xxx*	Ordered	Duplicate values allowed No duplicate keys	Key lookup	Content-based	No queries	Large
OSTreeSet	Unordered	No duplicates	Object lookup	Content-based	Can query and index	Large
OSVector	Ordered	Duplicate values allowed	None	Identity-based	Can query	Small, medium
OSVectorList	Ordered	Duplicate values allowed	None	Content-based	Can query	Small, medium

The **OSHashtable** class is not compatible with the JDK 1.2 API. All other collections in **COM.odi.util** are compatible with the JDK 1.2 API.

Using ObjectStore Utility Collections

To help you use ObjectStore utility collections, this section discusses the following topics:

Creating Collections

Each collection representation has one or more constructors that you can use to create collections. For details about each classes' constructors, see the *ObjectStore Java API Reference*. For example:

```
Database db = Database.create(args[1], ALL_READ | ALL_WRITE);
Transaction.begin(UPDATE);
db.createRoot("collection", new OSTreeSet(db));
Transaction.current().commit();
```

Navigating Collections with Iterators

The **Iterator** and **ListIterator** interfaces help you navigate within a utility collection. An *iterator*, an instance of the **COM.odi.util.Iterator** or **COM.odi.util.ListIterator** interface, designates a position in a collection. You can use iterators to traverse collections, as well as to remove elements from collections.

With the JDK 1.2, **Iterator** takes the place of **Enumeration**. **Iterator** provides the same capabilitiess as **Enumeration** (though method names are different), and it also allows you to remove elements from the underlying collection. When the JDK 1.2 is released, ObjectStore will implement the JDK 1.2 **Iterator** and **ListIterator** interfaces and will no longer provide **Iterator** and **ListIterator** in **COM.odi.util**.

The **ListIterator** interface extends the **Iterator** interface. A class that implements **ListIterator** must also implement **List**. The additional methods that **ListIterator** provides allow you to

- Insert objects relative to the current position of the iterator.
- Traverse the list in reverse, as well as forward.
- Replace an element in the underlying list.
- Retrieve the index of an element.

The **IndexIterator** interface, also in **COM.odi.util**, allows you to traverse an index or map structure. You can use the **IndexIterator** interface to obtain the key and value for elements in the underlying collection.

Performing Collection Updates During Iteration

While you are iterating through a collection, you can use the **Iterator** and **ListIterator** interface methods to modify that collection. This assumes that the implementation of the **Iterator** or **ListIterator** interface supports the methods that modify underlying collections. (The JDK 1.2 defines some of these methods as optional. You should check the API reference information for the particular class you are using to determine exactly which behaviors are supported.)

You cannot use any other methods to update the collection while you are iterating through its elements. If you try to, ObjectStore throws ConcurrentModificationException.

When a thread is iterating over a collection, that thread and cooperating threads can modify the object returned by the iteration. If you are using an **Iterator**, your application cannot add elements to the collection or change the order of the collection. If you are using a **ListIterator**, your application can only use **ListIterator** methods to modify the collection.

Suppose you do add an element in the middle of an iteration, and then try to use the same iterator. ObjectStore recognizes that the collection has been modified and throws ConcurrentModificationException. At this point, if you create a new iterator, it recognizes the updated collection and does not throw an exception.

Querying ObjectStore Utility Collections

The **COM.odi.util.query.Query** class provides a mechanism for querying collections objects that implement the **COM.odi.util.Collection** interface. A query applies a predicate expression (an expression that evaluates to a **boolean** result) to all elements in a collection. The query returns a subset collection of all elements for which the expression is true. You can query the following types of collections:

- **OSHashBag**
- **OSHashSet**
- **OSTreeSet**
- **OSVector**
- **OSVectorList**

To accelerate the processing of queries on particularly large collections, you can build indexes on the collection. For information about indexes, see the next section, Enhancing Query Performance with Indexes on page 222.

This section provides the following information about queries on ObjectStore utility collections:

Creating Queries

To create a query, run the **COM.odi.util.query.Query** constructor and pass in a **Class** object and a query string. Here is the constructor:

public Query(Class elementType, String queryExpression)

There is also a constructor that allows you to specify a **FreeVariables** map.

elementType

The **elementType** class or interface provides the context in which the query facility interprets **queryExpression**. This must be a publicly accessible class or interface. When your application calls the **Query.select()** or **Query.pick()** method to execute the query against a particular collection, every element of that collection must be an instance of (in the sense of **instanceof**) the **elementType** that was specified when the query was created. Any element of the collection that is not an instance of **elementType** is not returned in the query result (even if it evaluates to true for the predicate).

queryExpression

The **queryExpression** is a predicate (that is, an expression with a **boolean** result) that the query facility evaluates on each element of the collection. The **queryExpression** operands can be literals and names.

Literals

Literals can be of any of the Java primitive types, including the special values **true**, **false**, and **null**. Since the query expression is a **String**, you must enclose any embedded strings in escaped quotation marks, like **\"this\"**.

Names

Names can consist of a single identifier, or they can consist of a sequence of identifiers separated by periods. Names can be either free variables or member accesses. You must explicitly specify free variables in the **freeVariables** argument of the three-argument **Query** constructor. Any name that is not a free variable is interpreted as a member access.

Member accesses are interpreted as accessing public members (including static members) of an object of class/interface **elementType**, if possible. This interpretation works as though there were an implicit **this** argument, of **elementType**, at the root of the name expression. Any member access that cannot be interpreted as a member access on **elementType** is interpreted as a

static access. Static accesses are resolved as if the package containing **elementType** were imported.

Queries can contain methods that take arguments.

Example

For example:

```
Query q = new Query(Employee.class, "salary < 50000");
```

The query expression can refer to classes without specifying a package name. ObjectStore treats the query expression as if it were defined in a file in another package that has imported the package of the **Class** object that was passed to the **Query** constructor. This default package only matters for class names, though, not for member access. Only public classes and members are accessible within the query.

An application can run the example query on a specific collection with a call to the **Query.select()** method that specifies the collection to be queried as the argument. For example:

```
Query q = new Query(Employee.class, "salary < 50000");
Collection employees = db.getRoot("employees");
Set result = q.select(employees);
```

When you create a query, you do not bind it to a particular collection. You can create a query, run it once, and throw it away. Alternatively, you can reuse a query multiple times against the same collection (perhaps with different bindings for free variables), or against different collections.

If something in your query is wrong, you find out at the point where you create the query. You do not need to wait for the application to optimize or execute the query. However, the query facility cannot detect incorrect free variable bindings until you specify them when you execute the query on a collection.

Description of Query Syntax

ObjectStore performs syntax analysis of the query expression in the context of the **elementType** class or interface that is passed to the query constructor. This must be a publicly accessible class or interface. It can also be a derived type.

When the query is executed against a particular collection using the **select()** or **pick()** method, every element of that collection must be an instance of (in the sense of **instanceof**) the **elementType** that was specified when the query was created.

The **queryExpression** is a predicate (that is, an expression with a **boolean** result). The query is executed on a collection by evaluating this query expression on each element of the collection. However, it might not be necessary to explicitly fetch and examine all elements of the collection. This depends upon the available indexes and query optimization strategy.

Supported operations

Queries on utility collections can include most Java operations:

- Arithmetic: **+ / - * %**
- Bitwise: **^ | &**
- Unary numeric: **~ -**
- Unary logic: **!**
- Relational: **> < <= >= instanceof**
- Equality: **== !=**
- String concatenation: **+**
- Conditional And, Or: **&& ||**
- Shift operations: **<< >> >>>**
- Cast operations: (*type*)

Unsupported operations

The following operations are not supported:

- Assignment: **= += *= /= %= -= <<= >>= >>>= &= ^= |=**
- Conditional: **?:**
- Array dereference: **[]**
- New: **new**
- Prefix / Postfix: **++ --**

Statements are not permitted. Only expressions are permitted.

For details on operations and the operands, see the *Java Language Specification*.

The operators have their usual Java meaning except for the relational and equality operators when used with **String** operands. In a query expression, ObjectStore uses these operators to compare the contents of the two strings. Null **String**s are considered to be less than all other values.

String literals

In a query expression, you must enclose **String** literals in escaped quotation marks. For example:

```
new Query(Foo.class, "name == \"Davis\"")
```

You can specify wildcards in query strings. You can search for substrings, and perform case insensitive searches. See Matching Patterns in Query Strings on page 215.

Wrapper objects

The query facility treats wrapper objects just like other **Object**s. For example, suppose you have the query expression **"A==B"**. **A** and **B** refer to **Integer** wrappers. This results in an identity check on the objects. The query facility determines whether **A** and **B** both refer to the same wrapper instance. The query facility does not check that the values of **A** and **B** are equal. You can specify **"A.intValue()==V.intValue()"** to compare contents.

This behavior might change in a future release so that the query facility treats wrapper objects in the way that it treats primitives. Consequently, you should not rely on the identity check for wrapper objects.

Miscellaneous

You can use parentheses to group expressions.

The precedence and associativity of the operators is the same as that for the Java language.

The entire query expression must resolve to a Boolean value.

Sample Program That Uses Queries

In the **COM/odi/demo/query** directory, there is a sample program that uses ObjectStore utility queries. See the **README.htm** file in that directory.

Matching Patterns in Query Strings

Specifying a pattern matching query

To specify a string pattern to be matched in a query, the Pattern Matching operator (**~~**) is used. This operator, which has greater precedence than the Multiplication operator (*****), has two arguments. These arguments must be either **String**s or null. The left-hand argument specifies the text to be checked for a match. The right-hand argument specifies the pattern to be matched.

Pattern matching characters

The following characters have special meanings when used in the right-hand argument of the Pattern Matching operator. All other characters match themselves.

Operator	*Function*
?	Matches any single character
*****	Matches 0 or more of any character
&	Escape character
[	Reserved
]	Reserved
(	Reserved
)	Reserved
|	Reserved

Note

The reserved characters are invalid if they are not preceded by an ampersand (**&**).

The following table shows special two character sequences, known as escape sequences, that start with an ampersand (**&**). These escape sequences are used to include characters literally in the pattern without their special meaning and to enable case insensitive matching.

Note that the ampersand (**&**) must appear in front of every sequence. An ampersand followed by any other character is invalid.

Escape Sequence	*Function*
&?	Matches a question mark
&*	Matches an asterisk
&[	Matches left square bracket
&]	Matches right square bracket
&(	Matches left parentheses
&)	Matches right parentheses
&|	Matches a vertical bar
&&	Matches an ampersand
&i	Enables case insensitive matching.

Case sensitivity in matching

By default, pattern matches are case sensitive. The **&i** escape sequence enables case insensitive matching for an entire pattern. This escape sequence can only be specified at the start of a pattern.

Optimizing pattern matching

Pattern matching operator takes advantage of any ordered indexes available on the text being matched. If the pattern starts with a character other than an asterisk (*****) or a question mark (**?**), then the query only searches the portion of the index that matches the initial, constant prefix. Therefore, patterns that specify a constant prefix produce much more efficient queries.

Pattern matching examples

The following pattern matching examples use the following class:

```
public class Person { public String name; }
```

- Matching a name beginning with the characters **Tom**:

```
new Query(Person.class,"name ~~ \"Tom*\"");
```

- Matching a name ending with the characters **man** or **burn**:

```
new Query(Person.class,"name ~~ \"*man\" || name ~~\"*burn\"");
```

- Matching a name using a single wildcard character with bound variable:

```
FreeVariables vars = new FreeVariables();
vars.put("var", String.class);
Query query = new Query(Person.class,"name ~~ var", vars);
FreeVariableBindings bindings = new FreeVariableBindings();
bindings.put("var","*Gr?y");
query.select(coll, bindings);
```

- Matching a name using a case insensitive match for **?foo**:

```
new Query(Person.class,"name ~~ \"&i&?foo\"");
```

- Matching a name using a case insensitive match for ***foo** appearing anywhere:

```
new Query(Person.class,"name ~~ \"&i*&*foo*\"");
```

- Matching a name **foo** appearing anywhere followed by **&bar**:

```
new Query(Person.class,"name ~~ \"*foo*&&bar*\"");
```

- Matching the name **(a)**:

```
new Query(Person.class,"name ~~ \"&(a&)\"");
```

Using Free Variables in Queries

Free variables are lexically the same as identifiers in the Java language. If you use free variables in your query, you must specify them in an optional third argument to the **Query** constructor. Use the **COM.odi.util.query.FreeVariables** class. This class implements the **Map** interface. In addition, it provides type-checking to ensure that the keys and values are **String**s and **Class**es, respectively. For example:

```
FreeVariables vars = new FreeVariables();
vars.put("INPUT_SALARY", int.class);
Query q = new Query(Person.class,
    "salary>=INPUT_SALARY", vars);
```

When you execute a query, you must bind any free variables to particular values. Do this by passing an additional argument to the **Query.select()** or **Query.pick()** method. This argument must be of type **COM.odi.util.query.FreeVariableBindings**. This class, like **FreeVariables**, implements the **Map** interface, and provides additional type-checking to ensure that the keys are **String**s.

The values you bind to the free variables must be of the type specified by the corresponding entry in the **FreeVariables** map that was specified at query construction. For primitive types, the type of value stored in the **FreeVariableBindings** must be the associated wrapper type. ObjectStore does not check that the correct types are bound until it executes the query.

For example, the **INPUT_SALARY** free variable is used in the previous example query. Your application might read in a value from a user in an interactive program, or compute the value in some other way. Regardless of how your application computes the value, the free variable is bound to a specific value only when the query is executed. For example:

```
int INPUT_SALARY = {user input or some other computation}
FreeVariableBindings bindings = new FreeVariableBindings();
bindings.put("INPUT_SALARY", new Integer(INPUT_SALARY));
Set result = q.select(employees, bindings);
```

Executing Queries

You can execute a query that

- Specifies predefined variables and / or free variables
- Returns one element or a set of elements

Obtaining a set

To obtain the set of elements that satisfy a query, call the **COM.odi.util.query.Query.select()** method. There are two overloadings:

public Set select(Collection coll)

public Set select(Collection coll,
FreeVariableBindings freeVariableBindings)

The **coll** argument specifies the collection to be queried. If this query has been explicitly optimized with the **Query.optimize()** method, any indexes specified in the optimization must be available on this collection. If this query has not been explicitly optimized, ObjectStore optimizes it for all indexes on the collection being queried. If the query has been explicitly optimized for indexes that are not available on the specified collection, ObjectStore throws QueryIndexMismatchException.

The **freeVariableBindings** argument specifies a **FreeVariableBindings** object that defines bindings for each free variable in the query. For each entry, the key is a **String** that identifies the free variable, and the value is the value that should be associated with the free variable during the evalution of the query. The value must be of the type specified by the corresponding entry in the **FreeVariable** argument passed to the **Query** constructor. For the query to be evaluated, every free variable associated with the query when it was constructed must have a corresponding binding. Also, every free variable binding must correspond to a free variable that was specified when the query was constructed. If the free variable bindings do not match the free variable definitions specified when the query was constructed, ObjectStore throws QueryException.

The **select()** method returns a **Set** that contains the elements that satisfy the query. If ObjectStore does not find any matching elements, it returns an empty collection. The returned **Set** is transient.

Obtaining a single element

To obtain one element that satisfies a query, call the **COM.odi.util.query.Query.pick()** method. There are two overloadings:

```
public Object pick(Collection coll)

public Object pick(Collection coll,
    FreeVariableBindings freeVariableBindings)
```

The **coll** and **freeVariableBindings** arguments are the same as for the **select()** method. The **pick()** methods return the first element found that satisfies the query. The returned element is transient. If no elements in the collection satisfy the query, ObjectStore returns NoSuchElementException.

Type of returned element

The **select()** and **pick()** methods never return elements that are not of the class that was specified as the collection element type when the query was constructed.

Null values

Queries ignore null elements but not null fields. The result set of a query never includes null elements. When a query reaches a null element, execution continues to the next element. Suppose you have a query like this:

```
name != "fred"
```

A query that evaluates this on a collection returns elements with null **name** fields, as well as elements with names that are not **"fred"**.

Now suppose you have a query like this:

```
spouse.name != "fred"
```

On a collection that includes elements that do not have spouses, this query does not return those elements without spouses. It only returns the elements that have spouses with names that are not **"fred"** plus the elements that have spouses with null **name** fields.

Limitations on Queries

When a query refers to a class or field, the class or field must be public.

When a query refers to a method, the method must return something. In other words, in a query string, you cannot refer to a method that returns void.

Queries no longer have the limitation against methods that take arguments. Queries can contain methods that take arguments.

Enhancing Query Performance with Indexes

When you want to run a query on a particularly large collection, it is useful to build indexes on the collection to accelerate query processing. An index provides a reverse-mapping from a field value or from the value returned by a method when it is called, to all elements that have the value. A query that refers to an indexed member executes faster. This is because it is not necessary to examine each object in the collection to determine which elements match the predicate. Also, ObjectStore does not need to fetch into memory every element.

This section discusses the following topics:

How Indexes Work

When you add an index to a collection, ObjectStore examines every element of the collection to determine the value of the indexed field or method. After you build the index, you can run queries against the collection without reexamining the elements to determine the values of any indexed members. The query examines the index instead of the collection.

A query can include both indexed fields / methods and nonindexed fields / methods. ObjectStore evaluates the indexed fields and methods first and establishes a preliminary result set. ObjectStore then applies the nonindexed fields / methods to the elements in the preliminary result set.

Adding Indexes to Collections

You can add indexes to any collection that implements the **COM.odi.util.Collection** interface, directly or indirectly. To add an index to a collection, the collection must implement the **COM.odi.util.IndexedCollection** interface, directly or indirectly. Note that the **IndexedCollection** interface extends the **Collection** interface.

The **IndexedCollection** interface provides methods for adding and removing indexes, and updating indexes when the indexed data changes. In this release of ObjectStore, **COM.odi.util.OSTreeSet** is the only collection class that already implements **IndexedCollection**. You can, of course, define other **COM.odi.util.Collection** classes that implement **IndexedCollection**. Call the **COM.odi.util.IndexedCollection.addIndex()** method to create an index. There are three overloadings:

- **addIndex(Class elementType, String path)**
- **addIndex(Class elementType, String path, boolean ordered, boolean duplicates)**
- **addIndex(Class elementType, String path, boolean ordered, boolean duplicates, Placement placement)**

The **elementType** argument indicates the type to which the index applies. Objects of other types can be in the collection that you index, but they are ignored by the index. A query that uses the index does not return such elements.

The **path** argument indicates the member to be indexed. A method member can have no arguments or one constant argument.

The **ordered** and **duplicates** arguments allow you to specify whether the index is ordered and whether it allows duplicates. If you do not specify the **boolean** arguments, the index is unordered and it allows duplicates.

Finally, the **placement** parameter indicates the database or segment in which to store the index. The path must be either the name of a public field or a call to a public instance method. If it is not, ObjectStore throws IndexException. The public instance method can be in a superclass. Indexes on paths that specify more than one field or method access are not allowed. If you do not pass a **Placement** argument, ObjectStore stores the index in the same database and segment as the collection.

Dropping Indexes from Collections

Call the **COM.odi.util.IndexedCollection.dropIndex()** method to remove an index from a collection. Here is the method signature:

public boolean dropIndex(Class elementType, String path)

The **elementType** argument indicates the type to which the index applies.

The **path** argument indicates the member for which the index is being removed.

Sample Program That Uses Indexes

In the **COM/odi/demo/query** directory, the **QueryCustomers** class includes the following example of using an index:

```
IndexedCollection collection = new OSTreeSet(db);
try {
    collection.addIndex(Employee.class, "salary");
}   catch (IllegalAccessException e) {
    System.err.println("Couldn't access field: " + e);
    System.exit(1);
}
Set result = q.select(employees);
```

Modifying IndexValues

After you add an index to a collection, ObjectStore automatically maintains it as you add or remove elements from the collection. However, it is your responsibility to manage index maintenance when indexed members are modified for instances that are already members of an indexed collection.

For example, suppose you insert **Lee** into your collection of employees. You build an index for this collection on the **phoneExtension** field. A query of **"phoneExtension == 1234"** returns **Lee**. If you remove Lee from the collection, ObjectStore updates the index so it no longer includes **Lee**. However, if you leave **Lee** in the collection, but change **Lee**'s phone extension, you must manually correct the index so that **Lee** refers to the correct phone extension.

Methods

There are three methods that you can use to manually maintain an index:

- **IndexedCollection.removeFromIndex()** removes a value from the index.
- **IndexedCollection.addToIndex()** inserts a value into the index.
- **IndexedCollection.updateIndex()** removes a value from the index and replaces it with a value that you specify.

After an application calls one of these methods, the next time the application uses that index it uses the updated index. A call to **updateIndex()** does the same thing as a call to **removeIndex()** followed by a call to **addToIndex()**. Except, **removeIndex()** and **addToIndex()** inspect the value to determine the index key. That is, they apply the index's path expression to obtain the key from the value. With **updateIndex()**, you pass in the old key and the new key. ObjectStore does not have to inspect the value to determine its key. For this reason, and because there is a single call, using **updateIndex()** is more efficient.

Removing and adding index values

The **removeFromIndex()** method has two overloadings:

```
public void removeFromIndex(Object value)
```

```
public void removeFromIndex(Class elementType,
    String path, Object value)
```

The **addToIndex()** method has two parallel overloadings:

```
public void addToIndex(Object value)
```

```
public void addToIndex(Class elementType,
    String path, Object value)
```

Usually, after you remove a value from an index, you should add a value to replace it.

If you know exactly which value you need to add or remove, you can use the form that specifies **elementType**, **path**, and **value**. If you do not know what indexes exist, or if you modified a lot of different fields and want to update all indexes, use the short form. In this case, ObjectStore iterates over all indexes and updates all of them.

Here is an example of removing and adding values to an index:

```
Employee lee = new Employee("Lee", 1234);
collection.insert(lee);
try {
    collection.removeFromIndex(lee);
    lee.setExtension(5678);
    collection.addToIndex(lee);
}   catch (IllegalAccessException e) {
    System.err.println("Could not access field: " + e);
    System.exit(1);
}
```

Updating indexes

The **updateIndex()** method has the following signature:

```
public void updateIndex(Class elementType,
    String path, Object oldKey, Object newKey, Object value)
```

Here is an example of updating an index:

```
Employee lee = new Employee("Lee", 1234);
collection.insert(lee);
lee.setExtension(5678);
collection.updateIndex(
    Employee.class, "extension",
    new Integer(1234), new Integer(5678), lee);
```

Managing Indexes and Index Values

When you add or drop an index, you do it at the class level. That is, you specify the class and member that the index is on. For example, you might add an index on the **name** field of the **Employee** class:

```
employeeCollection.addIndex(Employee, "name")
```

But when you perform maintenance on an index, that is, when you call **removeFromIndex()**, **addToIndex()**, or **updateIndex()**, you do it at the instance level. For example, suppose you have an employee named Jones with an employee ID number of 1234. The employee's name changes to Smith. You must update this index entry at the instance level. One way you can do it is like this:

```
employeeCollection.removeFromIndex(employee1234);
employee1234.setName("Smith");
employeeCollection.addToIndex(employee1234);
```

For each index on the **Employee** class, these methods update the index's value for **employee1234**. If there are multiple indexes on **Employee**, the one-argument overloading of **removeFromIndex()** and **addToIndex()** updates all of them. You do not have to specify that you want to update the index on the **name** field. For example, there might be indexes on the **Employee.salary** and **Employee.location** fields, as well as the **Employee.name** field. The previous code fragment would update the indexes on **salary** and **location**, as well as the index on **name**, even though only the index on **name** needs to be updated. This technique is useful when you make a lot of changes to different fields.

If you use the three-argument overloading of **removeFromIndex()** or **addToIndex()**, you can update just the index that needs to be updated. You must know the type of the indexed element, the name of the indexed member, and the value to be removed or added. For example:

```
employeeCollection.removeFromIndex(
   Employee, "name", employee1234);
employee1234.setName("Smith");
employeeCollection.addToIndex(
   Employee, "name", employee1234);
```

Optimizing Queries for Indexes

If you do not explicitly optimize a query for a particular set of indexes, ObjectStore automatically optimizes the query when it applies the query to a collection. This means that ObjectStore optimizes the query to use exactly those indexes that are available on the collection being queried.

Preparation

Before you optimize a query, you must obtain an instance of **IndexDescriptorSet**. An **IndexDescriptorSet** implements a set of **IndexDescriptor** objects. An **IndexDescriptor** is an object that describes an **IndexMap** on an instance of **IndexedCollection**. Typically, you can obtain an **IndexDescriptorSet** with a call to **IndexedCollection.getIndexes()** on any collection that has exactly the indexes for which you want to optimize your query.

Explicit optimization

To explicitly optimize a query, call the **Query.optimize()** method. the method signature is:

public synchronized void optimize(IndexDescriptorSet indexes)

The **indexes** argument is an instance of **IndexDescriptorSet** that contains **IndexDescriptor** objects that describe the indexes against which to optimize.

Reoptimizing

If you apply an optimized query to the same collection again, or to another collection with the same indexes, ObjectStore uses the same optimization. Reoptimization is not required. However, suppose you apply an optimized query to a collection that does not have all the indexes that were present when the query was first run. In this situation, ObjectStore must reoptimize the query. ObjectStore does this automatically; your intervention is not required.

Manual optimization

Automatic index optimization is convenient, and effective. However, suppose a query is to be run multiple times against more than one collection, potentially with different indexes available. In this situation, it might be best to manually control the query optimization strategy.

For example, consider that the same query is to be run repeatedly against two different collections, where the collections have different indexes. One alternative is to create two separate query objects, one for each collection. This avoids the overhead of recomputing the indexing optimization strategy each time you apply the query to a different collection. A second alternative is to explicitly optimize a query to use only the intersection of the indexes that are available on both collections. You can do this with a call to **Query.optimize()**. Pass in an **IndexDescriptorSet** object that contains descriptions of only the common indexes.

Restriction

If you explicitly optimize a query with the **Query.optimize()** method, it cannot run against a collection that does not have the specified indexes. If you try to do this, ObjectStore throws QueryIndexMismatchException. In this way, an explicitly-optimized query differs from an automatically-optimized query. An automatically optimized query reoptimizes itself as needed when you run it against a collection with different indexes.

This might be useful when it would be undesirable to run a particular query on a collection that does not have the required indexes. For example, this is useful when the collection is very large and the overhead of examining every element of the collection is prohibitive.

-noclassgc option to Java VM

To evaluate query expressions efficiently, ObjectStore compiles query expressions into classes and methods that are loaded when the query is evaluated. Each new query can potentially result in the creation of a new class with a new internal name to represent the compiled state of the query. When the query is no longer referenced, this class is normally garbage collected by the Java VM garbage collector and its storage reclaimed.

With the JDK 1.1.7, you can disable garbage collection of classes with the **-noclassgc** option to the Java VM. If you use this option, you risk running out of heap storage as the query expression classes are accumulated over time and the **-noclassgc** option prevents them from being reclaimed.

Manipulating Indexes Outside the Query Facility

You can use the IndexMap interface to directly access and manipulate indexes outside the query facility. This interface is useful when you want a sorted result set and you can represent the query as a single range expression on an indexed member. Instead of running a query, you can iterate over the index directly. See *ObjectStore Java API Reference*, **COM.odi.util.IndexMap**.

Storing Objects as Keys in Persistent Hash Tables

The **COM.odi.util.OSHashtable** class introduces a new requirement for classes of objects that will be stored as keys in persistent collections: these classes must provide a suitable **hashCode()** method. ObjectStore and the class file postprocessor provide facilities for doing this conveniently.

This section discusses the following topics:

Requirements for Hash Code Methods

Objects that are stored as keys in persistent hash tables must provide hash codes that remain the same across transactions. ObjectStore can create a new transient Java object in each transaction to represent a particular persistent object, so it is important that the **hashCode()** method used for persistent objects return the same hash code for these different transient objects.

The default **Object.hashCode()** method supplies an identity-based hash code. This identity hash code might depend on the virtual memory address or some internal implementation-level metadata associated with the object. Such a hash code is unsuitable for use in a persistent identity-based hash table because it would effectively be different each time an object was fetched in a transaction.

Providing an Appropriate Persistent Hash Code Method

In cases where a persistence-capable class does not override the **hashCode()** method it inherits from **Object**, the class file postprocessor arranges for the class to implement a **hashCode()** method suitable for storing instances in persistent hash tables. It does this by adding an **int** field to the class. This field is initialized to an appropriate hash code when an instance is created and returns the value stored in the field from its **hashCode()** method. This hash code value is guaranteed to remain unchanged for the lifetime of the object.

Applications need to provide their own **hashCode()** methods for classes that define **equals()** methods that depend on the contents of instances rather than on object identity. If the **equals()** method just uses the **==** operator to compare the argument with **this** (or inherits **Object.equals()**), then it is identity-based and the **hashCode()** method provided by the class file postprocessor is appropriate. If the **equals()** method compares the contents of the objects, then it is contents-based and your application must supply a **hashCode()** method that returns the same hash code value for all objects whose contents make them return true when compared with the **equals()** method.

If an application does not need to store instances of a particular persistence-capable class as keys in a persistent hash table, there is no special requirement for that class's **hashCode()** method. In this case, to avoid making all your instances one word larger, have the class define or inherit a **hashCode()** method that calls the superclass's **hashCode()** method:

```
public int hashCode() { return super.hashCode(); }
```

Doing this ensures that the **hashCode()** method inherited from **Object** will be used, which returns a hash code that can be used only in a nonpersistent context.

Storing Built-In Types as Keys in Persistent Hash Tables

You can use the following built-in Java types as **OSHashtable** keys without overriding the **hashCode()** method:

- **java.lang.String**
- Wrapper classes, for example, **Character**, **Integer**, **Long**, **Float**

There is no way to override the **hashCode()** method for arrays. Therefore, do not use Java arrays as keys in persistent hash tables. You can, however, define a class that stores the array as a field and provides an appropriate **hashCode()** method.

Java wrapper classes work nicely as keys because their **hashCode()** methods are based on the value of the object rather than its address.

Using Third-Party Collections Libraries

You can use a third-party Java collections library with ObjectStore. The advantages of doing so are that it might have features that you need or you might be familiar with how to use it. The disadvantage is that it might not scale to the degree that you need.

One third-party library you can use is Doug Lea's collections library. An example of using this is in the **collection** subdirectory of the ObjectStore **demo** directory.

Chapter 8
Automatically Generating Persistence-Capable Classes

This chapter provides information and instructions for using the class file postprocessor to make classes persistence-capable. Reference information for all postprocessor options is in Chapter 13, Tools Reference, on page 377.

Caution

For simple applications, it is best to postprocess all classes together. For more complex applications, you can postprocess your classes in correctly grouped batches. See Postprocessing a Batch of Files Is Important on page 239.

Failure to postprocess the correct classes together can result in problem situations that appear when you try to run the application and that are hard to diagnose. There are postprocessor options that allow you to determine which classes are made persistence-capable.

Contents

This chapter discusses the following topics:

Overview of the Class File Postprocessor

To store an object in a database, the object must be persistence-capable. For an object to be persistence-capable, it must include code that allows persistence. ObjectStore includes the class file postprocessor to automatically insert the required code, referred to as *annotations*, into your class files.

The command you use to run the class file postprocessor is **osjcfp**. The postprocessor provides a number of command options that allow you to tailor the results to your needs.

You can run the postprocessor on classes or class libraries that you create or that you purchase from a vendor. See **COM\odi\demo\collections\README.htm** for an example of making a third-party library persistence-capable.

You must explicitly postprocess or manually annotate each class that you want to be persistence-capable. The capacity for an object to be stored in a database is not inherited when you subclass a persistence-capable class.

When you postprocess or manually annotate a class, this registers the class with ObjectStore. If a class is not postprocessed or manually annotated, ObjectStore throws ClassNotRegisteredException.

This overview provides the following information:

Description of the Annotations

The class file postprocessor annotates classes you define so that they are persistence-capable. This means that the postprocessor makes a copy of your class files, places them in a directory you specify either the source directory or another directory), and adds byte-code instructions (annotations) that are required for persistence. These annotations are

- Modifying the class to implement the **COM.odi.IPersistent** interface.
- Defining methods to initialize instance fields with data from the database, writing modified fields to the database, and resetting instance fields to default values.
- Modifying methods to fetch the contents of persistent instances from the database as needed and to mark modified instances so their changes can be written to the database at transaction commit.

  Before an application can access the contents of a persistent object, it must call the **ObjectStore.fetch()** method to read the object or the **ObjectStore.dirty()** method to modify the object. These calls make the contents of the object available to your application. The postprocessor inserts these calls in methods of classes that it makes persistence-capable or persistence-aware.
- Defining an additional class that provides schema information about the persistence-capable class. This new class is a subclass of the **COM.odi.ClassInfo** class.

Description of the Process

Before you run the postprocessor, you must compile your source files. The set of files you run the postprocessor on can contain a combination of class files, **.zip** files, and **.jar** files. The postprocessor generates annotated class files and places them in a directory that you specify.

This destination directory is never the original directory, unless you specify the **-inplace** option, (see page 383). When you are in a development cycle, it is best to specify a directory other than the original directory. Doing so avoids errors and provides both a persistence-capable and a transient version of the same class.

It is not necessary to recompile all classes before iteratively running the postprocessor. The requirement is that the compiled classes be consistent.

The postprocessor tries to minimize the amount of work it does. It checks file modification times and only reprocesses those files that have changed.

Postprocessing a Batch of Files Is Important

In one execution of the postprocessor, the postprocessor must operate on a correctly grouped set of files. For example, an application might use a file, perhaps a library, that is already annotated. You must not specify the annotated files when you run the postprocessor on the rest of the files in your application. Hence, the term *batch* means all files that the postprocessor must annotate in one execution of the **osjcfp** command. Each batch must have its own postprocessor destination directory for this to work correctly.

You can use the postprocessor **-inplace** option to create multiple batches. When you do, there is no requirement for the separate batches to be stored in different directories.

Example of one batch

When you write a program that uses persistence, the program usually consists of a batch (a set) of classes, for example, classes **A**, **B**, and **C**. They are typically defined in files called **A.java**, **B.java**, and **C.java**. It is possible for each class to reference the other classes. For example, **B** might refer to **C**, and **C** might refer to **B**. There is no ordering or layering; there are no rules for references among the classes.

When this is the scenario, you must run the postprocessor on all of these classes at the same time. You cannot run the postprocessor on each file individually. This is because when the postprocessor operates on **A**, it might refer to **B** and **C**. The postprocessor must have information about **B** and **C** to correctly annotate **A**.

Example of two batches

In relatively simple programs, there is only one batch involved. But sometimes there might be more than one batch in an application. Suppose, for example, that you want to write a persistent program that uses an existing library. An example of this is **djgl**, which is Object Design's persistence-capable version of ObjectSpace's JGL library. Your program consists of **A**, **B**, and **C** plus the JGL library.

Now, in a simple (one-batch) program, when you run the postprocessor, you always specify all files in your application. In this case, you do not want the postprocessor to operate on JGL because it has already been postprocessed. In fact, you probably do not have the class files that have not been postprocessed.

It is correct to run the postprocessor on only **A**, **B**, and **C**. This is because there is a rule: JGL classes never know about **A**, **B**, and **C**. After all, JGL was written, finished, and put on the shelf before **A**, **B**, and **C** were created.

There are two batches here:

- The first batch contains the persistence-capable JGL library. Object Design runs the postprocessor on this batch.
- The second batch contains your own classes, **A**, **B**, and **C**. You run the postprocessor on this batch.

Whenever you run the postprocessor, you must run it on a whole batch. Each batch must have its own postprocessor destination directory.

Checking for correct batches

To determine if you have correctly grouped your files in batches, you can apply this rule: Class **A** and class **B** must be in the same batch if either of the following is true:

- Class **B** inherits from class **A** and either class is persistence-capable.
- Class **A** is persistence-capable or persistence-aware and it directly refers to the fields of class **B**, which is persistence-capable.

Manual Annotation

In exceptional situations, you might want to insert all required annotations needed for persistence and not use the postprocessor at all. See Chapter 9, Manually Generating Persistence-Capable Classes, on page 289. You can also manually annotate your code to meet some persistence requirements and then run the postprocessor to insert the other annotations.

Running the Postprocessor

To make classes persistence-capable, do the following:

1 Compile the source files.

2 Run the postprocessor on the resulting class files.

You must run the postprocessor on all class files in a batch at the same time.

Some Java-supplied classes are persistence-capable. Others are not persistence-capable and cannot be made persistence-capable. A third category of classes can be made persistence-capable, but there are important issues to consider when you do so. Be sure to read Java-Supplied Persistence-Capable Classes on page 360.

The topics discussed in this section are

Preparing to Run the Postprocessor

Before you run the postprocessor, ensure that the following **.zip** files or **.jar** files are explicitly specified in your **CLASSPATH**. An entry for the directory containing them is not sufficient.

- A **tools.zip** or **tools.jar** entry must be in your **CLASSPATH** environment variable.
- The **osji.zip** or **osji.jar** file must be in your **CLASSPATH** environment variable.
- Also, you must update your **PATH** environment variable to contain the **bin** directory from the ObjectStore for Java distribution.

  On Windows, you might set **PATH** to be something like this:

  PATH=c:\jdk117\bin;c:\odi\ostore\bin;c:\odi\osji\bin; c:\winnt\system32;c:\winnt

  On UNIX, it would be something like this:

  PATH=/usr/ucb:/usr/bin:/opt/jdk117/bin:/opt/SUNWspro/bin: /opt/ODI/ostore/bin:/opt/ODI/osji/bin

Requirements for Running the Postprocessor

The postprocessor requires specification of

- The **-dest** option with a destination directory for the annotated class files. This can be an absolute or relative path name. This directory must already exist when you specify it on the postprocessor command line. The postprocessor does not create it.

  You can also specify the -inplace option to instruct the postporcessor to overwrite your original class files. If you do, the **-dest** option is still required.

- A batch of files. A batch includes all files in your application except for already annotated files that your application refers to. You can specify
  - One or more **.class** files
  - One or more **.zip** files
  - One or more **.jar** files
  - One or more class names (you must specify the package names)
  - Any combination of the previous items

Insert a space between specifications and be sure to specify the required destination parameter. When you run the postprocessor, each batch must have its own destination directory. For example:

```
osjcfp -dest osjcfpout COM.odi.demo.threads.Institution Banking.zip Account.class
```

You can specify additional options, which are described in osjcfp: Running the Postprocessor on page 381.

Example of Running the Postprocessor

To make the **Person** class persistence-capable, enter a command like this:

```
osjcfp -dest ..\osjcfpout Person.class
```

This command assumes that the **Person.class** file is in the current directory and the **osjcfpout** directory is a sibling to the current directory. If the postprocessor successfully generates an annotated version of the **Person.class** file, it also generates the **PersonClassInfo.class** file. This file contains information needed by ObjectStore to persistently store instances of **Person**.

The postprocessor places the annotated **Person.class** file and the **PersonClassInfo.class** file in a package-relative subdirectory of the **osjcfpout** directory. For example, suppose the **Person** class package name is **COM.odi.demo.people**. Further suppose that the **osjcfpout** directory is in the **\users\kim** directory. The postprocessor writes the annotated class file to a file whose name is made up of the destination directory, the class package, the class name, and the **.class** extension. It writes the **PersonClassInfo.class** file to a similar location:

```
\users\kim\osjcfpout\COM\odi\demo\people\Person.class
\users\kim\osjcfpout\COM\odi\demo\people\PersonClassInfo.class
```

Note that both of the following commands have the same results, as specified previously.

```
osjcfp -dest ..\osjcfpout Person.class
osjcfp -dest \users\kim\osjcfpout COM.odi.demo.people.Person
```

About the Postprocessor Destination Directory

The postprocessor never overwrites the class files specified on the postprocessor command line, unless you specify the **-inplace** option when you run the postprocessor. If you do specify the **-inplace** option, the postprocessor does overwrite the original class files with the annotated class files.

If you specify a destination directory in such a way that it would store the annotated class file in the same location as the unannotated class file, and you do not specify the **-inplace** option, the postprocessor displays an error message and terminates. It does not produce any class file output.

The postprocessor ignores classes that are rooted in the destination directory. If you try to postprocess a class that exists only in the destination directory (and you do not specify **-inplace**), the postprocessor reports that it cannot find the file. For example, if you specify the following command when you run **osjcfp**, you receive an error as shown.

```
setenv CLASSPATH /usr/devo/java/test:/opt/ODI/osji.zip:/optODI/tools.zip
cd /usr/devo/java/test
javac com/users/jobs/teacher.java
osjcfp -d . com.users.jobs.teacher
...
Error: Class com/users/jobs/teacher could not be found.
```

Because the postprocessor ignores the destination directory in the **CLASSPATH** when it looks up classes, it is unable to locate the specified class. Consequently, the destination directory you specify cannot be the root directory for any of the classes you want to postprocess or any classes referenced by classes you want to postprocess.

Typically, after you run the postprocessor, you have a transient version of a class (your original file) and a persistence-capable version of the class (in the destination directory).

If there are no errors, the postprocessor places a version of all files specified on the command line in the destination directory. The postprocessor annotates those files that require annotations and does not modify those files that do not.

How the Postprocessor Interprets File Names

If a name you specify ends with **.class**, **.zip**, or **.jar,** the postprocessor assumes it is an explicit file name for a class file, **.zip** file, or **.jar** file, respectively.

If a name you specify does not end with **.class**, **.zip**, or **.jar,** the postprocessor assumes it is a class name delimited with periods, for example, **a.b.C**. The postprocessor uses the **CLASSPATH** environment variable or the **-classpath** specification on the postprocessor command line to locate the **.class** file, which can be in a **.zip** file or **.jar** file. (The use of the **-classpath** option does not affect the class path used for the execution of the postprocessor.)

Here is an example of adding the **-classpath** option. It assumes that you are using ObjectStore on UNIX with the JDK installed in **/usr/local/JDK-1.1**.

```
osjcfp -dest osjcfpout -classpath \
/usr/osji:/usr/local/JDK-1.1/java/lib/classes.zip:\
/usr/osji/lib/osji.zip COM.odi.demo.threads.Institution \
Banking.zip Account.class
```

CLASSPATH and **-classpath**

The postprocessor uses the class path you specify in the command line to locate the specified files. This is in place of the **CLASSPATH** environment variable. At run time, Java implementations append the location of the system classes to the end of the **CLASSPATH** environment variable. You must do this manually if you specify the **-classpath** option. This is shown in the examples above as **classes.zip**.

Order of Processing

The postprocessor processes the class files in the order in which they appear on the command line and according to the persistence mode that is in effect when the postprocessor reaches the file name. The persistence mode indicates whether the postprocessor is

- Annotating the class to be persistence-capable
- Annotating the class to be persistence-aware
- Copying the class to the destination directory without annotating it

Persistence mode options

The default persistence mode is that the postprocessor generates persistence-capable classes. Here are the options you can specify to determine the persistence mode:

Persistence Mode Option	*Description*
-pc | **-persistcapable**	Classes specified after this option are made persistence-capable. The postprocessor annotates these classes to include all annotations required by ObjectStore for an object to be persistent. This is the default. If you do not specify a persistence mode option in the postprocessor command line, the postprocessor makes all specified classes and superclasses of those classes persistence-capable.
-pa | **-persistaware**	Classes specified after this option are made persistence-aware. The postprocessor annotates these classes so that they can operate on persistent objects, but instances of these classes cannot themselves be stored persistently.
-cc | **-copyclass**	Classes specified after this option are not annotated. Specify this option for classes that should not be annotated either because they are nonpersistent or already annotated. The postprocessor copies these classes to the destination directory along with the annotated classes.

If you specify a **.class** file or class name, the postprocessor processes it according to the persistence mode that is in effect when the postprocessor reaches the file name. If you specify a **.zip** file or **.jar** file the postprocessor processes all class files in the **.zip** file or **.jar** file according to the persistence mode that is in effect when the postprocessor reaches the name of the. **zip** file or **.jar** file in the command line. For example:

```
osjcfp -dest osjcfpout -persistaware Tent.class Family.class \
-persistcapable Campers.zip Site.class -copyclass Weather.class
```

Example

After you run the postprocessor with the previous command,

- The **Tent** and **Family** classes are persistence-aware.
- The **Site** class, its superclass if it has one other than **java.lang.Object**, all classes in the **Campers.zip** file, and any of their superclasses (other than **java.lang.Object**) are persistence-capable.
- The **Weather** class is not annotated and is copied as is to the destination directory.

How the Postprocessor Handles Duplicate File Specifications

It is permissible for a class to be specified more than once in a command line. For example, a file can be in a **.zip** file and you can also explicitly specify it. Or on UNIX, a file can be included in a wildcard specification and you can also explicitly specify it. In the previous example, the **Family** class could be in the **Campers.zip** file. If it were, the postprocessor would annotate the **Family** class to be persistence-capable. This is because making a class persistence-capable supersedes making it persistence-aware. Likewise, making a class persistence-aware supersedes copying it as is to the destination directory.

If you specify the same class more than once on a command line, both specifications must resolve to the same disk location. For example, suppose you specify both **Person.class** and **COM.odi.demo.people.Person**. This is allowed only if the class path causes **COM.odi.demo.people.Person** to resolve to the same **Person.class** that is explicitly specified.

How the Postprocessor Handles Files Not Found

The postprocessor must be able to find every file that you specify on the command line. If it cannot find one or more files, it displays an error message and stops processing. It does not produce any annotated class files.

Zip and Jar Files as Input to Postprocessor

If a class originates in a **.zip** file or **.jar** file, either because you specify a **.zip** file or **.jar** file when you run the postprocessor or because the class path search locates the class in a **.zip** file or **.jar** file, the postprocessor writes the annotated class to the package-appropriate subdirectory of the destination directory.

How the Postprocessor Handles Previously Annotated Classes

If the postprocessor previously annotated a **.class** file, you can only specify that **.class** file to be copied. You cannot specify it to be annotated. If you do, the postprocessor displays a message that states which specified class was already annotated, and terminates without producing any annotated files.

Troubleshooting OutOfMemory Error

The JDK 1.1 imposes a memory limitation of 16 MB, unless you override it. If you receive a java.lang.OutOfMemory error during postprocessing, you must increase the run-time memory pool. Do one of the following:

- Set the **OSJCFPJAVA** environment variable to include the **-mx** option. For example, Solaris **csh** users can enter

  setenv OSJCFPJAVA "java -mx32m"

  Windows users can enter

  set OSJCFPJAVA=java -mx32m

- Edit the **osjcfp** script (Solaris) or **osjcfp.bat** script (Windows) to incorporate the **-mx** option in the invocation of Java near the end of the script. On Solaris, the line to change is

  $OSJCFPJAVA $javaargs COM.odi.filter.OSCFP $args

  On Windows, the line to change is

  %osjcfpjava% COM.odi.filter.OSCFP %1 %2 %3 %4 %5 %6 %7 %8

  Add **-mx32m** before the **COM.odi.filter.OSCFP** entry. This allows the Java virtual machine to increase the heap to 32 MB. You can increase this value further if you need to.

How the Postprocessor Handles Inner Classes

When you define a class inside another class, you must explicitly make both the outer class and the inner class persistence-capable. For example, suppose you define the following class:

```
Class Foo {
   int a;
   public class Bar {}
}
```

You must specify both the **Foo** class and the **Bar** class when you run the postprocessor:

```
osjcfp -dest ../osjcfpout -pc Foo.class -pc Foo$Bar.class
```

Creating Smaller Annotated Files

To create smaller annotated files, specify the **-optimizeclassinfo** option when you run the postprocessor. This option prevents the postprocessor from generating *xxx***ClassInfo.java** classes for classes that are public or abstract. This reduces the disk footprint and application startup times, since there are fewer classes to load when the application starts.

When the postprocessor does not create a **ClassInfo** class, it uses the Java reflection API instead. Some of the reflection API is subject to security and access constraints that are enforced to varying degrees depending on the version of the JDK and the platform. In other words, you can use the **-optimizeclassinfo** option if the Java environment in which you intend to run the application does not restrict the use of the reflection API.

Managing Annotated Class Files

After you run the postprocessor, there are two versions of your class files:

- The unannotated class files in the original directory
- The annotated class files in the destination directory

It is important to keep these versions separate because

- When you compile source code, you must ensure that any class files the compiler reads in are unannotated class files. The compiler must find unannotated class files before it finds annotated class files with the same names.

  Note: While the above is true for simple applications, it might not be true for more complex applications. See Using the Right Class Files in Complex Applications on page 255.

- When you run your application, you must ensure that ObjectStore finds the annotated class files before it finds the unannotated class files with the same names.

There are several ways to accomplish this. Object Design recommends that you

- Specify the **-classpath** argument to the compiler so it finds the unannotated class files first.
- Modify your **CLASSPATH** environment variable so Java can find the annotated class files first when it runs your application.

To help you manage annotated class files, this section discusses

Ensuring That the Compiler Finds Unannotated Class Files

There are two ways in which the compiler can locate class files:

- Through the **CLASSPATH** environment variable
- Through the **-classpath** argument to the compiler

CLASSPATH is convenient when you compile, but when you try to run your application ObjectStore finds the unannotated files before it finds the annotated files. The **-classpath** option is more cumbersome to use since it means that the path to Java system classes must be listed explicitly in the argument, but it is safe. It ensures that the compiler does not operate on annotated class files.

Example 1

For example, suppose ObjectStore is installed in **c:\osji** and you are building an application in **c:\app**. Your destination directory for annotated class files is **c:\app\osjcfpout**. Your **CLASSPATH** variable might look like this:

```
CLASSPATH=c:\osji\osji.zip;c:\app\osjcfpout;c:\app
```

When you run the compiler, specify the **-classpath** option with the following path. This removes the destination directory from the class look-up path and adds the Java classes to the path.

```
javac -classpath c:\app;c:\osji\osji.zip;c:\jdk117\lib\classes.zip App.java
```

Example 2

Here is an example of why it is important for the compiler to operate on unannotated class files. Suppose you have two classes named **X** and **Y** in the same postprocessor batch. Neither of these classes is explicitly declared to implement **COM.odi.IPersistent**. Now, suppose you add the following two methods to class **Y**:

```
void foo(COM.odi.IPersistent p) {}
void bar() { foo(new X()); } // Trying to pass an X instance to
        // a function that is expecting COM.odi.IPersistent
```

If you recompile only **Y.java** and the compiler finds the annotated classes, examination of the annotated class file allows the compiler to determine that **X** implements **IPersistent**, which allows **Y.bar()** to compile. If you then recompile both **X** and **Y**, the compiler recognizes that **X** is not declared to implement **COM.odi.IPersistent** and refuses to compile class **Y**, even though it successfully compiled earlier.

Ensuring That ObjectStore Finds Annotated Class Files

When you run your application, ObjectStore must find the annotated class files before it finds the unannotated class files. The recommended way to do this is to define a **CLASSPATH** environment variable that has the postprocessor destination directory before the source file directory.

Example

Consider the following example:

- ObjectStore is installed in **c:\odi\osji**.
- You are building an application in **c:\app**.
- You create a directory named **c:\app\osjcfpout** to hold the annotated class files.

In this scenario, use the following **CLASSPATH**:

c:\app\osjcfpout;c:\app;c:\odi\osji\osji.zip

After you modify your **CLASSPATH** environment variable, you can run the postprocessor with no special action. The postprocessor excludes the destination directory from the class path when it does class-path-based searches.

Using the Right Class Files in Complex Applications

There are situations when you want the compiler to read in annotated class files. In these cases, the referenced classes are similar to an independent library on which you are building your application. The referenced classes form a batch, which is a group of class files that must be postprocessed together. The other files in your application form a second batch.

Independent library

For example, suppose this second batch is named **X**. Specify the **-classpath** option so that it points to the

- Unannotated class files for any classes in **X**
- Annotated class files for any classes that are in other batches and are referenced by classes in **X**

This is the most common multiple-batch scenario. Your application is in one batch and the other batches are existing reusable libraries. Each batch has its own postprocessor destination directory.

Classes referenced by other classes

Now suppose that you are not using an existing library. Your application itself contains a group of referenced classes (first batch) and then another group of classes (second batch) that reference the first batch. The following instructions show how to build your application in stages:

1 Compile the source files and postprocess the class files in the first batch. (This is the batch of files that are referenced by other classes in the application.)

2 Compile the source files in the second batch. You might not want to compile all files in this batch at the same time. Specify the **-classpath** option to point to the annotated class files in the first batch and any unannotated class files in the second batch.

3 Run the postprocessor on the second batch. Specify a destination directory that is different from the destination directory that was specified when the postprocessor operated on the first batch. You can package the result of postprocessing the second batch in a **.zip** file or **.jar** file.

Alternatives for Finding the Right Files

In some circumstances, updating your **CLASSPATH** environment variable might be cumbersome or might not work well with your development environment. (This is true for the Symantec Cafe product.) In these cases, you can copy annotated files back to the building directory. However, if you do this, you must remove the annotated files before you recompile. This ensures that subsequent compilation and postprocessing operates on unannotated class files.

Two other alternatives are to

- Delete the contents of the destination directory before you recompile.
- Specify the **-classpath** option when you run the postprocessor, just as you did for compilation.

How the Postprocessor Determines Whether to Generate an Annotated Class File

When you run the postprocessor, it checks if any annotated file it is going to create already exists. If an annotated file does not already exist, the postprocessor generates it. If an annotated file does exist, the postprocessor compares the date on the compiled input file with the date on the annotated output file. If the input file date is after the output file date, the postprocessor generates a new output file. If the input file date is before the output file date, the postprocessor does not generate a new file. It assumes that the annotated file that already exists is still valid.

This works fine when you run the postprocessor repeatedly with the same command line. However, when you change input parameters to the postprocessor, it is a good idea to remove the previously annotated class files from the destination directory. The reason for this is that a comparison of dates might not cause a new annotated file to be generated when the specification of a new input parameter requires a new annotated file to be generated.

To force the postprocessor to overwrite existing annotated files, specify the **-f** or **-force** option when you run the postprocessor.

Creating Persistence-Aware Classes

If you know that a class will never need to be stored persistently, you can run the postprocessor to make the class persistence-aware. A persistence-aware class can operate on persistent objects, but it cannot be persistent itself. For an example of how you might use persistence-aware classes, see **COM\odi\demo\pport\README.htm**.

Persistence-aware annotations require less space than persistence-capable annotations. The postprocessor only adds calls to **ObjectStore.fetch()** and **ObjectStore.dirty()** where they are needed to operate on persistent objects. When the postprocessor makes a class persistence-aware, it does *not* annotate that class's superclass. You only need to make a class persistence-aware, instead of copying it as is, if

- The class accesses fields of a persistence-capable class instead of using methods to access the fields.
- The class accesses elements of persistent arrays.

You must make a class persistence-aware (or persistence-capable) when it includes methods that obtain arrays from persistent objects.

Specifying the Postprocessor Command Line

To create a persistence-aware class, specify the **-pa** or **-persistaware** option followed by the names of the classes that you want to be persistence-aware. For example:

```
osjcfp -dest osjcfpout -persistaware Compute.class
```

The preceding command line annotates **Compute.class** so that it has calls to the **fetch()** and **dirty()** methods.

No Changes to Superclasses

Another reason to make a class persistence-aware is that doing so does not require changing its superclasses. This is important for classes such as **java.lang.Thread**, whose superclass should not be modified. **java.lang.Thread** is inherently transient, so it makes no sense for it to become persistent because it is not useful when you take it out of the database. Typically, Java system classes are restricted from annotations by the postprocessor.

How the Postprocessor Works

This section describes postprocessor behavior relative to various components in your application. It is important to be familiar with the information here so that the postprocessor produces the results you expect. The topics covered in this section are

Ensuring Consistent Class Files

When you run the postprocessor on more than one class file at a time, all specified classes must be consistent. To ensure class consistency, compile all classes together. The postprocessor does not detect inconsistencies among files it operates on. For example, suppose you modify and recompile a class without also recompiling its subclasses. This can cause inconsistencies, which the postprocessor does not detect when it annotates the class files.

Modifications to Superclasses

When you run the postprocessor to make classes persistence-capable, it generates annotated class files for the specified classes and for any superclasses that are in the same packages as the specified classes. ObjectStore requires annotations to superclasses for all classes that the postprocessor makes persistence-capable. If a superclass is not in the same package as one of its subclasses that is being made persistence-capable, you must explicitly specify the superclass on the postprocessor command line.

Effects on Inheritance

If a class that the postprocessor is annotating has no superclass, other than **java.lang.Object**, the postprocessor annotates the class to implement the **COM.odi.IPersistent** interface.

Implementation of the **IPersistent** interface is mandatory for objects that you want to be persistent. You must define classes so that if they inherit from another class, it is a class that can implement **IPersistent**.

Problem with **Object.hashCode()**

Every class inherits from the **Object** class, which defines the **hashCode()** method and provides a default implementation. For a persistent object, this default implementation often returns a different value for the same persistent object (the object on the disk) at different times. This is because ObjectStore fetches the persistent object into different Java objects at different times (in different transactions or different invocations of Java).

This is not a problem if you never put the object into a persistent hash table or other structure that uses the **hashCode()** method to locate objects. If you do put them in hash tables or something similar, the hash table or other structure that relies on the **hashCode()** method might become corrupted when you bring the objects back from the database.

To resolve this problem, you can define your own **hashCode()** method and base it on the contents of the object so it returns the same thing every time. The signature of this method must be

```
public int hashCode()
```

If you do not provide a **hashcode()** method, the postprocessor adds one if it is necessary. If the default behavior of the postprocessor is not ideal for your application, you can specify the **-hashcode** and **-nohashcode** options to control where the postprocessor adds a **hashCode()** method.

Location of Annotated Class Files

When you run the postprocessor, you must specify a destination directory with the **-dest** option. The postprocessor uses the destination directory as the root directory of the class hierarchy of annotated files. The postprocessor places the annotated class file in the package-relative subdirectory of the destination directory. With the destination directory specified in your **CLASSPATH** environment variable, Java can find the annotated classes.

You must create the destination directory before you specify it in an **osjcfp** command line. The postprocessor creates the required subdirectories in the destination directory.

For example, suppose you specify **osjcfpout** as the destination directory. When you run the postprocessor on the **Person.class** file, which is in the **COM.odi.demo.people** package, the postprocessor places the annotated file in

osjcfpout\COM\odi\demo\people\Person.class

The package name of the annotated class file remains the same, unless you specify an option to change it. The class name of the annotated class file is always the same as the class name of the unannotated class file.

Postprocessor Errors and Warnings

If an error occurs while the postprocessor is running, it terminates without writing any annotated class files.

For any warnings from the postprocessor, you might determine that you can safely ignore the warning. In this case, you can stop the postprocessor from warning you about the field in question. To do so, specify the **-quietfield** option followed by the fully qualified name of the field for which you want to suppress warnings. Alternatively, you can specify **-quietclass** to suppress all warnings on the class.

Handling of Final Fields

You cannot make **final** fields persistent. If you try to do this, the postprocessor displays a warning message and treats the fields marked as **final** as though you declared them to be **transient**. To allow such fields to be stored persistently, you must remove the **final** keyword.

Handling of Static Fields

The postprocessor never stores static fields in the database and never causes the values of static fields to be altered. You must write your own code to update static fields and to store static fields in the database, if that is what you want to do.

Static fields that can hold persistent values

The postprocessor displays a warning for a static field that can hold potentially persistent values. The postprocessor cannot determine the type of the object that will actually be pointed to. Consequently, depending on the type of object referenced, the warning might not be applicable. For example, suppose you have a persistence-capable class named **X**. The **X** class has a static member named **y** of a type that implements **COM.odi.IPersistent**. When you run the postprocessor, it displays a warning like this:

```
X.y is a static field of a type that implements COM.odi.IPersistent and that might refer to a persistent object. If this field does refer to a persistent object it must be user maintained.
```

Referring to a persistent object

If the field mentioned in the warning is intended to refer to a persistent object, you can write your application as follows:

- When you create a database, create an object of the desired type and create a database root to refer to the object. This makes the object persistent and provides a mechanism you can use to find the object.
- When you start a new transaction, if the object referenced by the static field is null or stale, look up the database root and set the value of the static field to the root value.

If you specify **ObjectStore.RETAIN_STALE** when you commit or abort a transaction, you must ensure that you correctly access the objects at the beginning of the next transaction. This is because ObjectStore does not make the object referenced by **X.y** persistent if it is only reachable from **X.y**. If ObjectStore makes it persistent because it is reachable from some other point, the object referenced by **X.y** might become stale at the end of the transaction in which it becomes persistent. If it does, and if the object referenced by **X.y** does become persistent, it is possible that the application might try to use the stale version of the object.

How can **X.y** be reachable from some other point? Perhaps some other persistent object or an object that is going to be persistent refers to the object that the static data member is referring to.

When ObjectStore commits the transaction and performs transitive persistence, it finds the object that the static data member is referring to.

References to stale objects

An issue to consider is stale references to stale objects. To avoid the inadvertent use of stale objects, update **X.y** at transaction boundaries. Set **X.y** to null or to some other value to ensure that if a stale object is referenced by **X.y**, it is no longer accessible through **X.y**. Then you can suppress the warning with the **-quietfield** option.

Summary

For class **X**, the important points are listed below.

```
class X { static OSHashtable y = new OSHashtable(); }
```

- **X.y** does not become persistent just because class **X** is persistence-capable or because an instance of **X** becomes persistent.
- If you want **X.y** to become persistent, you must make it reachable from a root through a path that does not involve a static field, for example, **db.createRoot("X.y", X.y)**.
- If **X.y** does become persistent, you must be aware that the **OSHashtable** object referenced by **X.y** might become stale at transaction boundaries. If it does, you must update **X.y** to refer to a nonstale instance.

Which Java Executable to Use

The postprocessor is a Java program; it requires a Java virtual machine to run. It uses the first Java executable that it finds in your **PATH** environment variable. If you want the postprocessor to use some other Java executable, set the **OSJCFPJAVA** environment variable to the name of the Java executable you want the postprocessor to use. The default is **java**.

If the postprocessor cannot find a Java executable, it generates a Bad command or file name error message.

Line-Number and Local-Variable Information

When the postprocessor annotates a class file, it maintains any existing line-number and local-variable information.

Using a Debugger

The class file postprocessor annotates methods with VM instructions for automatically performing **fetch()** and **dirty()** operations on objects. It does this in such a way that the debugging information in the class files remains intact. For the most part, the annotations are invisible to an application. However, it is possible to encounter them under certain circumstances when using a debugger. For example, you might encounter the following when you use the **Step into** command:

```
x = foo(y.m);
```

Stepping into that statement might cause you to enter the ObjectStore code that causes the contents of the **y** object to be fetched. In such a situation, use the **Step out** command to leave the ObjectStore code. Then use the **Step into** command again, which should then step into the call to the **foo()** method.

You should rely on the **Step over** command whenever possible. However, there are situations where you must use the **Step into** command. If you inadvertently step into an ObjectStore method, step out of the ObjectStore code and return to your own code by doing one of the following:

- Use a **Step out** command.
- Set a breakpoint in the calling code.
- Use repeated **Step over** commands until the method returns.

Handling of finalize() Methods

The Java garbage collector calls the **java.lang.Object.finalize()** method on an object that is no longer referenced. The garbage collector does this before it frees the space occupied by the object. In this way, the **finalize()** method provides a hook that you can use to free resources that are not freed by garbage collection, for example, memory that was allocated by a native method call.

If your persistence-capable class defines a **finalize()** method (Object Design recommends that it should not), the class file postprocessor inserts annotations at the beginning of the **finalize()** method that change the persistent object to a transient object. See Avoiding finalize() Methods on page 182.

Description of Postprocessor Optimizations

The postprocessor optimizes **fetch()** and **dirty()** calls in several ways. If you determine that an optimization is preventing insertion of a required call to **fetch()** or **dirty()**, you can disable the optimization.

- For array objects in looping constructs, the postprocessor inserts the call to **fetch()** or **dirty()** only in the first loop iteration. To disable this optimization, specify the **-noarrayopt** option when you run the postprocessor. This causes the postprocessor to insert calls to **fetch()** or **dirty()** in every iteration.
- For constructors, the postprocessor does not insert full annotations that would allow constructors to handle modifications to newly constructed objects. To disable this optimization, specify the **-noinitializeropt** option when you run the postprocessor. This causes the postprocessor to fully annotate constructors so that they can correctly handle modifications to objects that become persistent during constructor execution.

  If your application inserts objects into ObjectStore collections during construction of the objects being inserted, you must specify the **-noinitializeropt** option. Doing so avoids errors in the handling of modifications to the newly constructed objects.
- For access to fields relative to **this** in nonstatic member methods, the postprocessor optimizes calls to **fetch()** and **dirty()**. To disable this optimization, specify the **-nothisopt** option when you run the postprocessor. This causes the postprocessor to insert a **fetch()** or **dirty()** call for each access to a field in **this**.

You should disable these optimizations if you commit transactions or evict persistent objects as follows:

- If you call **commit()** or **evict()** while iterating over persistent array elements, specify **-noarrayopt** when you run the postprocessor.
- If you call **commit()** or **evict()** in between accesses to different fields of **this**, specify **-nothisopt** when you run the postprocessor.

If you want to disable all three optimizations, specify the **-noopt** option instead of the three individual options.

Including Transient and Already Annotated Classes

After you run the postprocessor, the annotated class files are in the package-relative subdirectory of the destination directory (root directory) you specified. You might want other class files in this destination directory. These could be transient (nonpersistence-capable or nonpersistence-aware) class files or files that have already been annotated.

Copying Classes to the Destination Directory

To copy some files to the destination directory along with the annotated files, specify the **-copyclass** option followed by the name of the file you want to copy. For example:

```
osjcfp -dest osjcfpout a.zip -copyclass b.class
```

In this example, the postprocessor annotates the files in **a.zip** and copies them to the package-relative subdirectory of the **osjcfpout** directory. The postprocessor also copies **b.class** to the **osjcfpout** directory, but it does not modify the **b.class** file.

You can follow the **-copyclass** option with one or more **.class** file names, class names, **.jar** file names, or **.zip** file names. This option applies to each name that follows it, until the postprocessor reaches a **-pc** or **-pa** option.

Specifying Classes to Be Copied and Classes to Be Persistence-Capable

Classes for which you specify the **-copyclass** option can overlap with classes for which you specify the **-persistcapable** or **-persistaware** option. For example:

```
osjcfp -dest osjcfpout -copyclass *.class -persistcapable a.class
```

This allows you to keep all files in a package together and only annotate the classes that need to be annotated. You need not partition classes into those that need annotations and those that do not. You can specify the same file with more than one persistence mode option because the **-persistcapable** option and the **-persistaware** option override the **-copyclass** option.

When Can a Class Be Transient?

Suppose you have a persistence-capable class, class **A**. A class that refers to class **A** can be transient if all access to **A**'s nontransient data members is through methods on **A.** The methods of **A** will be properly annotated. Since all other classes only use **A**'s methods, the other classes do not need to be persistence-aware. Consequently, you do not need to postprocess any classes that refer to **A**.

Any class that directly accesses **A**'s nontransient data members must be either persistence-capable or persistence-aware. Any other class that refers to **A** and does not directly access nontransient data members can be transient. That is, you do not have to postprocess it.

An important exception to this is that if a class manipulates an array object that might be persistent (specifically, setting and getting array elements), that class must be annotated to be persistence-aware. However, if the code that provides access to the array is annotated to access the values of the array, you can avoid making the class persistence-aware. It is difficult to reliably implement this in the general case.

If you compile with optimization the classes that use the methods that get and set array values, the compiler might inline the get and set methods. In this case, you must make the class that uses the get and set methods persistence-aware.

Putting Processed Classes in a New Package

Normally, the postprocessor places the annotated files in a package-relative subdirectory of the destination directory and the annotated files have the same package names as the original files. However, there is an option that allows you to change the package name of files specified in the postprocessor command line. The **-translatepackage** option modifies the package name so that the persistence-capable version of the class is in one package and the transient version (the original) is in another package.

To help you use the **-translatepackage** option, this section discusses the following topics:

Using the -translatepackage Option

To create persistence-capable classes whose package name is different from the original package name, specify the **-translatepackage** option followed by the current package name and then the new package name. The format for this option is

{ **-translatepackage** | **-tp** } *orig_pkg_name new_pkg_name*

For example, suppose you have the **a.b.C** class and you want to create the **d.e.C** persistence-capable class. Run the postprocessor like this:

```
osjcfp -dest osjcfpout -translatepackage a.b d.e C.class
```

Exact match required

The specification for the original package name must exactly match the package name of the specified file. If there is not an exact match, the postprocessor does not place the annotated file in the new package. For example, suppose you have two classes named **COM.odi.demo.New** and **COM.odi.Old**. You want to move **COM.odi.Old** to the **COM.odi.beta** package and you specify the following command:

```
osjcfp -dest osjcfpout -tp COM.odi COM.odi.beta
COM.odi.demo.New COM.odi.Old
```

The postprocessor places the annotated file for the **COM.odi.Old** class in **COM.odi.beta.Old** in the package-relative subdirectory of the **osjcfpout** directory (**osjcfpout\COM\odi\beta\COM.odi.beta.Old.class**).

The postprocessor does not place the annotated file for **COM.odi.demo.New** in a different package because the original package name is **COM.odi.demo** and not just **COM.odi**. The postprocessor annotates **COM.odi.demo.New** and places it in **osjcfpout\COM\odi\demo\COM.odi.demo.New.class**.

How the Postprocessor Applies the Option

The postprocessor applies the **-translatepackage** specification to

- All classes in the original package that are locatable by means of the **CLASSPATH** environment variable or the **-classpath** option if you specify it. The **-classpath** specification overrides the **CLASSPATH** environment variable.
- Files on the command line whose package name exactly matches the specification for the original package name. This is true for files processed with the **-persistcapable**, **-persistaware**, or **-copyclass** option.

When copying files

It does not matter whether the postprocessor is making any other changes to the specified files. The postprocessor changes the package names of files for which the **-copyclass** option is specified right along with new persistence-capable or persistence-aware files.

Multiple option specifications

You can specify this option more than once on a command line to specify several package translations. If you accidentally specify more than one translation for the same package, the postprocessor performs the last translation you specify in the command line.

Updating References to New Package Name

A change to the package name of a class requires updating all references to that class to reflect the new name.

The postprocessor updates the references in classes that it is currently operating on. This includes each class specified on the command line and each class found in a**. zip** file or **.jar** file that is specified on the command line.

The postprocessor cannot detect if there are **.class** files for which the postprocessor was not called that refer to the renamed package. You must either run the postprocessor on the complete set of class files or modify the Java source of any files that the postprocessor is not annotating.

References to Transient and Persistent Versions of a Class

You might want a class to refer to both the transient and persistence-capable versions of some other class.

It is not possible for the postprocessor to determine which references should be to persistence-capable objects. Because of this, you must code the class so it uses the full path name of the different versions of the class. This is the only way to clarify which version of the class is wanted. However, this technique works correctly only when you are operating across batches. It does not work when you are within the same batch.

Example

Here is an example of what that means. Suppose you have a utility class called **a.b.C**. You want to have both a transient and a persistence-capable version of **a.b.C**. When you run the postprocessor, you specify **-translatepackage** to create a persistence-capable version called **y.z.C**. Then, in another class called **a.b.D**, you try to use both versions of the class. You write source code in **a.b.D** that explicitly refers to **y.z.C**, something like

```
int n= y.z.C.countThem()
```

When you try to compile **a.b.D**, compilation can succeed only if you put the annotated classes into the class path of the compiler. Otherwise, the compiler reports an error, because there is no such thing as **y.z.C**.

Also, it is not possible for **a.b.C** and **a.b.D** to be in the same batch, because the **-translatepackage** option would apply to **a.b.D**. This would make all of **a.b.D**'s calls go to the persistence-capable version, which is not what you want.

Steps to follow

To use persistence-capable and transient versions of the same class, follow these steps:

1 Create a utility library.

This is the first batch. This library creates transient versions of the class.

2 Run the postprocessor on the first batch and specify options that put the two different versions of the class in two different packages.

This step creates the persistence-capable version of the class.

3 Use the library from an application.

The application is the second batch.

4 Compile the application with the annotated files of the first batch, but not the second batch, in the compiler's class path.

References to Transient Instances of a Persistence-Capable Class

You can use instances of a persistence-capable class in a transient-only fashion. No special action is required and the calls to **ObjectStore.fetch()** and **ObjectStore.dirty()** do nothing.

There is no need for the unannotated version of the class to be available at run time.

To use the annotated version of the class, even if you are using it transiently, the **osji.zip** or **stublib.zip** file must be available in the **CLASSPATH** at run-time. If you are only using the class transiently, it can be the **stublib.zip** that is available.

Creating Persistence-Capable Classes with Transient Fields

You can create a persistence-capable class with transient fields. A transient field is a field that is not stored in the database. The postprocessor ignores transient fields. Use the **transient** keyword to create a transient field. For example:

```
class A {
    transient java.awt.Component myVisualizationComponent;
    int myValue;
    ...
}
```

In this class, the **myVisualizationComponent** field is declared to be a transient reference to **java.awt.Component**. **java.awt** is a package that contains GUI classes that do not lend themselves to being persistence-capable.

In your persistence-capable class, you might have transient fields that you want to be able to access outside a transaction. In this situation, you can specify the **-noannotatefield** or **-naf** option for the field when you run the postprocessor. This option prevents access to the specified field from causing **fetch()** and **dirty()** calls on the containing object. Normally, access to a transient field causes **fetch()** or **dirty()** to be called to allow the **postInitializeContents()** and **preFlushContents()** methods to convert between persistent and transient state.

Transient Fields and Serialization

If you have a class that has fields that are declared as transient, this causes the default handling of these fields by object serialization to be to ignore the fields. If you want them ignored by object serialization and you also want them to be stored persistently, specify the **-ignoretransient** option for the class when you run the postprocessor.

On the other hand, there might be a field that must be available for object serialization, but you do not want to store that field in the database. In this situation, specify the **-transientfield** option for the field when you run the postprocessor. This option causes the postprocessor to treat the specified field as though it has a **transient** modifier, even if it does not.

Initialization of Some Transient Fields

In the declaration of a transient field in a persistence-capable class, you might want to initialize the value of the transient field. However, when the postprocessor creates the hollow object constructor for the class, it does not define the constructor to initialize the transient field. This is true even when you specify the **final** keyword. The postprocessor does not initialize such fields, because the initialization occurs as inlined code in each of the constructors for the class. For example:

```
private transient final MyField myField = new MyField();
```

The **final** keyword indicates to the postprocessor that initialization is required. However, the initialization code is not readily available and **myField** is not initialized. There are several ways to handle this situation.

You can create the hollow object constructor manually. For example, suppose you define the **MyField** class, which extends the **MyFarm** class:

```
...
public MyField(COM.odi.ClassInfo dummyClassInfo) {
    super(dummyClassInfo);
}
```

This requires you to also manually define a hollow object constructor for the **MyFarm** class, and for each superclass of the **myFarm** class.

Alternatively, you can remove the **final** qualifier and initialize the transient field in an **IPersistent.postInitializeContents()** method.

If you include an inline initialization of a field declared to be **transient** and **final**, the postprocessor displays an error message and stops processing. If you include an inline initialization of a field declared to be **transient**, but not **final**, the postprocessor warns you about the situation and continues processing. If you determine that you can safely ignore the message, you can turn it off with the **-ignoretransient** option to the postprocessor.

See also Transient Fields in Persistence-Capable Classes on page 180.

Customizing Updated Classes

There are several ways you can customize persistence-capable and persistence-aware annotations. You can implement your own versions of methods that the postprocessor typically adds. You can implement hook methods that ObjectStore calls at specified points. You can define a hollow object constructor in place of the hollow object constructor the postprocessor typically defines. You can also insert your own **fetch()** and **dirty()** calls.

Implementing Customized Methods and Hook Methods

The three methods described below are among the several annotations that the postprocessor adds to persistence-capable classes.

- The **initializeContents()** method loads real values into hollow instances of your persistence-capable class. In other words, hollow objects become active objects with an internal clean state.
- The **flushContents()** method copies values from a modified instance (active persistent object) back to the database. This changes the internal clean or dirty state of the persistent object to the clean state.
- The **clearContents()** method resets the values of an instance to the default values. This changes a clean active object to a hollow object.

Alternatives

If you want to, there are two ways that you can customize the behavior of these methods:

- Implement the method yourself. See Defining Required Methods in the Class Definition on page 292. If you do, the postprocessor does not add the method. However, if you implement any of the three methods listed above, you must implement all of them. Also, you must define the **ClassInfo** subclass, define an instance of it, and register the instance. This is because the **ClassInfo** instance and the three above methods must agree on the conventions for field numbering. An example of a program that implements these methods is in the **COM/odi/demo/rep** directory in the **Rectangle.java** file. See **COM/odi/demo/rep/README.htm.**

- Implement the hook method that corresponds to the method you want to customize. The postprocessor does not annotate hook methods. These hook methods provide a way to perform transient field maintenance. You might also be able to use these methods as an update mechanism for notification about a change:
  - **postInitializeContents()** — If you define this method, ObjectStore calls it immediately after it calls the **initializeContents()** method.
  - **preClearContents()** — If you define this method, ObjectStore calls it just before it calls the **clearContents()** method.
  - **preFlushContents()** — If you define this method, ObjectStore calls it just before it calls the **flushContents()** method.

Warning

The body of a hook method must not call any methods of the class and must not start or end a transaction. This is because the class methods are annotated and consequently make calls to **fetch()** and **dirty()**. Such calls in the middle of initializing or writing the object are not allowed because they might cause the virtual machine to encounter a stack overflow.

Sample program with hook methods

Here is an example of a program that implements these hook methods.

```
import COM.odi.*;

/**
* PColor provides a persistent representation of colors that can be
* used with the Java AWT package. The java.awt.Color class itself
* cannot be stored persistently, because some of its internal state
* depends on the particular kind of color display being used. If a
* java.awt.Color were created on a computer that used a 24-bit-deep
* color monitor, stored in a database, and then retrieved and used
* on a different computer that had a gray-scale monitor, it would not
* function correctly. PColor stores the color value as three
* integers, and then recreates the java.awt.Color object whenever
* the PColor object is brought into Java from persistent storage.
*
* For expository purposes, this example pretends that the value of a
* java.awt.Color object can change after the object is created. The
* real java.awt.Color class is immutable, and so the setBlue method
* below would not work, and the preFlushContents method would
* not actually be needed.
*/

public class PColor {

/*These instance variables are stored persistently. They represent
    the color value. */
    int red;
    int green;
    int blue;

/*This instance variable is declared transient, so it is not stored
persistently. It is managed by the methods below. */

transient java.awt.Color color;
    PColor(int r, int g, int b) {
        red = r;
        green = g;
        blue = b;
        color = new java.awt.Color(r, g, b);
    }
/*When a PColor is brought into Java from persistent storage, the
    java.awt.Color object is created. Note that this method runs
    after the initializeContents, so that it can use the values
    of the persistent instance variables. */

    public void postInitializeContents() {
        color = new java.awt.Color(red, green, blue);
    }
```

```
/*When a PColor is sent out from Java to persistent storage, the
color value from the java.awt.Color object is copied into the
persistent instance variables, so that it will be saved.
Note that this method runs before flushContents, so that it
can set up the values of the persistent instance variables. */

    public void preFlushContents() {
        red = color.getRed();
        green = color.getGreen();
        blue = color.getBlue();
    }
/*When clearContents happens, this method sets the color
instance variable to null, so that this PColor object won't be
stopping the java.awt.Color object from being reclaimed. */

    public void preClearContents() {
        color = null;
    }
/*Equality for PColor objects is the same as equality of the
underlying java.awt.Color objects. */

    public boolean equals(Object obj) {
        if (obj instanceof PColor) {
            return color.getRGB() == ((PColor)obj).color.getRGB();
        }
        return false;
    }
    public java.awt.Color getColor() {
        return color;
    }
    public int getBlue() {
        return color.getBlue();
    }
    public int setBlue(int b) {
        color.setBlue(b);
    }
    /* and so on.... */
}
```

Creating a Hollow Object Constructor

For each persistence-capable class, the postprocessor finds or generates a hollow object constructor. The hollow object constructor takes a single argument whose type is **COM.odi.ClassInfo**. Typically, you do not need to define a hollow object constructor, but you can if you want to.

Why define one?

A reason to define your own hollow object constructor is to initialize transient fields that you want to be usable even if the **fetch()** method has not been called.

You should avoid performing actions in a hollow object constructor that would cause the object to be fetched. Doing so might cause infinite recursion to occur.

For example, if a class has a persistent **hashCode()** method, it is a bad idea to define a hollow object constructor to register the instances of the class in a hash table. Doing so would cause the **hashCode()** method to be called, which in turn would attempt to fetch the object.

Creation steps

When the postprocessor creates the hollow object constructor, it follows these steps:

1 The postprocessor selects an appropriate superclass hollow object constructor.

   If the superclass has an accessible constructor that takes a single **COM.odi.ClassInfo** argument, or if it will have one because the postprocessor will add it during this execution of the tool, the postprocessor uses that constructor. The postprocessor reports an error if it cannot find an accessible constructor.

2 The postprocessor creates a public constructor that

   - Accepts a **COM.odi.ClassInfo** argument
   - Invokes the selected superclass constructor
   - Initializes all persistent fields to an appropriate default state that is equivalent to the result of the **clearContents()** method

You can define the hollow object constructor instead of allowing the postprocessor to do it. If you define one, the postprocessor does not generate one.

Optimizing Operations That Retrieve Persistent Objects

Before an application can access the contents of a persistent object, it must call the **ObjectStore.fetch()** method to read the object or the **ObjectStore.dirty()** method to modify the object. These calls make the contents of the object available to your application. The postprocessor inserts these calls in methods of classes that it makes persistence-capable or persistence-aware. However, the postprocessor might not annotate your code for best performance. You might find that you can improve performance by inserting the **fetch()** and **dirty()** calls yourself.

Caution

If you insert a **fetch()** or **dirty()** call in a method, the postprocessor does not add any additional **fetch()** or **dirty()** calls to that method.

Procedure for Optimizing Operations

Before you add the calls yourself, first allow the postprocessor to add the **fetch()** and **dirty()** calls. Then run and monitor your program. If you want to try to improve performance, add the calls to your source file and recompile. When you run the postprocessor again, it recognizes that the **fetch()** or **dirty()** call is already in place and does not add any **fetch()** or **dirty()** calls to any methods that already contain such a call.

If you do this annotation, you should also add **implements IPersistent** to the definition of any class that is accessed with a **fetch()** or **dirty()** call. When you do this, the compiler can effectively use the multiple overloadings of the **fetch()** and **dirty()** methods, which take **COM.odi.IPersistent** arguments. Also, the compiler can generate more efficient code when you declare the class to implement **IPersistent** in your source.

Inlining Code

An important consideration when annotating by hand is that the compiler might inline the code into calling methods. This makes it appear to the postprocessor that the code annotations are in the calling method, which might not be true.

When using the JDK **javac** compiler, this occurs when you specify the **-O** (capital O, as in Oslo) option.

To ensure that the postprocessor functions correctly, you must do one of the following:

- Prevent the compiler from inlining code.
- If you add **fetch()** and **dirty()** calls to a method that is a candidate for inlining, then also annotate all the methods that call that method. A method is a candidate for inlining if it calls **static**, **final**, or **private** methods, or invokes methods with the **super.** qualification construct.

Preventing Fetch of Transient Fields

You might want to avoid the insertion of the **fetch()** call in methods that operate only on transient fields. A strategy for doing this takes advantage of the fact that the postprocessor does not annotate a method if it already includes a **fetch()** or **dirty()** call. If you know that a method operates on only transient fields, you can prevent insertion of the **fetch()** call with code such as the following:

```
try {
    method body goes here
}   catch (SomeRuntimeExceptionThatWillNotOccur) {
        ObjectStore.fetch(this);
}
```

This imposes no execution time and prevents the postprocessor from inserting the **fetch()** method. You can create your own exception, which inherits from java.lang.RuntimeException, or select an existing one. The safest approach is to create your own exception so that you can be sure that the exception is never thrown.

Specifying the Number of Array Dimensions in Persistence-Capable Classes

By default, three is the maximum number of dimensions in a persistent array. If you need a persistent array that has more than three dimensions, you can run the postprocessor with the **-a** or **-arraydims** option followed by an integer, which specifies the new maximum number of array dimensions. This option applies to all classes that the postprocessor annotates during this execution of the tool. It does not apply to a class that the postprocessor does not annotate. For example:

```
osjcfp -dest osjcfpout -a 4 Person.class Pet.class -copyclass
Car.class
```

This allows arrays of type **Person** and **Pet** to have as many as four dimensions. The maximum number of array dimensions does not change for the **Car** class because the postprocessor does not annotate that file, it only copies it to the destination directory.

An alternative, and far more complex, way to increase the number of allowed array dimensions is to manually implement the class of the **ClassInfo** instance associated with the persistence-capable class. See Defining a ClassInfo Subclass on page 295.

Performing a Test Run of the Postprocessor

You can run the postprocessor without actually updating any files. The tool performs all processing and error checking and can display messages that indicate what it is doing. This allows you to make corrections before creating the persistence-capable versions of your classes.

To perform a test run of the postprocessor, specify the **-nowrite** option on the command line. For example:

```
osjcfp -dest osjcfpout -nowrite classes.zip
```

This command processes all class files in the **.zip** file and displays any error messages. To view information messages from the postprocessor, include the **-verbose** option. For example:

```
osjcfp -dest osjcfpout -nowrite -verbose classes.zip
```

It does not matter where you place the **-nowrite** or **-verbose** option in the command line. Wherever you place them, they apply to all files that the postprocessor processes.

To suppress nonfatal warning messages, specify the **-quiet** option. The **-quiet** and **-verbose** options are mutually exclusive. The last one used on the command line applies to the entire execution. For example, the following line suppresses warning messages during the processing of all specified files because the **-quiet** option comes after the **-verbose** option.

```
osjcfp -dest osjcfpout -nowrite -verbose classes.zip -quiet more.zip
```

You can also suppress some warnings but not all warnings. Specify the **-quietclass** option followed by the fully qualified name of a class to suppress warnings for that class. Specify the **-quietfield** option followed by the fully qualified name of a field to suppress warnings that pertain to that field. These options apply only to the element whose name immediately follows the option. If the **-verbose** option is also specified, these options take precedence.

Using an Input File

When you are running the postprocessor on a lot of files and specifying many options, the command line can be very long. As a convenience, you can enter the options and file names in a file and then specify the file name as a postprocessor option. Be sure to prefix the file name with the @ symbol.

Windows

On Windows systems, there is a limit of eight arguments on a command line. Consequently, you usually must use input files on Windows.

Format

You can include comments in the input file. You can place items on different lines and line continuation symbols are not required. Line breaks are treated as white space. Otherwise, enter data in the input file exactly as you would enter it on the command line.

Indicate comments with a # sign. The postprocessor ignores any subsequent characters on the same line as the # sign.

Example

For example, suppose you enter some postprocessor options and files for the postprocessor to operate on in an input file named **optionsAndFiles**. You specify this file as follows:

```
osjcfp @optionsAndFiles
```

You can intersperse input file specifications with options and files that you enter on the command line. For each specified input file, the postprocessor removes any comments from the input file and replaces the input file specification with the data in the input file. The postprocessor then begins to process the command line. For example:

```
osjcfp -dest osjcfpout @file1 -tp old.pack new.pack @file2
```

The postprocessor

1. Replaces **@file1** with the contents of **file1**.
2. Replaces **@file2** with the contents of **file2**.
3. Executes the command line starting with the **-dest** option.

Nesting and wildcards

You cannot nest input file specifications. That is, you cannot include the **@***file_name* option in an input file. Also, you cannot use wildcards in an input file. The postprocessor does not expand them.

Annotations You Must Add

There are some annotations that the postprocessor either cannot perform or does not perform because of execution performance considerations. You must include these annotations when you code your source files.

Keep in mind that when you add even one **fetch()** or **dirty()** call to a method, the postprocessor recognizes that the method is already annotated and does not add any other **fetch()** or **dirty()** calls to that method. If you do annotate a method, be sure to add all required calls.

This section provides information about the following topics:

Interfacing with Nonpersistent Methods

It is possible for a method in a persistence-capable class to pass a persistent object to a nonpersistent method. When this happens, you must ensure that there is a **fetch()** or **dirty()** call for the persistent object before it is passed to the nonpersistent method.

If all access to persistent objects is through annotated methods (methods in persistence-capable or persistence-aware classes), then manual annotations are not required. For arrays, there is no way to define a class so that arrays of that class can only be accessed by persistence-aware classes. You must be sure to call the **fetch()** or **dirty()** method on a persistent array before passing it to a method in a nonpersistent class.

Interfacing with Native Classes

The postprocessor cannot analyze or annotate native methods. If your code passes a persistent object to a native method, and if the native code might try to access the object other than through annotated methods, be sure to insert a call to **fetch()** or **dirty()** for the persistent object before it is passed. In cases where native code might access and / or navigate among persistent objects, you must do one of the following:

- Modify the native code to call **fetch()** or **dirty()** itself.
- Make the necessary **fetch()** and **dirty()** calls before calling the native method.

Annotating Subclasses

After you create a persistence-capable or persistence-aware class, you can define a subclass of that class. Doing so does not make the subclass persistence-capable or persistence-aware. You must run the postprocessor on the subclass.

If you forget to run the postprocessor on a subclass and if the subclass is reachable from a persistent root (other than through a transient field), ObjectStore might try to migrate instances of the subclass to the database. This attempt causes an error because the subclass is not persistence-capable.

Passing Arrays

In your application, you might pass an array to a nonpersistent method when the nonpersistent method is defined as having a parameter of type **java.lang.Object**. In this situation, the postprocessor cannot determine that it should insert **fetch()** or **dirty()** calls for the array in the calling method before passing the array. You must annotate the calling method yourself.

If the called method is declared to accept an array argument, the postprocessor recognizes that a **fetch()** call might be needed and inserts it.

Implementing the Hollow Object Constructor for Some Instance Fields

A class can include nonstatic (instance) fields that contain initializer expressions in their declarations. Postprocessor-generated **ClassInfo** constructors do not run these initializers. Normally, this is not a problem. The constructor allows hollow object initialization and the **initializeContents()** method overwrites these fields when the object is fetched.

However, there might be transient nonstatic fields that have initializer expressions or fields that are treated as transient by your implementation of the **ClassInfo** type and the **initializeContents()** and **flushContents()** methods. In this case, you must manually implement the hollow object constructor or ObjectStore does not run the initializer. It is impossible for the postprocessor to detect such cases, and no warning message can be provided. See Creating a Hollow Object Constructor on page 279.

Using the Java Reflection API with Persistence-Capable Objects

You can use the **java.lang.reflect.Field** class to get and set fields of persistence-capable objects. To do so, you must

- Call the **ObjectStore.fetch()** method for an object before you call any of the **java.lang.reflect.Field** *get* methods to get the value of any of the object's fields.
- Call the **ObjectStore.dirty()** method for an object before you call any of the **java.lang.reflect.Field** *set* methods to set the value of any of the object's fields.

Class File Postprocessor Limitations

It is possible to cause invalid references when you run the postprocessor and rename the package. In an annotated class, the postprocessor locates and updates class names if they are in field, method, or class references. The postprocessor cannot locate and update string arguments to **Class.forName()** if the name specifies a class whose package has been renamed.

Chapter 9
Manually Generating Persistence-Capable Classes

This chapter provides information about how to explicitly define persistence-capable and persistence-aware classes in your program without using the automated class file postprocessor supplied with ObjectStore. Object Design recommends that you use the automated postprocessor. See Chapter 8, Automatically Generating Persistence-Capable Classes, on page 235.

However, you might choose the manual method if you want to

- Manually optimize the code
- Perform translation between nonpersistent objects and a custom persistent representation

You can partially manually annotate a class and then run the postprocessor to insert the remaining required annotations.

You must explicitly postprocess or manually annotate each class that you want to be persistence-capable. The capacity for an object to be stored in a databases is not inherited when you subclass a persistence-capable class.

Contents

This chapter discusses the following topics:

Explicitly Defining Persistence-Capable Classes

Follow these steps to annotate your program so that classes you define are persistence-capable.

1 Define your class to implement the **IPersistent** interface. See page 291.

2 In the class definition, define the required fields. See page 291.

3 In the class definition, define the required methods. See page 292.

4 In the class definition, define accessor methods so that they make the appropriate **ObjectStore.fetch()** and **ObjectStore.dirty()** method calls. See page 294.

5 If required, define a class that extends the **ClassInfo** class. See page 295.

Interfaces never require **ClassInfo** classes.

If you will be running your application in an environment that allows the unrestricted use of the Java reflection API, public or abstract classes with hollow object constructors do not require **ClassInfo** classes. However, all classes that define indexable fields on objects stored in peer (**COM.odi.coll**)collections do require **ClassInfo** classes.

6 For any **ClassInfo** subclasses you define, create an instance of the **ClassInfo** subclass. Only one instance of this subclass is ever needed. See page 298.

7 Call the static **get()** method on **ClassInfo**. (Typically, this is in static initializer code for the manually annotated class.) See page 298.

Some Java-supplied classes are persistence-capable. Others are not persistence-capable and cannot be made persistence-capable. A third category of classes can be made persistence-capable, but there are important issues to consider when you do so. Be sure to read Java-Supplied Persistence-Capable Classes on page 360.

About interfaces

Interfaces are always persistence-capable. You must specify them when you run the postprocesor, but other than that you do not need to do anything to make an interface persistence-capable.

Implementing the IPersistent Interface

Every persistence-capable class must implement the **IPersistent** interface or be a subclass of a class that implements it. As with any interface, every method defined in the **IPersistent** interface must be defined in a class that implements **IPersistent**. If you do not define all methods, you can run the postprocessor to insert the missing methods. If you do not define all methods, and you do not run the postprocessor, you receive a compilation error.

Defining the Required Fields

The following code must be in your class definition. You can add this code yourself, or you can run the postprocessor to add it.

```
transient private COM.odi.imp.ObjectReference ODIRef;
transient public byte ODIObjectState;
```

The **ODIRef** field stores a reference. The **ODIObjectState** field holds some object state bits. The underlying run-time classes in ObjectStore access these fields through the **IPersistent** accessor methods as needed.

Defining Required Methods in the Class Definition

This section describes the methods that must be defined in a class that implements the **IPersistent** interface.

initializeContents()

Define the **initializeContents()** method to load real values into hollow instances of your class. This changes a hollow object to an active object. ObjectStore provides methods on the **GenericObject** class that retrieve each **Field** type. Be sure to call the correct methods for the fields in your persistent object. There is a separate method for obtaining each type of **Field** object. ObjectStore calls the **initializeContents()** method as needed. The method signature is

```
public void initializeContents(GenericObject genObj)
```

Here is an example:

```
public void initializeContents(GenericObject handle) {
    name = handle.getStringField(1, PCI);
    age = handle.getIntField(2, PCI);
    children = (Person[])handle.getArrayField(3, PCI);
}
```

If the class you are annotating implements **IPersistent** through a superclass, you must also initialize superclass fields by invoking **initializeContents()** on the superclass.

flushContents()

Define the **flushContents()** method to copy values from a modified instance (active persistent object) back to the database. This method changes an active clean or dirty object to an active clean object. ObjectStore provides methods on the **GenericObject** class that set each **Field** type. Be sure to call the correct methods for the fields in your persistent object. There is a separate method for setting each type of **Field** object. ObjectStore calls the **flushContents()** method as needed. The method signature is

```
public void flushContents(GenericObject genObj)
```

Here is an example:

```
public void flushContents(GenericObject handle) {
    handle.setClassField(1, name, PCI);
    handle.setIntField(2, age, PCI);
    handle.setArrayField(3, children, PCI);
}
```

If the class you are annotating implements **IPersistent** through a superclass, you must also flush superclass fields by invoking **flushContents()** on the superclass.

clearContents()

Define the **clearContents()** method to reset the values of an instance to the default values. This method changes an active clean object to a hollow object. This method must set all reference fields that referred to persistent objects to null. ObjectStore calls this method as needed. The method signature is

```
public void clearContents()
```

Here is an example:

```
public void clearContents() {
    name = null;
    age = 0;
    children = null;
}
```

If the class you are annotating implements **IPersistent** through a superclass, you must also clear superclass fields by invoking **clearContents()** on the superclass.

Hook methods

The following methods must also be in the class definition. You can define them as methods with empty bodies. If you do not define them and your class does not directly implement the **IPersistent** interface, you can use the postprocessor to add these methods with empty bodies.

- **postInitializeContents()** is called by ObjectStore immediately after it calls the **initializeContents()** method.
- **preFlushContents()** is called by ObjectStore immediately before it calls the **flushContents()** method.
- **preClearContents()** is called by ObjectStore immediately before it calls the **clearContents()** method.
- **preDestroyPersistent()** is called by ObjectStore immediately before it calls the **ObjectStore.destroy()** method.

Field accessor methods

The following accessor methods must be in the class definition.

- **public ObjectReference ODIgetRef()**
- **public void ODIsetRef(ObjectReference objRef)**
- **public byte ODIgetState()**
- **public void ODIsetState(byte state)**

If you do not want to define them, you can run the postprocessor to insert them for you, but you must not declare the class to implement **IPersistent.** However, if you explicitly define an **ODIget***xxx***()** method, you must explicitly define its associated **ODIset***xxx***()** method. Likewise, if you explicitly define an **ODIset***xxx***()** method, you must explicitly define its associated **ODIget***xxx***()** method.

If you add the code yourself, it must look like this:

```
public COM.odi.imp.ObjectReference ODIgetRef() {
    return ODIRef;
}

public void ODIsetRef(COM.odi.imp.ObjectReference objRef) {
    ODIRef = objRef;
}

public byte ODIgetState() {
    return ODIObjectState;
}

public void ODIsetState(byte state) {
    ODIObjectState = state;
}
```

Making Object Contents Accessible

In each class that you want to be persistence-capable, you must annotate your class definition to include calls to the **ObjectStore.fetch()** and **ObjectStore.dirty()** methods. It does not matter whether the class explicitly implements **IPersistent** or inherits from a class that implements **IPersistent**. These calls are required for the class to be persistence-capable.

With some exceptions, before your application can access the contents of an object, it must call the

- **ObjectStore.fetch()** method on the object to read its contents
- **ObjectStore.dirty()** method on the object to modify its contents

Calls to **fetch()** or **dirty()**

Your application calls the method and passes an object whose contents you want to access. This makes the contents of the object available. Modify the methods that reference nonstatic fields to call the **ObjectStore.fetch()** and **ObjectStore.dirty()** methods as needed. While this step is not mandatory, it does provide a systematic way to ensure that the application calls the **fetch()** or **dirty()** method before accessing or updating object contents.

Remember that you can add some annotations and run the postprocessor to add other annotations. You might want to define the required methods and the **ClassInfo** subclass, but let the postprocessor insert the required **fetch()** and **dirty()** calls.

Exceptions

You do not need to call the **fetch()** or **dirty()** method on instances of primitive wrapper classes (see Description of Java-Supplied Persistence-Capable Classes on page 360). If you do call **fetch()** or **dirty()** on these objects, nothing happens and processing continues.

You do not need to call the **fetch()** or **dirty()** method on instances of **java.lang.String**. You do not need to call the **fetch()** or **dirty()** method on the objects listed below. If you do call **fetch()** or **dirty()** on these objects, nothing happens and processing continues.

- Instances of primitive wrapper classes
- Java peer objects (remember that ObjectStore collection objects are Java peer objects)

If you call **fetch()** on instances of **java.lang.String**, nothing happens. If you call **dirty()** on instances of **java.lang.String**, ObjectStore throws ObjectException.

Defining a ClassInfo Subclass

If required, define a public class that inherits from the **ClassInfo** class. (See page 290 for requirements.) You must define this class in a separate file. If you plan to use the postprocessor to insert any annotations, the name of this class must be one of the following:

- The name of the persistence-capable class followed by **ClassInfo**, for example, **PersonClassInfo**.
- The suffix specified with the **-classinfosuffix** option to the postprocessor.

In each **ClassInfo** subclass definition, you must include the methods described below.

create()

Define a **create()** method to create instances of your persistence-capable class with default field values:

```
public IPersistent create() { return new Person(this); }
```

This should call a constructor, referred to as a hollow object constructor, that leaves fields in the default state. For an abstract class, the **create()** method can return null.

getClassDescriptor()

Define the public **getClassDescriptor()** method to obtain the class object for your class. For example:

```
public Class getClassDescriptor()
    throws ClassNotFoundException {
    return Class.forName("COM.odi.demo.people.Person"); }
```

getFields()

Define the public **getFields()** method to allow access to the names and types of the fields of the class. For example:

```
public Field[] getFields() { return fields; }
private static Field[] fields = {
    Field.createString("name"),
    Field.createInt("age"),
    Field.createClassArray("children", "Person", 1)
};
```

The definition of the **getFields()** method can specify create methods for fields that are not in the class definition and can omit create methods for fields that are in the class definition.

Example of a Manually Annotated Persistence-Capable Class

Here is an example of a definition of a manually annotated persistence-capable class. Three consecutive periods indicate lines from a complete program that have been left out here because they are not pertinent to creating a persistence-capable class.

Class definition

```
package COM.odi.demo.people;
import COM.odi.*;

// Define a class that implements IPersistent:

class Person implements IPersistent {

   // Fields:

   String name;
   int age;
   Person children[];
   // Other fields ...

   // Constructor:

   public Person(String name, int age, Person children[]) {
    this.name = name; this.age = age; this.children = children;
   }

   // Hollow object constructor:

   public Person(ClassInfo info) { }

   // Accessor methods that have been modified to call
   // the fetch() and dirty() methods:

   public String getName() {ObjectStore.fetch(this); return name; }
   public void setName(String name) {ObjectStore.dirty(this);
       this.name = name; }
   public int getAge() {ObjectStore.fetch(this); return age; }
   public void setAge(int age) {ObjectStore.dirty(this);
       this.age = age; }
   public Person[] getChildren() {ObjectStore.fetch(this);
      return children; }
   public void setChildren(Person children[]) {
      ObjectStore.dirty(this); this.children = children;
   }
   // Other methods ...

   // Additions required for ObjectStore:

   // Define the initializeContents() method to load real
   // values into hollow persistent objects, which makes
   // them active persistent objects:

   public void initializeContents(GenericObject handle) {
    name = handle.getStringField(1, myClassInfo);
    age = handle.getIntField(2, myClassInfo);
```

```
    children = (Person[])handle.getArrayField(3, myClassInfo);
  }

  // Define the flushContents() method to copy the
  // contents of a persistent object to the database:

  public void flushContents(GenericObject handle) {
    handle.setClassField(1, name, myClassInfo);
    handle.setIntField(2, age, myClassInfo);
    handle.setArrayField(3, children, myClassInfo);
  }

  // Define the clearContents() method to reset the values
  // of a persistent instance to the default values.
  // This method must set all reference fields that
  // referred to persistent objects to null:

  public void clearContents() {
    name = null;
    age = 0;
    children = null;
  }

    // Hook methods.
    public void preFlushContents() { }
    public void preClearcontents() { }
    public void postInitializecontents() { }
    public void preDestroyPersistent() { }

    // Define the ODIRef and ODIObjectState fields and
    // their accessor methods.
  transient private COM.odi.imp.ObjectReference ODIRef;
  transient public byte ODIObjectState;
  public COM.odi.imp.ObjectReference ODIgetRef() {
     return ODIRef;
  }
  public void ODIsetRef(COM.odi.imp.ObjectReference objRef) {
     ODIRef = objRef;
  }
  public byte ODIgetState() {
     return ODIObjectState;
  }
  public void ODIsetState(byte state) {
     ODIObjectState = state;
  }

  // Create an instance of the subclass of ClassInfo and
  // register that instance:

  static ClassInfo myClassInfo =
     ClassInfo.get("COM.odi.people.Person");
}
```

ClassInfo definition

In a separate file, define the subclass of the **ClassInfo** class if its definition is required. For example:

```
// Define the subclass of ClassInfo. A recommended naming
// convention is to prefix the name of your persistence-capable
// class to "ClassInfo".

package COM.odi.demo.people;

import COM.odi.*;

public class PersonClassInfo extends ClassInfo {

    // Define a create() method to create instances of your
    // class with default field values. The method
    // calls the hollow object constructor and passes this,
    // which is an instance of the ClassInfo subclass:

    public IPersistent create() { return new Person(this); }

   // Define these public methods to provide access to
   // the name of the persistence-capable class, the name of its
   // superclass, and the names of its fields.
   // The array returned by getFields() must contain the
   // fields in the order of their field numbers.

    public Class getClassDescriptor()
        throws ClassNotFoundException {
    return Class.forName("COM.odi.demo.people.Person"); }
        public Field[] getFields() { return fields; }
        private static Field[] fields = {
        Field.createString("name"),
        Field.createInt("age"),
        Field.createClassArray(
        "children", "COM.odi.demo.People.Person", 1)

   };
}
```

It does not matter whether the **ClassInfo** class explicitly implements **IPersistent** or inherits from a class that implements **IPersistent**.

ClassInfo is an abstract class for managing schema information for persistence-capable classes. ObjectStore requires the schema information to manage the object. If you do not explicitly define a **ClassInfo** class, ObjectStore uses the Java reflection API to creates the needed information at runtime.

After you perform the steps described in this section, you can store instances of your class in a database.

ObjectStore does not let you store **final** instance variables persistently. This is because it is not possible to write the **initializeContents()** and **clearContents()** methods to correctly handle final instance variables.

Additional Information About Manual Annotation

This section provides additional information about manually annotating a class to be persistence-capable. It discusses the following topics:

Defining a hashCode() Method

Every class inherits from the **Object** class, which defines the **hashCode()** method and provides a default implementation. For a persistent object, this default implementation often returns a different value for the same persistent object (the object on the disk) at different times. This is because ObjectStore fetches the persistent object into different Java objects at different times (in different transactions or different invocations of Java).

This is not a problem if you never use the object as a key in a persistent hash table or other structure that uses the **hashCode()** method to locate objects. If you do use the object as a key, the hash table or other structure that relies on the **hashCode()** method might become corrupted when you bring the objects back from the database.

To resolve this problem, you can define your own **hashCode()** method and base it on the contents of the objects so it returns the same thing every time. The signature of this method must be

public int hashCode()

Defining a clone() Method

If your persistence-capable class implements the **Cloneable** interface, your class must define a **clone()** method. This **clone()** method must ensure that it correctly initializes and checks the **ODIRef** and **ODIObjectState** fields when it performs a clone operation. For new cloned objects, your application should initialize **ODIRef** to null and **ODIObjectState** to zero.

Working with Transient-Only and Persistent-Only Fields

The definition of the **ClassInfo.getFields()** method returns an array of **COM.odi.Field** instances. There is one element for each field that you want to store and retrieve in a persistent object. ObjectStore does not require an exact match between each field in the Java class definition and each field array element returned by the **getFields()** method. Furthermore, fields listed in the **getFields()** return value need not directly represent fields in the class. They can represent state from which values for fields in the class are synthesized.

Transient-only fields

A persistence-capable Java class can define a field that does not appear in the list of fields returned by the **ClassInfo.getFields()** method. Such a field is a transient-only field. The **initializeContents()** method that is associated with the class can be used to initialize transient-only fields based on persistent state.

For example:

```
class A {
   transient java.awt.Component myVisualizationComponent;
   int myValue;
      ...
   }
```

In this class, the **myVisualizationComponent** field is declared to be a transient reference to **java.awt.Component**. **java.awt** is a package containing GUI classes that do not lend themselves to being persistence-capable.

Number of fields

The number of nonstatic, nontransient declared fields in the class should generally be equal to the number of fields reported by the **getFields()** method, unless the **flushContents()** and **initializeContents()** methods are written to combine or split fields. If they are so written, you can define an arbitrary mapping of persistent fields to Java instance fields. For example:

```
class Some {
    int a;
    int b;
    int aPlusb;

    initializeContents(GenericObject, go) {
        a=go.getField(1, SomeClassInfo);
        b=go.getField(2, SomeClassInfo);
        c=a+b;
    }
    ...
}
```

In a separate file:

```
public class SomeClassInfo
    static Field[] fields=
{   field.createInt"a");
    field.createInt("b");
}
```

Persistent-only fields

The list of **Field** objects returned by the **getFields()** method might include one or more fields that are not in the Java class definition. Such fields are persistent-only fields. The **flushContents()** method associated with the class must set the field value in the generic object based on other fields of the class.

Variable initializers

If you manually annotate a class, you should avoid using variable intializers to initialize persistent fields of persistence-capable objects. Instead, perform the initialization in the contructor. This is because the values computed by the variable initializer expression are typically overwritten by the **COM.odi.IPersistent.initializeContents()** method. When an object is actually fetched from the database, the fields are initialized with their correct persistent values.

Example

An example of how you might use transient-only and persistent-only fields is in the **demo** directory that is included in ObjectStore. In the **rep** example, **Rectangle.a** and **Rectangle.b** are transient-only fields, while **ax**, **ay**, **bx**, and **by** are persistent-only fields. Here is the part of the example that shows this:

```
package COM.odi.demo.rep;

/**
* A Rectangle has two Points, representing its upper-left
* and lower-right corners. However, its persistent representation
* is formed by storing the x and y coordinates of the two points,
* rather than the points themselves. This demonstrates the control
* that the definer of a persistent class has over the persistent
```

```
* representation. Note that Identity of the Point objects is not
* preserved, since the Point objects are not persistent objects. */

import COM.odi.*;

public class Rectangle implements IPersistent {

    transient private COM.odi.imp.ObjectReference ODIref;
    transient public byte ODIobjectState;

    transient Point a;
    transient Point b;

    static ClassInfo classInfo
        = ClassInfo.register(new RectangleClassInfo());

    public COM.odi.imp.ObjectReference ODIgetRef() {
        return ODIref;
    }

    public void ODIsetRef(COM.odi.imp.ObjectReference objRef) {
        ODIref = objRef;
    }

    public byte ODIgetState() {
        return ODIobjectState;
    }

    public void ODIsetState(byte state) {
        ODIobjectState = state;
    }

    Rectangle(Point a, Point b) {
        this.a = a;
        this.b = b;
    }

    void describe() {
        System.out.println("Rectangle with two points:");
        a.describe();
        b.describe();
    }

    /* Annotations for persistence. */

    Rectangle(ClassInfo ignored) {}

    public void initializeContents(GenericObject handle) {
        a = new Point(handle.getIntField(1, classInfo),
            handle.getIntField(2, classInfo));
        b = new Point(handle.getIntField(3, classInfo),
            handle.getIntField(4, classInfo));
    }

    public void flushContents(GenericObject handle) {
        handle.setIntField(1, a.x, classInfo);
        handle.setIntField(2, a.y, classInfo);
        handle.setIntField(3, b.x, classInfo);
```

```
        handle.setIntField(4, b.y, classInfo);
    }

    public void clearContents() {
        a = null;
        b = null;
    }

    public void postInitializeContents() {};

    public void preFlushContents() {};

    public void preClearContents() {};

    public void preDestroyPersistent() {};

    /* This class is never used as a persistent hash key. */
    public int hashCode() {
        return super.hashCode();
    }
}
```

In a separate file:

```
public class RectangleClassInfo extends ClassInfo
{

    public IPersistent create() { return new Rectangle(this); }
    public Class getClassDescriptor() throws
        ClassNotFoundException {
        return Class.forName("COM.odi.demo.rep.Rectangle");
    }

    public Field[] getFields() { return fields; }
    private static Field[] fields =
    { Field.createInt("ax"),
        Field.createInt("ay"),
        Field.createInt("bx"),
        Field.createInt("by"), };}
```

Defining Persistence-Aware Classes

A persistence-aware class is a class whose instances

- Can operate on persistent objects
- Cannot be stored in a database

For a class to be persistence-aware, you must annotate it so that it includes calls to the **ObjectStore.fetch()** and **ObjectStore.dirty()** methods. The **fetch()** method makes the contents of a persistent object available to be read. The **dirty()** method makes the contents of a persistent object available to be modified.

To make a class persistence-aware, modify each method that references

- Nonstatic fields of persistence-capable classes
- Array elements of arrays that might be persistent

Modify each method so that it calls the **ObjectStore.fetch()** or **ObjectStoret.dirty()** method. This call must be before any attempt to access the contents of the persistent object. The **fetch()** and **dirty()** methods make the contents of persistent objects available.

A persistence-aware class includes the **fetch()** and **dirty()** annotations. It does not include the other annotations that are required for a class to be persistence-capable.

Following Postprocessor Conventions

If you plan to explicitly define all required annotations, you need not be concerned with postprocessor conventions. However, if you plan to explicitly insert some annotations and use the postprocessor to insert other annotations, you must follow these postprocessor conventions.

- The name of the **ClassInfo** subclass must have the following format:

  *class_name***ClassInfo**

  For example, if you define the **Boat** class, the name of the associated subclass of **ClassInfo** must be **BoatClassInfo**.

- In the **ClassInfo** subclass definition, when you define the hollow object constructor, it must take a single argument of type **ClassInfo**. See page 297.

Annotating Abstract Classes

Persistence-capable classes and their superclasses, even if they are abstract, must each have a corresponding **ClassInfo** subclass. But an application does not create instances of abstract classes, so you cannot write the required **create()** method in the **ClassInfo** subclass in the usual way. Define the **create()** method so that it returns null. Since this method will never be called, it is safe to define it this way.

Now, suppose you define the following two classes:

```
abstract class Y {
    int yValue;
    abstract void doSomething();
}

class X extends Y {
    float xValue;
    void doSomething() {}
}
```

Class **Y** must have an associated **ClassInfo** subclass and class **X** must have an associated **ClassInfo** subclass. The **ClassInfo** subclass associated with **X** does not extend the **ClassInfo** subclass associated with **Y**.

In the **ClassInfo** subclass for **X**, the **Field** array must include only those fields defined explicitly in **X**; **XClassInfo.getFields()** must report only the immediate persistent fields in **X**. The **ClassInfo** subclass for **Y** defines a **Field** array that contains the fields explicitly defined in **Y**.

Removing ClassInfo Classes From Existing Applications

If you have applications that you created with earlier ObjectStore releases, you can remove your **ClassInfo** classes for classes that meet all of these conditions:

- The class is defined as **public** or **abstract**.
- The class has a hollow object constructor.
- The class does not define an indexable field on an object that is stored in a peer (**COM.odi.coll)** collection.

Creating and Accessing Fields in Annotations

As part of the process of manually defining a class that is persistence-capable, the required annotations must (among other things)

- Define an **initializeContents()** method in the persistence-capable class.
- Define a **flushContents()** method in the persistence-capable class.
- Define a **getFields()** method in the **ClassInfo** subclass.

To correctly define these methods, you must know how ObjectStore makes persistent objects accessible and what methods are available to create and access individual fields in an object. To help you do this, this section discusses the following topics:

Making Persistent Objects Accessible

The **ObjectStore.fetch()** method makes the contents of a persistent object available to be read by an application. The **ObjectStore.dirty()** method makes the contents of a persistent object available to be updated by an application.

To execute a **fetch()** or **dirty()** call, ObjectStore first checks whether a **fetch()** or **dirty()** call was already invoked on the object in the current transaction. If it was, ObjectStore does nothing and the program continues. If it was not, ObjectStore executes the **fetch()** or **dirty()** call as required.

Call to **initializeContents()**

When ObjectStore retrieves a persistent object, it calls the **initializeContents()** method that you defined. The **initializeContents()** method calls methods on **GenericObject** to obtain the field values for the persistent object. The result is that your program has access to the desired data.

Description of **GenericObject**

ObjectStore provides the **GenericObject** class for transferring data between a database and a Java application or applet. A generic object represents an object's data as it is stored in the database. A generic object is a temporary buffer that ObjectStore uses while it is copying data from the database into a persistent object or writing data into the database from a persistent object. ObjectStore creates instances of **GenericObject** as needed. You do not define subclasses of **GenericObject** nor do you create instances of **GenericObject**.

For an object that was not already retrieved, ObjectStore copies the contents of the object from the database into the **GenericObject** instance. It then passes this instance to the **initializeContents()** method defined in the persistence-capable class.

Call to **flushContents()**

Suppose you called the **dirty()** method on a persistent object and modified it. To update the object in the database, commit the transaction. This causes ObjectStore to create an instance of **GenericObject** to hold the contents of your object. Then ObjectStore calls the **flushContents()** method that you defined when you defined the persistence-capable class.

The **flushContents()** method must call methods on the **GenericObject** instance that store the object's field values in the generic object. ObjectStore calls the **flushContents()** method as needed to copy the new contents of the object into the database.

Creating Fields

ObjectStore provides the **Field** class to represent a Java field in a persistent object. When you define a persistence-capable class, you must define a **getFields()** method in the required **ClassInfo** subclass. This method provides a list of the nonstatic fields (also called instance variables) whose values are being stored and retrieved.

Description of **getFields()**

The **getFields()** method must return an array that contains the nonstatic persistent object fields. The order in which they appear in the array implies their associated field numbers. This array must include only those fields defined in the persistence-capable class and not any inherited fields.

Field numbers represent the position of a nonstatic field within the list of all nonstatic fields defined for the class and its superclasses. The first field has field number 1. (Note that the first field number is *not* 0.)

Order of fields

When you define the **getFields()** method in the **ClassInfo** subclass, you determine the order, and hence the number, of each field even though you do not explicitly assign any numbers. ObjectStore assigns the numbers according to the order in which the values are returned from the field create methods defined in the **getFields()** method. The field numbers are consecutive with no gaps. For example:

Example

```
public Field[] getFields() { return fields; }

    private static Field[] fields = {
        Field.createString("name"),
        Field.createInt("age"),
        Field.createClassArray("children",
            "COM.odi.demo.people.Person", 1)
    };
```

The definition above causes ObjectStore to associate **1** with the **name** field, **2** with the **age** field, and **3** with the **children** field.

When you define the **initializeContents()** and **flushContents()** methods, you must specify the correct field number for each field that the methods get and set.

Creation methods

The **Field** class provides a create method for each Java data type. Minimally, the create methods on the **Field** object

- Return the created **Field** object
- Take a **String** parameter that specifies the name of the field

There are separate create methods for singleton and array fields of each primitive type. There are also string fields, class fields, and interface fields. The complete list of Field create methods is in Methods for Creating Fields and Accessing Them in Generic Objects on page 314.

Getting and Setting Generic Object Field Values

As described earlier, ObjectStore provides the **GenericObject** class to transfer objects between the database and an application. Consequently, when you define a persistence-capable class, you must define the **initializeContents()** method to retrieve values from fields in instances of **GenericObject**, and the **flushContents()** method to set values in fields of instances of **GenericObject**.

When you define the **initializeContents()** and **flushContents()** methods, you must use a method that is appropriate for the type of each field in the instance of **GenericObject**. For example, for each character field, you must use the

- **getCharField()** method in the **initializeContents()** method
- **setCharField()** method in the **flushContents()** method

There is a different method for getting and setting each Java type. To get or set an array of any type, you define the **getArrayField()** and **setArrayField()** methods, respectively. In the **initializeContents()** method, be sure to call the methods that get the values. In the **flushContents()** method, be sure to call the methods that set the values. The methods that get and set fields in a generic object are listed in the table in Methods for Creating Fields and Accessing Them in Generic Objects on page 314.

Methods for Creating Fields and Accessing Them in Generic Objects

Kind of Java Field	*Method That Operates on It*
Single byte (**byte**)	**Field.createByte()** **GenericObject.getByteField()** **GenericObject.setByteField()**
Array of bytes (**byte[]**)	**Field.createByteArray()** **GenericObject.getArrayField()** **GenericObject.setArrayField()**
Single character (**char**)	**Field.createChar()** **GenericObject.getCharField()** **GenericObject.setCharField()**
Array of characters (**char[]**)	**Field.createCharArray()** **GenericObject.getArrayField()** **GenericObject.setArrayField()**
Single 16-bit integer (**short**)	**Field.createShort()** **GenericObject.getShortField()** **GenericObject.setShortField()**
Array of 16-bit integers (**short[]**)	**Field.createShortArray()** **GenericObject.getArrayField()** **GenericObject.setArrayField()**
Single 32-bit integer (**int**)	**Field.createInt()** **GenericObject.getIntField()** **GenericObject.setIntField()**
Array of 32-bit integers (**int[]**)	**Field.createIntArray()** **GenericObject.getArrayField()** **GenericObject.setArrayField()**
Single 64-bit integer (**long**)	**Field.createLong()** **GenericObject.getLongField()** **GenericObject.setLongField()**
Array of 64-bit integers (**long[]**)	**Field.createLongArray()** **GenericObject.getArrayField()** **GenericObject.setArrayField()**
Single 32-bit floating-point number (**float)**	**Field.createFloat()** **GenericObject.getFloatField()** **GenericObject.setFloatField()**
Array of 32-bit floating-point numbers (**float[]**)	**Field.createFloatArray()** **GenericObject.getArrayField()** **GenericObject.setArrayField()**
Single 64-bit floating-point number (**double**)	**Field.createDouble()** **GenericObject.getDoubleField()** **GenericObject.setDoubleField()**

Kind of Java Field	*Method That Operates on It*
Array of 64-bit floating-point numbers (**double[]**)	**Field.createDoubleArray()** **GenericObject.getArrayField()** **GenericObject.setArrayField()**
Single Boolean value (**boolean**)	**Field.createBoolean()** **GenericObject.getBooleanField()** **GenericObject.setBooleanField()**
Array of Boolean values (**boolean[]**)	**Field.createBooleanArray()** **GenericObject.getArrayField()** **GenericObject.setArrayField()**
Single string value (**String**)	**Field.createString()** **GenericObject.getStringField()** **GenericObject.setStringField()**
Array of string values (**String[]**)	**Field.createStringArray()** **GenericObject.getArrayField()** **GenericObject.setArrayField()**
A class	**Field.createClass()** **GenericObject.getClassField()** **GenericObject.setClassField()**
A class array	**Field.createClassArray()** **GenericObject.getArrayField()** **GenericObject.setArrayField()**
An interface	**Field.createInterface()** **GenericObject.getInterfaceField()** **GenericObject.setInterfaceField()**
An interface array	**Field.createInterfaceArray()** **GenericObject.getArrayField()** **GenericObject.setArrayField()**

Chapter 10
Controlling Concurrency

This chapter provides information about ways that you can control concurrency. The APIs described in this chapter make it easier for your application to access data and less likely that your application must wait to access that data. In addition, you can choose to limit access by other users to the same data.

Contents

This chapter discusses the following topics:

Reducing Wait Time for Locks

What can you do to reduce the overhead of waiting for locks? One application can reduce the waiting overhead for other concurrent applications by avoiding locking data unnecessarily, and by avoiding locking data for unnecessarily long periods of time. This section describes several techniques for minimizing wait time.

Clustering

One way to help avoid locking data unnecessarily involves not clustering objects together when they are not normally accessed together. Suppose that, during a given transaction, an application requires **object-a** but not **object-b**. The two objects are stored on the same page. Since ObjectStore performs page-level locking, when you access one of these objects ObjectStore locks both of them. This prevents other processes from accessing either object until the end of the transaction. If you store **object-b** in a different segment from **object-a**, you guarantee that the objects are on different pages. Therefore, the objects will not be locked together.

Transaction Length

Making transactions shorter is one way to avoid locking data for unnecessarily long periods of time. If you do this, you must still ensure that objects in the database are in a consistent state between transactions.

The disadvantage of using shorter transactions is that it can mean using a greater number of transactions. This can increase network overhead, because each transaction commit requires the client to send a *commit message* to the Server. Nevertheless, this extra network overhead is often outweighed by the savings from shorter waits for locks to be released.

It is sometimes particularly important to make transactions that store new objects in the database or destroy persistent objects as short as possible. This is because many write locks are required, which can decrease the level of concurrency.

Multiversion Concurrency Control (MVCC)

Read-only transactions can use *multiversion concurrency control*, or MVCC. With MVCC, an application can perform nonblocking reads of a database. This allows another application to update the database concurrently, with no waiting by either the reader or the writer. See Using Multiversion Concurrency Control (MVCC) on page 320.

Lock Timeouts

Lock timeouts provide the ability to limit the time that ObjectStore waits to obtain a lock. When you try to obtain a lock on an object, segment, or database, you can specify the number of milliseconds for which it is all right to wait for the lock. See Locking Objects, Segments, and Databases to Ensure Access on page 328.

Conflicts Caused by Schema Installation

If you find that there are concurrency conflicts caused by incremental schema installation, you can install schema information in batch mode. See Installing Schema Information in Batch Mode on page 333.

Using Multiversion Concurrency Control (MVCC)

When an application uses multiversion concurrency control (MVCC), it can perform nonblocking reads of a database. This means that another ObjectStore application can concurrently update the database. Neither the reader nor the writer has to wait for the other. To use MVCC, specify **ObjectStore.OPEN_MVCC** as the open type when you open a database.

When Is MVCC Appropriate?

MVCC is useful when your application contains a transaction that

- Does not modify a database
- Does not require a view of the database that is completely up to date, but can instead rely on a snapshot of the data
- Does not depend on data in a database opened for MVCC being transaction consistent with data in other databases

How Does MVCC Work?

In each transaction in which an application accesses a database opened for MVCC, it is as if the application were viewing a snapshot of the database. This snapshot

- Is taken *sometime* during the transaction
- Is internally consistent
- Might not contain changes that were committed by other sessions during the transaction
- Contains all changes that were committed before the transaction started

Obtaining Read Locks

When an application has a database opened for MVCC, the application never has to wait for read locks on the database. When an application reads data from a database opened for MVCC, the application never causes other applications to wait for write locks. In addition, when an application accesses a database it has opened for MVCC, the application never causes a deadlock.

Accessing Multiple Databases in a Transaction

When an application reads a database opened for MVCC, the snapshot it views is internally consistent but potentially out of date. This means that the snapshot might not be consistent with other databases accessed in the same transaction. Even two databases, both of which are opened for MVCC, might not be consistent with each other. Updates might be performed on one of the databases in between the times of their snapshots.

Serializability

A snapshot might be out of date by the time an application reads some data. However, if each transaction that accesses a database opened for MVCC accesses only that one database, MVCC retains serializability. Such a transaction views a database state that would have resulted from some serial execution of all transactions. All transactions produce the same effects as would have been produced by the serial execution.

Opening a Database for MVCC Access

To use MVCC, specify **OPEN_MVCC** as the open mode when you open a database. For example:

```
Database db = Database.open(
   "myDb.odb", ObjectStore.OPEN_MVCC);
```

After an application opens a database for MVCC, it can read that database without ever waiting for locks or blocking other applications.

You cannot change the open mode of an open database. To change the type of access to a database, you must close the database and then reopen it with a specification of the new open mode.

If you try to update data in a database that you opened for MVCC, ObjectStore throws UpdateReadOnlyException.

In a session, all cooperating threads use the same access mode for a particular database. For example, a thread cannot open a database for update if a cooperating thread has already opened that database for MVCC. However, if a thread opens a database for MVCC, a thread in another session, that is, a noncooperating thread, can open the same database for update.

In the same transaction, your application can open one or more databases for MVCC and open other databases for read-only or update.

Determining If a Database Is Opened for MVCC

To determine if a database is opened for MVCC, call the **Database.getOpenMode()** method. The method signature is

public int Database.getOpenMode()

If the database is opened for MVCC, ObjectStore returns the **ObjectStore.OPEN_MVCC** constant. Otherwise, if the database is open, ObjectStore returns either the **ObjectStore.OPEN_UPDATE** or **ObjectStore.OPEN_READONLY** constant.

When ObjectStore opens a database as a result of following a cross-database pointer, the automatic open mode can be **ObjectStore.OPEN_MVCC**. If it is and there are multiple databases open, it is possible that the databases are not consistent with each other. Each database is always internally consistent.

Updating the Snapshot

In a transaction in which you access a database that you opened for MVCC, you might want to update the snapshot periodically. There are two ways to do this:

- End the transaction with **commit()** or **abort()** and start a new transaction.
- Call **checkpoint()** on the transaction. This has the effect of committing the transaction and starting a new transaction, but it does not incur the overhead of a new transaction. See Checkpoint: Committing and Continuing a Transaction on page 325.

To help you decide when to do this, you can find out if, during your transaction, there were any write locks on the database you opened for MVCC. A write lock indicates that another application *might* have made a modification to the database. To do this, call the **Transaction.hasLockContention()** method on your in-progress transaction. The method signature is

public boolean hasLockContention()

ObjectStore returns true if a Server involved in your transaction has write-locked an object that was read by your application. PSE and PSE Pro always return false.

With a return value of true, if you commit your transaction and start a new one, or checkpoint your transaction, you might have access to updated data. But you might also have access to the same data if the application that had the write lock either aborted the transaction or did not commit any changes.

Where to Find Additional Information

Additional information about MVCC can be found in the ObjectStore C++ documentation. See *ObjectStore Advanced C++ API User Guide*, Chapter 2, Advanced Transactions, for information about

- MVCC and the transaction log
- Conflict detection
- Propagating data from the log to the database
- Transaction locking examples

Checkpoint: Committing and Continuing a Transaction

With the **Transaction.checkpoint()** method, you get the effect of committing a transaction and then continuing work in a new transaction in which you have read locks on all or most of the persistent objects that were locked in the committed transaction. This is useful when

- You are making modifications to a database. You want to periodically commit your changes but continue updating the database without intervention. For example, you might be loading new data into the database.
- You want to make your changes available to MVCC readers.
- You opened a database for MVCC and you want an updated snapshot.

Note

This checkpoint differs from a conventional checkpoint. In this checkpoint, an application might not have all the locks after the checkpoint that it had before the checkpoint. The details are explained in the next section.

Caution

If your application checkpoints a transaction while an annotated method is executing, your program might incorrectly access persistent objects after the checkpoint. For more information about this and a workaround, see Troubleshooting Access to Persistent Objects on page 183.

Advantages of a Checkpoint

The advantage of a checkpoint is that there is less overhead than when you actually end one transaction and start another. When you checkpoint a transaction, it is as if you committed the transaction and then immediately started a new transaction. But in the new transaction, you already have read locks on most or all of your persistent objects.

If another session is waiting for a write lock on a persistent object that was locked in your transaction, you lose that lock when you checkpoint the transaction. As long as another session is not waiting for a write lock on an object that was associated with your transaction, you reacquire as read locks any locks you had before the checkpoint.

After the checkpoint, the persistent objects are stale, or hollow according to which you specify when you call **checkpoint()**. If you specify that objects should be hollow, you do not have to start from a root object to set up your access to objects. Your application's access to objects is the same before and after the checkpoint.

After a checkpoint, ObjectStore has read locks on the same objects as before the checkpoint, unless another session was waiting for a write lock on one of these objects. In that case, your transaction loses the lock.

If there were any write locks before the checkpoint, ObjectStore changes them to read locks, or gives them to any sessions waiting for those write locks. Consequently, you might have to wait for locks or you might get a deadlock when you try to update the database again.

Suppose your application calls **Transaction.checkpoint()** and then the transaction started by the **checkpoint()** method is aborted. ObjectStore does not commit any changes to the database that were made after the checkpoint operation. Any changes made before the checkpoint remain committed.

Calling the checkpoint() Method

To checkpoint a transaction, call the **Transaction.checkpoint()** method. The method signature is

public void checkpoint(int retain)

The value of **retain** can be one of the following:

- **ObjectStore.RETAIN_STALE** resets the contents of persistent objects to default values and makes all persistent objects stale.
- **ObjectStore.RETAIN_HOLLOW** resets the contents of persistent objects to default values but makes the persistent objects hollow. Before and after the checkpoint, you can use references to the same objects.

When to checkpoint

Before you checkpoint a transaction, you must ensure that the database is in a consistent state because the state is made persistent.

Caution

During the checkpoint, you must ensure that no other thread tries to access the database.

Locking Objects, Segments, and Databases to Ensure Access

There are times when you want to ensure your own access to objects and also limit access to those objects by other sessions. This can be when you want to ensure that

- An operation completes without interruption.
- All objects are immediately available.
- There are no deadlocks.

To ensure your own access, you can lock an object, a segment, or a database. This means that

- A transaction from another session cannot block your transaction from updating the locked object.
- If you update the object, no other sessions can read the object, unless they use MVCC. The use of MVCC by other sessions prevents them from being blocked and from running into a deadlock.

Advantages of locking

The overhead for locking objects is far less than the overhead for reading a database with MVCC. Also, if you want to, you can update a locked object. However, if you lock many individual objects, the overhead might be comparable.

Disadvantages of MVCC

The use of MVCC has some disadvantages:

- You cannot update the database.
- The overhead for MVCC grows with each concurrent update.
- It is possible for multiple databases to be inconsistent with each other.

Note

The **COM.odi.useDatabaseLocking** property is a PSE/PSE Pro feature. If you are using PSE/PSE Pro as well as OSJI, beware of confusing this property with the OSJI **Database.acquireLock()** method. This method allows a session to explicitly lock a particular database for exclusive use.

PSE/PSE Pro

The **acquireLock()** methods do nothing in PSE and PSE Pro.

Description of Acquire Lock Methods

The methods that allow you to acquire locks on objects are described below.

- **ObjectStore.acquireLock()** obtains a lock on a specified object. The method signature is

  ```
  public static void ObjectStore.acquireLock(
      Object object, int lockType, int timeoutMillis);
  ```

- **Segment.acquireLock()** obtains a lock on a specified segment. This locks all the objects in the segment. The method signature is

  ```
  public static void Segment.acquireLock(
      int lockType, int timeoutMillis);
  ```

- **Database.acquireLock()** obtains a lock on a database. This locks all the segments, which locks all the objects in the segments. The method signature is

  ```
  public static void Database.acquireLock(
      int lockType, int timeoutMillis);
  ```

Locking Objects for Read or Write Access

The **lockType** parameter indicates whether you want to read or update the locked objects. You must specify one of the following:

- **ObjectStore.READONLY** instructs ObjectStore to make the contents of the locked objects available to be read. While you have this read lock, other sessions can obtain read locks as usual.
- **ObjectStore.UPDATE** instructs ObjectStore to make the contents of the locked objects available to be modified. You must be in an update transaction and the database must be opened for update. While you have this write lock, no other sessions can access the object, unless they use MVCC to do so.

In a transaction, you can reissue the **acquireLock()** call to change the **lockType**.

Specifying the Wait Time for a Lock

If the lock is not available, the **timeoutMillis** parameter indicates how many milliseconds you are willing to wait for the lock. You can specify

- **ObjectStore.WAIT_FOREVER** to wait until the lock is available.
- **0** if you do not want to wait at all.
- A positive number to indicate the number of milliseconds it is all right to wait. ObjectStore rounds up to the nearest number of seconds.

If ObjectStore cannot acquire the lock, either because you do not want to wait, or because the waiting period has been exceeded, ObjectStore throws LockTimeoutException.

Releasing Locks

To release locks, you must end the transaction in which you acquired them. **Transaction.checkpoint()** also releases locks, however you reacquire locks as read locks if no other session is waiting for a write lock for the objects you had locked.

Locking Peer Objects

When you lock a Java peer object that identifies a persistent C++ object, ObjectStore locks the entire object. However, ObjectStore does not lock subobjects that are logically part of the peer object. For example, when you lock a **COM.odi.coll** collection, you do not lock the contents of the collection.

You cannot lock Java peer objects that represent transient C++ objects.

Obtaining Information About Concurrency Conflicts

Instances of the **LockTimeoutBlocker** class represent clients that have been involved in concurrency conflicts.

Client list

To obtain a list of such clients, call the **LockTimeoutException.getBlockers()** method. The signature is

```
public LockTimeoutBlocker[] getBlockers()
```

Lock type

To find out whether a concurrency conflict was for a read lock or a write lock, call **LockTimeoutBlocker.getLockType()**. This returns **ObjectStore.READONLY** or **ObjectStore.UPDATE**. The method signature is

```
public int getLockType()
```

Client names

To obtain the name of the client that caused the conflict, call **LockTimeoutBlocker.getApplicationName()**. The method signature is

```
public String getApplicationName();
```

Process IDs

To obtain the process ID of the process running a client that caused a conflict, call the **LockTimeoutBlocker.getPID()** method. The method signature is

```
public int getPID()
```

Host names

To obtain the name of the host for the process that was running the client that caused the conflict, call the **LockTimeoutBlocker.getHostName()** method. The method signature is

```
public String getHostName()
```

Setting the Client Name

To allow other applications to see meaningful client names when ObjectStore throws LockTimeoutException, your application should set a client name for itself. Use the system property **COM.odi.applicationName** to do this. Specify the name of the application for the current client. Include it in the list of properties that you specify when you create a session.

Helping Determine the Transaction Victim in a Deadlock

When there is a deadlock, the Server must choose a transaction to abort. The Server uses several criteria to pick the victim. One of the criteria is the transaction priority, which you can set with the **Transaction.setPriority()** method.

To obtain the priority that is assigned to transactions started by the current session, call the **Transaction.getPriority()** method.

Every client has a transaction priority, which is a value in the range **0** to **0xffff**. The default value is **0x8000**, which is in the middle of the range. The value **0** has a special meaning.

In a deadlock situation, the ObjectStore Server compares the transaction priorities of clients involved in the deadlock. If the lowest transaction priority is held by only one client, this client is the victim. If the lowest transaction priority is held by more than one client, the Server chooses a victim according to the setting of the **Deadlock Victim** Server parameter. For information about this parameter, see the ObjectStore C++ Interface documentation, *ObjectStore Management*, Chapter 2, Server Parameters.

If all transactions in a deadlock have a transaction priority of **0**, the Server aborts all of them. While this is not a useful way to run a program, it is useful for debugging. You can run several clients under debuggers, and have them all set their priorities to **0**. When a deadlock happens, all of them abort and you can see what each one of them was doing. You should only use a priority of **0** if you want this special debugging behavior.

Installing Schema Information in Batch Mode

Sometimes, schema installation causes concurrency conflicts. If it does, you can minimize this by installing schema in batch mode.

By default, ObjectStore stores schema information in databases incrementally as needed. If you want to, you can instruct ObjectStore to install schema information in batch mode before it is actually needed. This section provides information about how to do that. The topics discussed are

Background About Schema Information

Schema information describes the classes of objects that are stored in the database. ObjectStore stores schema information in each database.

Kinds of schema information

When you install ObjectStore, you run the **setup** program. One of the things this program does is ensure that schema information for the ObjectStore classes is available to your applications. If your application accesses C++ classes, the process of building your Java/C++ application makes additional schema information available. ObjectStore also needs schema information for classes you define and for any third-party libraries that your classes use. The postprocessor creates this schema information.

Incremental and batch installation

ObjectStore dynamically stores schema information in each database as needed. This is referred to as incremental schema installation. If you want to, you can instruct ObjectStore to install internal schema information when it creates the database and application schema information when you invoke a method to do so. In this way, you install schema information before it is needed. This is referred to as batch schema installation.

Advantages and disadvantages

The advantages of batch schema installation are

- You minimize the chance of a concurrency conflict caused by schema installation.
- It is more efficient when a large percentage of the schema information is actually needed.

The disadvantages are

- It takes a little longer to create the database.
- An initial database is larger than an initial database that uses incremental schema installation.
- Schema information that your application never uses might be installed in a database.

Procedure for Installing Schema in Batch

To perform batch schema installation for a database, follow these steps:

1 Use the postprocessor to create one or more objects that identify the application types.

2 Create a database with the overloading of **Database.create()** that allows you to specify batch schema installation.

3 Start an update transaction.

4 Install the application types in the database. These are the types identified in the object or objects created in step 1.

In the unusual situation where you do not run the postprocessor, you follow a different procedure to perform batch schema installation. See If You Do Not Run the Postprocessor on page 340.

Identifying the Application Types

The application types are the persistence-capable classes you define and any persistence-capable classes in third-party libraries that your classes refer to. To install schema information in batch mode, you need to create one or more objects that identify all application types. There are two postprocessor options that you can specify to do this.

Individual types

When you run the postprocessor to make classes you define persistence-capable, specify the **-summary** option. The format is

-summary *gen_class_name*

This option instructs the postprocessor to generate a class with the name *gen_class_name*. This generated class extends the **PersistentTypeSummary** class. The *gen_class_name*.**getPersistentClasses()** method returns a list of the classes that were made persistence-capable in this execution of the postprocessor.

The generated class has a no-arguments constructor that passes arrays to the **PersistentTypeSummary** class constructor.

Types in libraries

Classes in libraries that have already been postprocessed must have an associated generated class (an associated summary) that identifies the persistence-capable classes in the library. You must do one of the following:

- Create this summary yourself when you specify the **-summary** option during postprocessing of the library.
- Receive this summary with its associated library from a third-party vendor.
- Manually code the summary yourself by defining a class that extends the **PersistentTypeSummary** class.

You can create one object that contains all relevant summaries, or you can have two or more summaries, which collectively identify all application types. Object Design recommends that you include the summaries for libraries in the summary the postprocessor creates for the classes it makes persistence-capable. To do this, run the postprocessor and specify the **-includesummary** option for each library summary. The format for specifying this option is

-includesummary *inc_class_name*

Replace *inc_class_name* with the name of a class generated by a previous execution of the postprocessor with the **-summary** option. You must know the name of the generated class. If you did not postprocess the library yourself, you must get the name of the generated class from the library vendor.

When you specify the **-includesummary** option, you must also specify the **-summary** option. The postprocessor includes the specified summaries in the new summary it creates. It does not matter whether or not the postprocessor is also annotating any classes.

Library providers

If you are providing a library that contains persistence-capable classes, you must also provide the summary object that identifies those classes. ObjectStore uses the summary object to install schema information for your library at run time. This allows users to install a new version of a library without having to rebuild the application type summary.

General use

For a given execution of the postprocessor,

- You can specify the **-summary** option either once or not at all.
- You can specify the **-includesummary** option zero, one, or more times.
- If you specify the **-includesummary** option, you must also specify the **-summary** option.

Creating a Database with Batch Schema Installation

When you create a database, you determine whether you can perform batch schema installation. To allow batch installation, use this overloading of **Database.create()**:

```
public static Database create(
    String name, int fileMode, int schemaInstallMode);
```

The **name** parameter specifies the path name of a file. The path can specify a relative name, fully qualified name, remotely mounted directory, or a host name and host-relative pathname. An ObjectStore Server must be available for the directory that contains the specified file.

The **fileMode** parameter specifies the access mode for the database. ObjectStore throws AccessViolationException if the access mode does not provide owner write access. Otherwise, ObjectStore ignores the access mode specification when you create a database. This is due to a limitation in the Java implementation.

The **schemaInstallMode** parameter specifies the schema installation mode for the database. The value of this parameter must be either of the following:

- **ObjectStore.INSTALL_SCHEMA_BATCH** instructs ObjectStore to immediately store schema information for all types other than your application types in the database being created.
- **ObjectStore.INSTALL_SCHEMA_INCREMENTAL** indicates that ObjectStore should install schema incrementally. ObjectStore installs some schema information immediately. When you store an object in the database, if the required schema information is not already in the database, ObjectStore automatically stores it at that time.

Incremental schema installation is the default. When you use the overloading of **Database.create()** that does not include a specification for **schemaInstallMode**, it is as if you specified **ObjectStore.INSTALL_SCHEMA_INCREMENTAL**.

After you create a database, you cannot change the schema installation mode.

C++ interoperability

If you use Java / C++ interoperability and you specify batch schema installation when you create a database, you must not use C++ to change the schema installation flag. Doing so might cause concurrency conflicts during future access to the database.

Installing Application Types in the Database Schema

If you have one or more **PersistentTypeSummary** objects that identify your application types, you can install schema information in batch in that database. To do so, ensure that the database is opened for update, start an update transaction, and then invoke the **Database.installTypes()** method. The signature is

```
public void installTypes(PersistentTypeSummary summary);
```

Summary parameter

The **summary** parameter must be an object that identifies the persistence-capable types used by the application. If there are multiple objects that collectively identify all application types, you must do either of the following:

- Invoke **installTypes()** for each summary object.
- Construct a summary object that identifies these individual summary objects as included libraries. Use this new summary object as the parameter to the **installTypes()** method.

Typically, you create the summary object with the **-summary** and **-includesummary** options to the postprocessor. In unusual circumstances, you might explicitly create a summary object with the **COM.odi.PersistentTypeSummary** constructor. Information about doing this is in the next section.

Example

For example, suppose you ran the postprocessor with this command:

```
osjcfp -dest .\osjcfpout Class1 Class2 -summary
COM.xyz.MySummary
```

In your program, create a database with batch schema installation and invoke **installTypes()** with an instance of **MySummary**:

```
Database db = Database.create(
    "mydb.odb",
    ObjectStore.OWNER_WRITE,
    ObjectStore.INSTALL_SCHEMA_BATCH);
    Transaction.begin(ObjectStore.UPDATE);
db.installTypes(new COM.xyz.MySummary());
Transaction.current().commit();
```

ClassInfo must be registered

The **installTypes()** method ensures that an instance of the **ClassInfo** subclass associated with each identified persistence-capable class is properly registered. This is normally done automatically by the postprocessor. If ObjectStore cannot find the registered **ClassInfo** subclass instance for a class, ObjectStore throws ClassNotRegisteredException. When ObjectStore throws this exception, it does not install any schema information for any classes. So, even if it can find the schema information for 99 out of 100 classes, it does not install any schema information at all if it cannot find schema information for one class.

If ObjectStore detects a problem in the type summary, it throws InvalidSummaryException.

Indirectly identified classes

ObjectStore installs schema information for persistence-capable classes that are directly and indirectly identified. That is, if a class contains a reference to a class that is not identified in the summary, ObjectStore tries to install the schema information for the referenced class. If the referenced class refers to some other class that is not in the summary, ObjectStore tries to install the schema information for that class.

ObjectStore installs schema information for any superclasses of specified classes.

Omitting batch installation

Suppose you specify batch schema installation when you create a database, but you forget to install the application types before you store objects in the database. ObjectStore behaves as though the incremental schema installation flag were set. When you store the objects, it loads any needed schema information.

You can invoke **installTypes()** at any time and ObjectStore installs schema information for application types if that information is not already in the database.

Omitting a type from the summary

Now suppose you forget to identify a persistence-capable class in the summary that you pass to **installTypes()**. When you store an object of this class in the database, ObjectStore dynamically stores the schema information for that class.

If You Do Not Run the Postprocessor

In the unusual circumstance that you manually annotate your code instead of running the postprocessor, there is a manual way to create a summary of your application's persistence-capable classes.

The **COM.odi.PersistentTypeSummary** class allows your application types to be identified. To create your own summary, invoke the **PersistentTypeSummary** constructor. The signature is

```
public PersistentTypeSummary(
    String[] persistentClasses,
    String[] includedLibrarySummaries);
```

The **persistentClasses** parameter must be an array of names of classes that are persistence-capable. This parameter can be null. The **PersistentTypeSummary.getPersistentClasses()** method returns an array of the class names that have been identified as persistence-capable.

The **includedLibrarySummaries** parameter must be an array of names of classes that extend the **PersistentTypeSummary** class. Typically, these classes identify the persistence-capable classes in a library. This parameter can be null. The **PersistentTypeSummary.getIncludedLibrarySummaries()** method returns an array of the class names that contain summaries of persistence-capable classes in libraries.

When you define a class that extends the **PersistentTypeSummary** class, it must have a no-argument constructor.

Suppose you identify a persistence-capable class in a summary, but that class is not in fact persistence-capable. ObjectStore throws ClassNotRegisteredException at run time if it recognizes this when you invoke the **installTypes()** method on the database.

Chapter 11
Using the Notification Facility

The notification facility allows an application to notify one or more sessions that an event has taken place. Each notification is associated with a location in a database. Your application determines what constitutes an event. In general, an event is anything you want your application to notify other sessions about. For example, a modification to a particular object in a database can be an event.

Contents

This chapter discusses the following topics:

Background About How Notification Works

This section provides information about how the notification system works. It covers these topics:

What Is a Notification?

A notification is an ordinary transient Java object. A notification always specifies

- A location. The location can be a persistent object, a segment, or a database.
- An integer value. This is the value of the **kind** argument in the **Notification** constructor.
- Information about the event. This is either a message string or an array of bytes. In the **Notification** constructor that takes only two arguments, ObjectStore uses a value of null for the third argument, which specifies the message string or byte array.

When a session subscribes to receive notifications for a segment or database, the subscribing session receives any notifications for objects in the segment or database.

In Release 1.3, only one session is allowed in a single Java VM, so only one session can subscribe to a notification. In a future release, it is expected that there can be multiple sessions in the same Java VM. In this release, separate processes can simultaneously subscribe to receive notifications for the same locations.

What Is the Flow of a Notification?

An application creates a notification. Sessions that want information about events related to a particular location subscribe to notifications that specify the location of interest.

When there is an event that involves the location in a notification, the application uses the notification API to send the notification to the ObjectStore Server.

When the Server receives a notification, it determines which sessions are subscribed for that notification. The Server then queues messages to be sent to the receiving sessions. The Server returns the number of messages queued and then asynchronously sends notifications to the Cache Manager of the receiving sessions (the subscribers).

There is a queue inside the Cache Manager for each session on that host. When the Cache Manager receives a notification from the Server, it puts the notification into the queue of each of the subscribing sessions.

A session that subscribes to one or more notifications usually dedicates a thread to receive notifications. When this thread finds a notification in the Cache Manager's queue for that session, it removes the notification from the queue and returns it to the associated session, which performs an application-specific action.

Threads and Notifications

A session starts a thread whose sole purpose is to receive notifications. This thread calls **Notification.receive()** with an argument that specifies how long to wait for notifications. Each session has its own dedicated thread.

When a session receives a notification, it performs an application-specific action. For example, it might post a Windows message, modify the application's transient data structures, or otherwise queue the notification for processing by another thread. It then waits for the next notification.

The thread dedicated to receiving notifications typically does very little work. It might do queue management, for example, maintaining a priority queue of notifications for another thread, or coalescing similar notifications. However, processing should be minimal, so the Cache Manager notification queue does not overflow. Queue overflow can happen if many notifications arrive in quick succession. The thread receiving the notifications might not be able to keep up with the process of removing the notification from the queue and returning it to the session.

In contrast to most other ObjectStore APIs, **Notification.receive()** is not locked out when other threads are in ObjectStore operations. If the thread does not access persistent data or call other ObjectStore APIs, it can run entirely asynchronously.

Transactions and Notifications

An application can send a notification

- Immediately
- When the transaction commits

Transactions are independent of immediate notifications, subscriptions, unsubscriptions, and notification retrieval.

The sending of commit-time notifications, however, is closely integrated with transactions. ObjectStore queues commit-time notifications inside a transaction, and sends them when the transaction commits. If the transaction aborts, the application never sends the commit-time notifications. This is useful when you want dispatch of the notification to be contingent on a database modification that becomes visible when a transaction commits.

There are no restrictions on transaction types. The enclosing transaction can be read-only or update. Databases can be opened read-only, update, or MVCC.

As always, database changes made by a session are not visible to other sessions until the transaction commits. Therefore, all notifications that indicate changes to persistent data should be made at commit time.

Security

To send or subscribe to notifications, a session must open the database that contains the referenced location. If a session does not open a database, it cannot send or receive notifications associated with that database. If you do not have permission to open a database, you cannot send or subscribe to notifications on objects in that database.

Within a database, notifications are not integrated with ObjectStore security. A session can subscribe to notifications and send notifications that reference database locations in any segment.

Creating Notifications

When an application creates a notification, the database that contains the referenced object must be open. It does not matter whether or not a transaction is in progress. However, an object that an application passes to a **Notification** method cannot be a stale object.

Descriptions of Constructors

The constructors for creating notifications have these signatures:

- **public Notification(Object location, int kind)**
- **public Notification(Object location, int kind, byte[] data)**
- **public Notification(Object location, int kind, String message)**

The **location** parameter specifies a persistent object. It indicates the location at the beginning of the object. You must specify a persistent object. It is not good enough for the **location** object to be persistence-capable. If the specified object is not persistent when you try to construct a notification, ObjectStore throws ObjectNotPersistentException. To avoid this, call the **ObjectStore.migrate()** method to store the object in the database before you create the notification.

You can specify a Java peer object if it identifies a persistent C++ object. See *Developing ObjectStore Java Applications That Access C++*, Chapter 3, Writing the Application.

The **kind**, **data**, and **message** parameters provide information about the event. Every notification has a **kind** parameter. If you specify a negative argument, ObjectStore throws IllegalArgumentException.

If you do not want to attach a message or data to the notification, specify null for the third argument when you create the notification, or do not specify a third argument. For example, the following two code fragments do exactly the same thing:

```
String a = null;
new Notification(foobar, 102, a);
```

```
new Notification(foobar, 102);
```

If you do specify a third argument, it is a sequence of bytes. For convenience, ObjectStore allows you to pass in a Java **String** instead of a sequence of bytes. ObjectStore uses UTF-8 encoding to encode **message** arguments into a sequence of bytes.

When ObjectStore sends a notification, it makes the **kind** parameter and the **data** or **message** parameter, if there is one, available to subscribers. If a notification includes a null string (**"null"**), it is received as an empty string (**""**).

Retaining References to Persistent Objects

A notification always contains a reference to a persistent object. You can create a notification in one transaction, and then use it in a subsequent transaction or between transactions.

To do so, you must ensure that the reference is not to a stale object. If it is, ObjectStore throws ObjectException.

To ensure a nonstale object, you can evict the referenced object or commit or abort the transaction with a **retain** type other than **ObjectStore.RETAIN_STALE**. If you evict the object so that you can still use it, but then you abort or commit the transaction with **ObjectStore.RETAIN_STALE**, this cancels the **retain** type specified for **evict()**.

Maximum Data Lengths

For the **byte[] data** parameter in the constructor, the **MAXIMUM_DATA_LENGTH** variable specifies the maximum length of the byte array:

public static final int MAXIMUM_DATA_LENGTH = 16383

For the **String message** parameter, this variable specifies the maximum length of the UTF-8 encoding of the string. If the **String** is ASCII, the compression is one-to-one.

Restriction on data Argument Content

The C++ interface to ObjectStore treats the **data** argument as a C or C++ string, so there cannot be embedded zeros. If your application is directly providing a data array, it must ensure that there are no zeros in the data array. If your application is providing the data in the form of a string, it works correctly because the UTF-8 encoding of strings never uses zero bytes. If you try to construct a notification with a **byte[] data** array and you include a zero, ObjectStore throws IllegalArgumentException.

Subscribing to Receive Notifications

A session uses the **Notification.subscribe()** method to register to receive notifications for the specified location or within the specified segment or database. The database that contains the referenced object or segment must be open when a session calls the **subscribe()** method. These are the overloadings of **subscribe()**:

- **public static void subscribe(Placement placement)**
- **public static void subscribe(Object location)**

The **placement** parameter can specify a database or segment. The subscribing session receives a notification if the application sends a notification that refers to an object in the specified segment or database.

The **location** parameter specifies a persistent object.

A session can subscribe to many locations, segments, and databases simultaneously. ObjectStore stores subscriptions in the ObjectStore Server for as long as the corresponding database is open for the associated session.

Discarding Subscriptions

When a session closes a database, ObjectStore discards any subscriptions in that session to notifications related to that database.

Unsubscribing from Notifications

A session can unsubscribe from particular notifications just as it can subscribe to them. The unsubscription is immediate. The **Notification.unsubscribe()** method has overloadings that are parallel to **subscribe():**

- **public static void unsubscribe(Placement placement)**
- **public static void unsubscribe(Object location)**

If you try to unsubscribe from a notification that you are not subscribed to, nothing happens.

Asynchronous processing

ObjectStore processes notifications asynchronously. Consequently, a session might unsubscribe from a particular notification but still receive notifications for that location because they were already queued.

The only way to cancel a segment or database subscription is to unsubscribe from that segment or database. You cannot unsubscribe from a segment or database by unsubscribing from the notifications about the objects in that segment or database.

Sending Notifications

An application can use any of the following methods to send notifications:

- **public void notifyImmediate()**
- **public void notifyOnCommit()**
- **public static void notifyImmediate(Notification[] notifications)**
- **public static void notifyOnCommit(Notification[] notifications)**

You can send one notification at a time or an array of notifications. You can send notifications immediately or when a transaction commits. If you wait to send a notification until a transaction commits, you ensure that a subscriber does not receive the notification until any associated change is visible in the database.

The object referenced in the notification cannot be stale and cannot have been destroyed.

Modification of an object does not cause a notification to be sent. A notification is sent only when the application program explicitly uses the notification API to send one.

Retrieving Notifications

The ObjectStore Server queues notifications in the Cache Manager that is associated with the subscribing session. It is up to the session to retrieve notifications from the Cache Manager. Typically, a session dedicates a thread to retrieve notifications from the queue. To do this, a thread calls **Notification.receive()**. The method signature is

public static Notification receive(int timeout)

The **timeout** parameter specifies the number of milliseconds to wait for a notification. Specify **ObjectStore.WAIT_FOREVER** to instruct the thread to wait forever. A value of **0** instructs the thread to return if there are no notifications in the queue.

The method returns a **Notification** object, or null, if there are no notifications in the allotted time. When the **timeout** parameter is **ObjectStore.WAIT_FOREVER**, ObjectStore never returns null.

When you call **receive()**, it does not matter whether or not a transaction is in progress. However, the thread from which you call **receive()** must be associated with a session.

A thread that calls the **Notification.receive()** method allows you to avoid polling your application. This method returns as soon as a notification is available, and until then it just waits. No polling is involved. The application is awakened when a notification arrives.

Reading Notifications

To extract the contents of a notification, use the following **Notification** class methods:

- **public Object getLocation()** obtains the persistent object.
- **public int getKind()** obtains the value for the **kind** parameter.
- **public byte[] getData()** returns the byte array associated with the notification. If the application specified a string when it created the notification, this method still returns the byte array. The notification itself does not maintain information about whether it was created with the specification of a string or a byte array.
- **public String getMessage()** decodes the byte array and returns the string associated with the notification. It does not matter whether you specified a string or a byte array when you created the notification. ObjectStore throws java.io.UTFDataFormatException if the string is not correctly encoded in UTF-8 format.

Managing the Notification Process

Managing the notification process involves consideration of the following:

Notification Queue

The Cache Manager maintains a queue of notifications for each session on a machine. To set the size of the queue, call **Notification.setQueueSize(int queueSize)**. ObjectStore throws NotificationException if a session subscribes to a notification before calling this method. The new queue size must not exceed the value specified for the **MAXIMUM_NOTIFICATION_QUEUE_LENGTH**.

Nothing forces a session to retrieve or read notifications. A session can subscribe to notifications but never retrieve any.

To avoid resource exhaustion in the Cache Manager, the size of the notification queue for each client is fixed. If the Cache Manager receives a notification and the queue is full, ObjectStore discards the notification. This is called an overflow. Overflows do not cause any exception to be signaled and do not cause the application, the Cache Manager, or the Server to crash.

The Cache Manager keeps statistics on the notification queue that include

- Queue size
- Number of pending notifications
- Number of overflows

To obtain these statistics, you can call the following **Notification** class methods:

- **public static int getQueueSize()**
- **public static getPendingNotifications()**
- **public static int getQueueOverflows()**

The information returned by these methods applies to only the session in which the method is called.

You can also use the ObjectStore utility **oscmstat** to obtain these statistics. The **ossvrstat** utility displays statistics on the number of notifications received and sent by the Server. See Chapter 4 in *ObjectStore Management*.

Performance Considerations

All notifications and subscriptions on a database go to the ObjectStore Server. The Server routes notifications to subscribed sessions through the Cache Manager, which queues the notifications for sessions. Because the Server acknowledges each notification, sending a notification requires a round-trip message to the Server.

Retrieving notifications only accesses shared memory and is very fast. If a session does not retrieve its notifications, the Cache Manager can run out of queue space. This causes the Cache Manager to discard notifications.

Every call to **subscribe()**, **unsubscribe()**, **notifyImmediate()**, and **notifyOnCommit()** requires one round-trip message to the ObjectStore Server.

If any commit-time notifications are queued during a transaction, there is an additional remote procedure call (RPC) to the ObjectStore Server during the commit operation.

Notifications are stored and forwarded in the Server, Cache Manager, and sometimes even in the receiving application. Therefore, delivery of notifications might not be particularly fast. Performance varies according to system load and the amount of notification processing. For example, delivery could range from milliseconds to several seconds.

As a general rule, if you plan your application to use notifications, you should not expect high throughput. Do not expect a client application to send or receive more than about ten notifications per second. In other words, do not use ObjectStore notification to meet high-speed requirements. The notification facility has a fairly simple design with limited buffering capability.

Network Service

When an ObjectStore application uses notifications, it automatically establishes a second network connection to the Cache Manager daemon on the local host. The application uses this connection to receive (and acknowledge the receipt of) incoming notifications from the Cache Manager. (Outgoing notifications are sent to the Server, not the Cache Manager.) See Chapter 1, Overview of Managing ObjectStore, in *ObjectStore Management* for specific information about defaults.

Chapter 12
Miscellaneous Information

This chapter provides miscellaneous information about ObjectStore.

Contents

This chapter discusses the following topics:

Java-Supplied Persistence-Capable Classes

Some Java-supplied classes are persistence-capable. Others are not persistence-capable and cannot be made persistence-capable. A third category of classes can be made persistence-capable, but there are important issues to consider when you do so.

Description of Java-Supplied Persistence-Capable Classes

The following Java classes are persistence-capable:

- **java.lang.String**
- The wrapper classes:
  - **java.lang.Boolean**
  - **java.lang.Byte**
  - **java.lang.Character**
  - **java.lang.Double**
  - **java.lang.Float**
  - **java.lang.Integer**
  - **java.lang.Long**
  - **java.lang.Short**
- Arrays of **Object**, of any of the primitive types (**boolean**, **byte**, **integer**, and so on), and of any persistence-capable type are all persistence-capable. (You can allocate an array and initialize it later, just as you would with any other field.)

Identity

ObjectStore does not always preserve identity for objects that are instances of the Java wrapper classes. It is more efficient to store these objects as values rather than as objects. Because identity is not always preserved, programs that use object identity to compare wrapper class objects work differently when used with persistent objects. For example, this method is incorrect:

```
boolean comparePersistIntegers(Integer x, Integer y) {
    return (x == y);
}
```

Instead, it should be written as

```
boolean comparePersistIntegers(Integer x, Integer y) {
    return x.equals(y);
}
```

Additional information about object identity is in About Object Identity on page 148.

VM overhead

When ObjectStore makes them persistent, **String** types, primitive wrapper types, and arrays have more runtime virtual memory overhead than types that implement **IPersistent**. This is because ObjectStore must create entries for these types in two hash tables. **COM.odi.IPersistent** requires an entry in a single hash table since some information is stored in fields in the object.

Persistent and persistence-capable

In your program, some wrapper objects might be persistent and some might be transient, though persistence-capable.

- If the application explicitly calls **ObjectStore.migrate()** on a wrapper object or stores it in a **COM.odi.coll** collection, the wrapper object becomes persistent.
- If the wrapper object is only reachable through transitive persistence, it does not become persistent when the transaction is committed. Instead, ObjectStore stores the object as an immediate value.

  This means that ObjectStore does not store the object in any of its internal hash tables and does not store the object as a separate value in the database. Instead, ObjectStore stores the object in the location of the reference to the object. The reference completely describes the object.

Any routine that requires a persistent object, as opposed to a persistence-capable object, notices the distinction between persistent and persistence-capable but transient. For example, if an application calls **Segment.of()** on an **Integer** object, the return value might be a segment in a database or ObjectStore might throw ObjectNotPersistentException. You cannot always predict what the return value will be because an **Integer**-valued field in a persistent object can contain either a persistent or transient value.

Unicode strings

ObjectStore stores Unicode strings. You can specify any Java string with Unicode characters in it, and ObjectStore can store it persistently and retrieve it correctly. ObjectStore uses UTF-8 encoding/compression to store regular English strings compactly. Sun's Java implementation uses the same mechanism.

Can Other Java-Supplied Classes Be Persistence-Capable?

There are many Java system classes that cannot be persistence-capable. There are other Java system classes that you can make persistence-capable, but you must consider some issues when you do so. In some situations, you can subclass the Java system class and make the subclass persistence-capable. Of course, this would not work for final classes.

Primitive types

You cannot store an object of a primitive type, such as an **int**, directly in a database as a discrete object. To store an object of a primitive type in a database you can

- Place it in a wrapper object, such as an **Integer**.
- Define it as a field in a persistence-capable class.

For example, you cannot make **byte** persistence-capable because all by itself, a **byte** is not an **Object**. But you can make **byte[]** persistence-capable because it is an **Object**.

Native methods

Classes that use native methods cannot be made persistence-capable by the postprocessor because the postprocessor cannot annotate the native methods the way it can annotate Java code. What this means is that if a class has native methods, and you postprocess the class, ObjectStore cannot guarantee that everything will work properly.

You might choose to postprocess your code and then add native code. If you do this, you must ensure that any persistent objects that your native code references are properly fetched from the database before the native method is called. Be careful, however, if a native method changes the value of an indexed instance variable. This does not work properly because the index is not updated.

Classes that hold state

Other system classes do not make sense as persistent objects because they hold state that is inherently tied to the process, such as open file channels or Java threads.

Postprocessing

For other classes, like **java.lang.Stringbuffer**, the above obstacles might not apply. If you postprocess the **.class** file for **java.lang.StringBuffer** and specify the **-modifyjava** option, the postprocessor produces a persistence-capable **StringBuffer** class:

```
osjcfp -dest \osjcfpout -modifyjava java.lang.StringBuffer
```

Then you must put the new **.class** file in your **CLASSPATH** variable ahead of the standard Java **.class** file. All subsequent use of the **StringBuffer** class in this environment would use the persistence-capable version.

Performance drawbacks

There are, however, some drawbacks to doing this. There will be some slowdown of some or all the methods, because the postprocessor must add new instructions to check whether the object needs to be brought in from the database or needs to be marked as modified.

How much slowdown is hard to determine. It depends on the details of the method. Even parts of your program that never handle persistence are affected by these extra instructions. This also applies to indirect uses of the class, for example, if **StringBuffer** is used heavily in some Java library that you are using, such as a user interface or network library.

Library version problems

There can also be problems with Java library version skew. If you postprocess **java.lang.StringBuffer** from version 1.1 of the Java Virtual Machine, and then your user uses your program with version 1.1.2, and **StringBuffer** has changed in some way between 1.1 and 1.1.2, your user will see the 1.1 version (persistence-capable) everywhere in the entire Java environment. If your user was depending, directly or indirectly, on the new 1.1.2 version of **StringBuffer**, something might not work properly.

Renaming the class

You might need to rename the newly created persistence-capable version so that the non-persistence-capable version is still available to the other Java system classes. To do this, specify the **-translatepackage** option when you run the postprocessor. See Putting Processed Classes in a New Package on page 268.

This avoids the problem and is generally safer. However, you might need the persistence-capable class to have the original class name. For example, suppose you have a library that has a method that takes an argument of type **java.lang.Stringbuffer**. You want to pass in a persistence-capable object. You cannot rename the class because the argument type would not match.

java.util.Hashtable

ObjectStore itself uses **java.util.Hashtable**. Consequently, invoking Java or using ObjectStore with a persistence-capable version of **java.util.Hashtable** that is available in your **CLASSPATH** is likely to cause trouble, such as infinite loops. A better approach is to substitute the ObjectStore-supplied class **COM.odi.util.OSHashtable**.

Description of Special Behavior of String Literals

There are special considerations when making **String** literals persistent.

When a Java program refers to a **String** literal by using quotation marks to name a string, Java treats the resulting **String** as a constant value. Multiple calls to a method with the **String** literal operate on the same **String** object.

The **COM.odi.stringPoolSize** initialization property allows you to control the way that ObjectStore causes **Strings**, other than literals with the same contents, to be represented by a single, shared instance in the database in certain circumstances. See Description of COM.odi.stringPoolSize on page 64.

Example of String Behavior

Consider the following example:

```
Object string() { String result = "string"; return result; }
Object intArray() { int[] result = { 1, 2, 3 }; return result; }
boolean stringsTheSame() { return string() == string(); }
boolean intArraysTheSame() { return intArray() == intArray(); }
```

The **stringsTheSame()** method always returns true because every call to **string()** returns the exact same **String** object. The **intArraysTheSame()** method always returns false because each call to **intArray()** constructs a new **int[]** object.

The behavior of **String** literals in Java has implications for making strings persistent. If a **String** literal becomes persistent, then subsequent calls to the method that contains the literal find that the string is already persistent. This can cause trouble if different calls to the method attempt to store references to the string in objects stored in different segments.

The following example demonstrates this problem:

```
void makeSomeObjects(Segment segment1, Segment segment2) {
    ObjectStore.migrate(makeObject(), segment1, false);
    ObjectStore.migrate(makeObject(), segment2, false);
}

Object makeObject() {
    Object[] result = new Object[1];
    result[0] = "string";
```

```
    return result;
}
```

The first call to **makeObject()** causes the **"string"** literal to be migrated into **segment1**. The second call to **makeObject()** tries to store a reference to **"string"**, which is now an unexported object in **segment1**, in an object in **segment2**.

The simplest solution to the problem is to avoid storing **String** literals in objects that become persistent.

The following example shows a modification to the **makeObject()** method that fixes this problem:

```
Object makeObject() {
    Object[] result = new Object[1];
    result[0] = new String("string");
    return result;
}
```

In this version, the **makeObject()** method stores a new **String** instance in each object that it returns. The result is that a new **String** instance is stored in the database for each call to **makeObject()**.

Another solution is to create an exported **String** object and have **makeObject()** always use that object. Decide on the approach you want to use according to the way you want to cluster objects in the database.

Another situation to consider is when an updated object refers to a **String** that has the same identity it had when the object was read from the database. ObjectStore does not store a new **String** in the database. If the updated object refers to a **String** with a different identity and that **String** is not stored in the database, ObjectStore migrates the **String** into the database. This is regardless of whether or not the database contains another **String** with the same contents.

As in Java, strings in ObjectStore are immutable. To change a string, you can destroy the old one and create a new one.

Destroying Strings

By default, **String** objects that become persistent during a transaction revert to being transient at the end of the transaction. Persistent objects are usually made stale at the end of a transaction. Unlike objects that implement **IPersistent**, when a **String** is made stale it becomes transient. As a result, the problem with making **String** literals persistent only occurs if the **String** literal is seen several times in the same transaction. If the **String** literal is only incorporated in a persistent data structure once in a transaction, then the problem does not occur.

When you destroy a **String**, in the transaction in which the destroy operation occurs, ObjectStore keeps track of the fact that the object was destroyed. An attempt to use a destroyed **String** literal causes ObjectStore to throw ObjectNotFoundException. The solution is to copy the **String** before you destroy it.

You should not destroy a **String** in a database unless you know that no other object in the database refers to that **String**. A safe, though possibly inefficient, way to handle this is to use

new String(String)

to force a new identity to each **String** that might be referenced. Also, you must disable the **String** pool by specifying **0** for the value of the **COM.odi.stringPoolSize** initialization property. This allows you to be sure that you can safely destroy the old **String** instance.

It is usually best to avoid destroying strings (or objects) altogether and let the persistent garbage collector take care of destroying such unreachable objects. The persistent garbage collector can typically destroy and reclaim such objects very efficiently, since it can batch such operations and cluster them effectively. If you set up the GC to run when the system is lightly loaded, you can effectively defer the overhead of the destroy operations to a time when your system would otherwise be idle, thus getting greater real throughput from your application when you really need it.

Serializing Persistent Objects

You can serialize many classes that implement **COM.odi.IPersistent**. For this to work, the definition of your persistence-capable class must implement the **java.io.Serializable** interface. The classes you can serialize include **COM.odi.util.OSVector** and **COM.odi.util.OSHashtable**.

During serialization, none of the transient fields in the **IPersistent** implementation need to be written out.

Before serializing an object, an application must always invoke **ObjectStore.deepFetch()** on the object to be serialized. The **deepFetch()** method ensures that the contents of all components of the object are accessible. This must be the case for an application to serialize an object.

Background about the necessity for **deepFetch()**

In an ObjectStore application, the first time you read or modify an object, ObjectStore makes the contents of the object available. The contents do not have to be available before you start the operation. You do not have to add Java code to make the contents available. When an ObjectStore program follows a reference from a source object to a target object, it automatically makes the contents of the target object available. This happens because the postprocessor recognizes the Java byte-code instructions that follow references and it inserts the code that fetches the object contents.

Serialization works differently. It follows references from one Java object to another without using Java byte codes. Serialization does not perform the automatic fetches the way that ObjectStore does. Consequently, before you initiate serialization of an object, its contents and the contents of all its components must already be available. The **ObjectStore.deepFetch()** operation does this for you.

Limitation

You cannot serialize Java peer objects. Consequently, you cannot serialize ObjectStore collection objects.

Example

When an application serializes and deserializes a persistent object with the default serialization methods, ObjectStore effectively creates a transient copy of the object and its components. Here is code that provides an example of serializing and deserializing persistent objects. In this example, **list2** is a transient copy of the persistent list.

```
public
class SerializationExample {

public static void main(String argv[])
    throws java.lang.ClassNotFoundException, java.io.IOException,
    java.io.FileNotFoundException {
    String dbName = argv[0];
    Session.createGlobal(null, null);
    /* Create a database with a list in it. */
    Database db = Database.create(dbname,
        ObjectStore.ALL_READ | ObjectStore.ALL_WRITE);

    Transaction tr = Transaction.begin(ObjectStore.UPDATE);
    List curr = new List("1", null);
    db.createRoot("list", curr);
    for (int i=2; i < 5; i++) {
        curr.next = new List(""+i, null);
        curr = curr.next;
    }
    tr.commit();

    /* Illustrate use of serialization in this example. */
    tr = Transaction.begin(ObjectStore.UPDATE);
    List head = (List)db.getRoot("list");

    /* Fetch the entire list prior to serializing it. */
    ObjectStore.deepFetch(head);

    FileOutputStream f = new FileOutputStream("tmp");
    ObjectOutputStream os = new ObjectOutputStream(f);
    os.writeObject(head);

    FileInputStream in = new FileInputStream("tmp");
    ObjectInputStream is = new ObjectInputStream(in);

    /* list2 is effectively a copy of the list denoted by head. */
    List list2 = (List)is.readObject();
    ...

    tr.commit();
    }
}

public class List implements java.io.Serializable {

    public Object value;
    public List next;

    List(Object value, List next) {
        this.value = value;
        this.next = next;
    }

    ...
}
```

Using Persistence-Capable Classes in a Transient Manner

The **stublib.zip** file contains stubs of ObjectStore classes that allow user-defined persistence-capable classes to be used in a purely transient manner. The annotations in the persistence-capable classes make calls to the various ObjectStore stub routines in **stublib.zip**.

The **stublib.zip** file provides a stripped down version of the ObjectStore API. This allows better performance and a smaller footprint than the complete zip file.

For example, you might want to use **stublib.zip** for the client in an RMI or CORBA application. The client might use persistence-capable classes, which make references to various ObjectStore methods, but the client never directly accesses a ObjectStore database. In this situation, the stub routines in **stublib.zip** satisfy the requirements of the Java VM's linker.

To use **stublib.zip**, put it in your **CLASSPATH** instead of **osji.zip**.

If your application uses any classes in **COM.odi.util**, you must use **osji.zip**. You cannot use **stublib.zip** because the stub definitions are not sufficient for the **COM.odi.util** classes.

Description of Java Persistent Storage Layouts

There are some differences between databases created by the Java and C++ interfaces to ObjectStore. These differences result in some restrictions on the use of databases by both the C++ and Java interfaces.

Databases created by the Java interface can be used by C++ programs, but the representation of Java primary objects is different from regular C++ objects. Because of these differences, accessing the contents of Java primary objects from C++ programs is not currently supported. C++ programs can store, access, and modify C++ objects in databases created by or modified by the Java interface.

Databases created by the C++ interface can be read or modified by the Java interface. C++ objects can be manipulated as described in the book *Developing ObjectStore Java Applications That Access C++*. In addition, the Java interface can store Java primary objects in databases created by the C++ interface.

Databases that hold Java primary objects have a segment that contains information used by the Java interface to describe the schema for the classes of the Java objects stored in the database. This segment also contains information about the location of other information used by the Java interface, which is stored in each segment that contains Java primary objects.

Segments that contain Java primary objects contain C++ data structures that describe the exported objects stored in the segment.

ObjectStore does not always align objects by page boundaries. Any Java object might cross page boundaries.

All Java primary objects contain a four-byte object header followed by data for the object fields. Java arrays are represented by a 12-byte header object and a separate C++ array that contains the array contents.

Fields of Java primary objects that contain object references are represented by an eight-byte or twelve-byte data structure rather than the four-byte pointer usually used by C++ objects. The larger data structure allows the Java interface to provide more features for object references. The eight-byte data structure is used for Java object reference fields that cannot contain arrays. This includes fields that are not of type **Object** or of an array type. The twelve-byte data structure is used for **Object** fields and fields of type array. The following table provides the exact element sizes.

Element Type	*Element Size*
byte	1
short	2
char	2
int	4
float	4
double	8
long	8
Object reference	8 or 12

Object references that are of type **java.lang.Object**, **java.land.Double**, **java.lang.Long**, or any array type, require 12 bytes. All other object references require eight bytes.

Differences Between C++ and Java Interfaces to ObjectStore

Here are some differences betweeen the Java and C++ interfaces to ObjectStore.

Timing of the Write Lock Acquisition

In the C++ interface to ObjectStore, as soon as you modify an object, you set (or try to set) the ObjectStore write lock for the page that the object is on. But in the Java interface to ObjectStore, this might or might not happen depending on the *lazy write locking* flag. The default is that it does *not* happen, and the write locking is deferred until later. So, if you have two sessions (in two VMs) that are accessing the same data and they are not both just reading, different timing of the write lock acquisition can cause the behavior to be different in the Java interface than it is in the C++ interface.

Opening the Same Database Multiple Times

In the Java interface to ObjectStore, each subsequent opening of a database after the initial open operation returns the same database object. For example:

```
db1 = Database.open("foo", ObjectStore.UPDATE);
db2 = Database.open("foo", ObjectStore.UPDATE);
```

In the Java interface to ObjectStore, the expression **db1 == db2** returns true. They refer to the same database object. Consequently, a call to **db1.close()** or **db2.close()** closes the same database. No matter how many times you open a database, a single call to the **close()** method closes the database.

This is different in the C++ interface to ObjectStore. In that interface, for example, if you call **open()** four times and **close()** three times all on the same database, the database is still open.

Environment Variables

ObjectStore includes the following environment variables:

- **OS_JAVA_VM** specifies the command for running the Java virtual machine, when it is set. The default is that this variable is not set. The Windows tool batch files use the value of this variable when it is set.
- **OSJCFPJAVA** specifies the name of the Java executable you want the postprocessor to use. The default is **java**. If this variable is not set, the postprocessor uses the first Java executable that it finds in your **PATH** environment variable If you want the postprocessor to use some other Java executable, set the **OSJCFPJAVA** environment variable to the name of the Java executable you want the postprocessor to use.

  If the postprocessor cannot find a Java executable, it generates a Bad command or file name error message.

Chapter 13
Tools Reference

This chapter provides reference information for the following tools:

osgc: Collecting Garbage in Databases

The command line utility for collecting garbage is **osgc**. Invoke this tool with the following format:

osgc [*options*] *database_name*

For example, execution of

osgc db

is the same as calling the **Database.GC()** method on the **db** database. You can specify the following options:

Option	***Description***
-seg *segment_id*	Collects garbage from only the specified segment. By default, the **osgc** utility operates on the entire database.
-retries *number*	Indicates the number of times the tool tries to resume the sweep phase of garbage collection after it waits for a lock. The default is **10**. This is identical to the **COM.odi.gc.retries** property.
-retryInterval *interval*	Indicates the number of milliseconds the sweep operation waits between sweep attempts for a concurrency conflict to be resolved before it tries to resume the sweep. The default is **1000**. This is identical to the **COM.odi.gc.retryInterval** property.
-lockTimeOut *interval*	Indicates the number of milliseconds the sweep operation waits for a lock conflict to be resolved. If it is not resolved in the specified length of time, the tool aborts the current transaction and starts a new transaction. ObjectStore rounds this value up to the nearest second. The default is **1000**. This is identical to the **COM.odi.gc.lockTimeOut** property.
-transactionPriority *n*	Specifies the transaction priority associated with transactions started by the tool. The Server uses this specification when it must determine which transaction must be the victim in a deadlock. This number is intentionally low so that the garbage collection transaction is the deadlock victim of choice. The default is **0**. This is identical to the **COM.odi.gc.transactionPriority** property.

Option	*Description*
-displayGarbage *level*	Displays information about the candidates for garbage collection instead of actually destroying the candidates. The level you specify determines the amount of information the tool displays. **1** lists the number of objects per segment that would be destroyed. **2** is not currently supported. **3** lists the location of each GC candidate. **4** lists the roots of garbage graphs. Level **4** can require intensive computations.
-statistics	Displays statistics for the garbage collection operation. This includes the total number of reachable objects and the total number of garbage objects.

The name of the garbage collection utility is **osgc** and not osjgc because this tool works on databases created with the C++ interface to ObjectStore, as well as databases created with the Java interface to ObjectStore.

See also Running osgc on C++ Databases or Segments on page 84.

osjbrowsedb: Browsing a Database

This release includes a beta version of a database browser. For information about how to use the browser, see the **browser.htm** file in the **doc** directory of your ObjectStore installation directory.

osjcfp: Running the Postprocessor

To make classes persistence-capable, compile the source files and then run the postprocessor on the resulting class files. You must run the postprocessor on all class files in a batch at the same time. The postprocessor can accept a command line that intersperses file names, options, and input file specifications. Complete information about the postprocessor is in Chapter 8, Automatically Generating Persistence-Capable Classes, on page 235.

The command format is

osjcfp -dest *destination_dir file_name* [*file_name*...] [*options*]

The following table describes the options you can specify.

Option	***Description***
@*input_file*	Causes the contents of the named input file to replace this argument in the command line. The postprocessor does this before any other argument processing. You can specify this option multiple times on one command line to include multiple files. You cannot nest this option. That is, the postprocessor does not expand this argument if it appears in an input file.
{ **-a** | **-arraydims** } *num_array_dimensions*	Specifies the maximum number of dimensions that ObjectStore allows for persistent arrays of objects whose classes are annotated during this execution of the postprocessor. If you do not specify this option, the default is three dimensions.
{ **-cis** | **-classinfosuffix** } *suffix_string*	Specifies the suffix that the postprocessor adds to the name of the **ClassInfo** subclass that the postprocessor generates for each class it makes persistence-capable. By default, the suffix is **ClassInfo**. This is useful when you need to limit the number of characters in file names. For all batches in an application, you must specify the same suffix if you do not use the default.
{ **-cpath** | **-classpath** } *class_path*	Specifies the path by which to locate class files for postprocessing. If you specify this option, ObjectStore uses it in place of the **CLASSPATH** environment variable. The **-classpath** option does not affect the class path used for the execution of the postprocessor.

Option	*Description*
{ **-cc** | **-copyclass** }	Copies classes to the destination directory without annotating them. This option applies to class names, **.class** files, **.jar** files, and **.zip** files that you specify on the command line after the **-copyclass** option and before the next **-persistcapable** or **-persistaware** option or the end of the command line. This option is useful when you want nonpersistent classes or classes that have already been annotated to be in the same directory as persistence-capable or persistence-aware classes being created. If you specify the **-persistaware** or **-persistcapable** option for any file for which you also specify the **-copyclass** option, the postprocessor ignores the **-copyclass** option for that file. If you specify the **-translatepackage** option and the **copyclass** option, the postprocessor modifies the class to accommodate the new package name.
{ **-d** | **-dest** } *destination_dir*	This option is required. The postprocessor uses the directory you specify for *destination_dir* as the root for locating the annotated files. The postprocessor places each class file it operates on in the package-appropriate subdirectory of the destination directory, as though the destination directory were in your class path. If the destination directory specification would cause the postprocessor to overwrite an original file, and you did not specify the **-inplace** option, the postprocessor reports an error and terminates without producing any output.
{ **-f** | **-force** }	Forces the postprocessor to overwrite existing annotated **.class** and **ClassInfo** files.
-hashcode *class_name*	Causes the postprocessor to add a persistent **hashCode()** method to the specified class. You typically use this option with the **-nodefaulthashcode** option. If you specify this option for a class for which you explicitly defined a **hashCode()** method, the postprocessor reports an error.
-includesummary *inc_class_name*	Instructs the postprocessor to include the specified summary in the new summary it creates. It does not matter whether or not the postprocessor is also annotating any classes. Replace *inc_class_name* with the name of a file generated by a previous execution of the postprocessor with the **-summary** option. You must know the name of the generated file. If you did not postprocess the library yourself, you must get the name of the generated class from the library vendor. When you specify the **-includesummary** option, you must also specify the **-summary** option.

Option	*Description*
{ **-index** | **-indexablefield** } *field*	Marks a field as indexable for a peer (**COM.odi.coll)** collection. This option applies only to the field that immediately follows it. You must specify a fully qualified field, for example, **COM.odi.demo.people.name**. You can specify this option multiple times. This option does not apply to utility (**COModi.util**) collections. This option is useful because it allows a query to run faster. The postprocessor does not actually add the index. You can add a persistent index with a call to the API at run time. When the index is present, queries that use the specified field are faster. Suppose you declare a field to be indexable and then you change the value of that field. Performance is slightly slower than if the field were not indexable. This is true for any object of the class, whether or not the object is in a collection. Now suppose an object with an indexable field is in a collection and the collection has an index on the indexable field. If you change the value of the field, performance is slightly slower than when the object is not in a collection. The extra time is needed to update the index. It is possible for an object to belong to many collections. Each collection can have an index on a particular field. If you change the value of that field, ObjectStore must update each index and the performance penalty is greater. If a class has any indexable fields, every instance of the class is larger by three 32-bit words in the database.
-inplace	Causes the postprocessor to annotate standalone files (files that are not in **.zip** files or **.jar** files) in place rather than writing the annotated file in the destination directory. When the postprocessor annotates a class in place, it overwrites the original class files with the annotated class files, and writes the **ClassInfo** subclass to the same directory as the persistence-capable class. If a class originates in a **.zip** file or **.jar** file, the postprocessor writes the annotated class and its corresponding **ClassInfo** subclass to the destination directory. Do not use this option when you are doing iterative development. During development, a separate output directory avoid errors and supports debugging. When you specify the **-inplace** option, you must still specify a destination directory, but the postprocessor ignores it.

Option	*Description*
{ **-it** | **-ignoretransient** } *field_name*	Instructs the postprocessor to ignore the transient attribute of the specified field and treat the field as a persistence-capable field. You must specify a fully qualified field name. The field is treated as persistence-capable only for the purposes of postprocessing. This option is useful when a persistence-capable class you are defining inherits from a class that includes a transient field. If you do not specify this option for a transient field, the postprocessor ignores the field, which can cause problems if you want to use the field.
-modifyjava	Allows the postprocessor to modify classes in standard Java packages. The default is that the postprocessor does not modify standard Java classes.
{ -naf | -noannotatefield } *qualified_field_name*	Prevents access to the specified field from causing **fetch()** and **dirty()** calls on the containing object. This is useful for transient fields when you access them outside a transaction. Normally, access to a transient field causes **fetch()** or **dirty()** to be called to allow the **postInitializeContents()** and **preFlushContents()** methods to convert between persistent and transient state.
-noarrayopt	Disables optimization of **fetch()** and **dirty()** calls for array objects in looping constructs. This causes **osjcfp** to insert the calls to **fetch()** or **dirty()** in every iteration rather than only in the first loop iteration.
{-nodefaulthashcode | -ndhc }	Prevents the postprocessor from automatically adding a **hashCode()** method to a class, except for classes for which you explicitly specify the **-hashcode** option. If you specify this option, it is your responsibility to ensure that there is a suitable **hashCode()** method for classes that are used as keys in persistent hash tables.
-noinitializeropt	Disables optimization of **fetch()** and **dirty()** calls in constructors. Specify this option when you want the postprocessor to perform full annotation of constructors. Full annotation means that if the object becomes persistent during constructor execution, modifications to the object are correctly handled. By default, the postprocessor does not fully annotate constructors to handle changes in the newly constructed object. Typically, this is the desired behavior. If your application inserts objects into ObjectStore collections during construction of the objects being inserted, you must specify the **-noinitializeropt** option. Doing so avoids errors in the handling of modifications to the newly constructed objects.

Option	*Description*
-noopt	Disables the three optimizations that are disabled by the **-noarrayopt**, **-noinitializeropt**, and **-nothisopt** options. The **-noopt** option is a shortcut you can use when you want to specify all three options. You might want to specify this option when the optimizations are preventing the postprocessor from inserting required **fetch()** and **dirty()** calls in your classes.
-nothisopt	Disables optimization of **fetch()** and **dirty()** calls for access to fields relative to **this** in nonstatic member methods. This causes **osjcfp** to insert a **fetch()** or **dirty()** call for each access to a field in **this**.
-nowrite	Performs process and error checking but does not actually annotate class files. This option allows a test run of the postprocessor. You use it to determine whether or not all specified classes are accessible, whether or not additional options are needed, and if you specify **-v** (verbose) you can see where the resulting files would be located.
-optimizeclassinfo | -oci	Prevents the postprocessor from generating *xxx***ClassInfo.java** files for public and abstract classes. This reduces the disk footprint and application startup times, since there are fewer classes to load when the application starts. When the postprocessor does not create a **ClassInfo** class, it uses the Java reflection API instead. Some of the reflection API is subject to security and access constraints that are enforced to varying degrees depending on the version of the JDK and the platform. In other words, you can use the **-optimizeclassinfo** option if the Java environment in which you intend to run the application does not restrict the use of the reflection API.
{ **-pa** | **-persistaware** }	Causes subsequent **.class** files, **.jar** files, and **.zip** files on the command line to be persistence-aware. This means that instances of the classes can operate on persistent objects but cannot be persistent. The postprocessor annotates persistence-aware classes so that there are calls to **ObjectStore.fetch()** and **ObjectStore.dirty()** where needed during operations on potentially persistent objects and arrays that might be used by the persistence-aware class. This option applies to class names, **.class** files, **.jar** files, and **.zip** files that you specify on the command line after the **-persistaware** option and before the next **-persistcapable** or **-copyclass** option or the end of the command line. In other words, the **-persistcapable** option or the **-copyclass** option alters this mode. The **-pc** option is in effect by default.

Option	*Description*
{ **-pc** | **-persistcapable** }	Causes subsequent **.class** files, **.jar** files, and **.zip** files on the command line to be persistence-capable. This option applies to class names, **.class** files, **.jar** files, and **.zip** files that you specify on the command line after the **-persistcapable** option and before the next **-persistaware** or **-copyclass** option or the end of the command line. The **-pa** (persistence-aware) option or the **-copyclass** option alters this mode. The **-pc** option is in effect by default.
{ **-q** | **-quiet** }	Causes the postprocessor to refrain from displaying warnings. A warning message provides information about something that the postprocessor recognizes as a possible problem, but cannot confirm as actually being a problem. This option cancels a previous **-verbose** option, if you specified one.
{ **-qc** | **-quietclass** } *class_name*	Causes the postprocessor to refrain from displaying warnings for the specified class. A warning message provides information about something that the postprocessor recognizes as a possible problem, but cannot confirm as actually being a problem. This option applies only to the name that immediately follows it. Specify a fully qualified class name. If the postprocessor is renaming the class, it does not matter whether you specify the old name or the new name. If you specify **-verbose** in the same command, this option takes precedence for the specified class.
{ **-qf** | **-quietfield** } *member_name*	Causes the postprocessor to refrain from displaying warnings for the specified class field. A warning message provides information about something that the postprocessor recognizes as a possible problem, but cannot confirm as actually being a problem. This option applies only to the name that immediately follows it. Specify a fully qualified class field name. If the postprocessor is renaming the class, it does not matter whether you specify the old name or the new name. If you specify **-verbose** in the same command, this option takes precedence for the specified class field.

Option	*Description*
-summary *gen_class_name*	Causes the postprocessor to generate a class with the name *gen_class_name*. This generated class extends the **COM.odi.PersistentTypeSummary** class. The *gen_class_name***.getPersistentClasses()** method returns a list of the classes that were made persistence-capable in this execution of the postprocessor. The generated class has a no-argument constructor that passes arrays to the **PersistentTypeSummary** class constructor. To identify persistence-capable classes in libraries and include them in this summary, specify the **-includesummary** option.
{ -tf | -transientfield } *qualified_field_name*	Causes the postprocessor to treat the specified field as though it has a **transient** modifier, even if it does not. This is typically useful when a field should not be stored in a database, but it must be available for object serialization.
{ **-tp** | **-translatepackage** } *orig_pkg_name new_pkg_name*	Renames classes that belong to *orig_pkg_name* so that they belong to *new_pkg_name*. The original **.class** files remain in the original location and the postprocessor does not annotate them. For example, suppose the postprocessor makes a class named **a.b.C** persistent with **-tp a.b a.b.x**. The persistent class has the name **a.b.x.C**. A package specification of "**.**" implies the default unnamed package. For example, the option **-tp . persist** causes the unpackaged class name **C** to be renamed **persist.C**. *orig_pkg_name* must exactly match the package name of the class being annotated. For example, for a file named **a.c.D**, a specification of **-tp a a.b** does not translate the package name. The package of **a.c.D** is **a.c**, not **a**. The postprocessor changes the package name of all classes in the original package that it can locate through the **CLASSPATH** environment variable or, if it is specified, the **-classpath** option.
{ **-v** | **-verbose** }	Causes the postprocessor to write descriptions of its actions to standard output. This option cancels a previous **-quiet** option, if you specified one.

You can specify any number of class names, **.class** files, **.jar** files, or **.zip** files on the command line. The postprocessor recognizes files as follows:

File Name	*Recognized As*
name.**class**	Explicit file name for a class file.
name.**zip**	Explicit file name for a class **.zip** file. The postprocessor processes all **.class** files in the **.zip** file according to the persistence mode that is in effect when the postprocessor encounters the name of the **.zip** file. The postprocessor places each file from the **.zip** file in the package-relative subdirectory of the destination directory. A **.zip** file allows the postprocessor to process multiple files without the specification of each one on the command line. Also, you can simply specify a **.zip** file. You do not need to unzip the file before processing the **.class** files.
name.**jar**	Explicit file name for a class **.jar** file. The postprocessor treats the file the same way that it treats a **.zip** file.
name	Qualified class name delimited by "**.**". The postprocessor uses the **CLASSPATH** environment variable or the specification for the **-classpath** option to locate the **.class** file, which can be in a **.zip** file or **.jar** file.

Because the postprocessor recognizes *name*.**class** as well as *name*, you can run a command such as

```
osjcfp -dest osjcfpout *.class
```

You do not need to derive qualified class names from the file paths.

osjcgen: Generating Peer Classes

To map C++ classes to Java peer classes, run the peer generator tool (**osjcgen**).

Additional information about the peer class generator is in the book *Developing ObjectStore Java Applications That Access C++*.

Description of Command Line Format

The command line format appears below. It shows the different components in the command line on different lines only for clarity.

osjcgen -package *destination_package*
-native_interface *interface_type*
-schema *mapping_schema_database*
-classdir *target_dir* **-libdir** *target_dir* [*options*] *class_list*

-package *destination_package*
Identifies the package in which the peer generator creates peer classes. Required.

-native_inteface *native_interface*
Specifies the virtual machine interface for which the tool should generate C++ glue code. Specify **jni** for the Sun VM or **ms_raw** for the Microsoft VM.

-schema *mapping_schema_database*
Identifies the mapping schema database from which the tool extracts type definitions. Required.

You must have specified the **-store_member_functions** (**-smf**) and **-store_function_parameters** (**-sfp**) options when you ran **ossg** to create this schema. If you did not, the tool cannot define peer methods. These options cause additional information to be stored in the schema database. **osjcgen** needs this information to correctly process member functions.

The mapping schema database you specify is not the application schema database that the application uses.

If the C++ code for which you want to define peer classes does not define methods, you need not specify the **-smf** and **-sfp** options when you run **ossg**.

-classdir *target_dir*	Specifies the directory relative to which the tool writes the peer classes and other generated Java files. For example, if the specified *destination_package* is **MyPkg** and the specified *target_dir* is **MyDir**, the tool creates the peer classes in **MyDir/MyPkg**. The directory you specify must exist. If the directory you specify does not contain a subdirectory with the name of the package, the tool creates this subdirectory. Required.
-libdir *target_dir*	Specifies the directory in which the tool creates the C++ files. In the directory you specify, the tool creates a directory that has the name you specify for *destination_package*. The tool places the C++ files in this package subdirectory of the directory you specify with the **-libdir** option. The directory you specify must exist. Required.
options	Any of the options described in the next table. Optional.
class_list	Lists the names of the classes and structs for which you want the tool to generate peer classes. The tool places each peer class in the specified destination package. You can intersperse class name specifications with the options described in the next table. Required. Do not specify **enum**s. If a peer method accepts or returns an **enum**, the tool automatically generates the peer class for the **enum**.

Description of Additional Options

The following table describes the additional options you can specify on the **osjcgen** command line. These options can precede, follow, or be interspersed with the required **osjcgen** options.

-boolean *typedef_name*	Indicates that the specified **typedef** name identifies a **boolean** data type. You can specify this option multiple times.
-classpath *class_path*	Instructs the tool to use the *class_path* you specify rather than the setting for the **CLASSPATH** environment variable. When you specify this option, it affects only the lookup of classes by the peer generator tool. It has no effect on the execution of the peer generator tool itself.
-full	Turns off the **-leaf** option. The tool generates complete peer classes for types specified subsequently. The **-full** option is in effect by default.

-import *package*

Specifies a package name in which to look for a peer class before creating it. For example, suppose you want to generate a peer class for the C++ class **X**. The tool looks in the package specified with the **-import** option for a Java version of **X**. If such a version is found, the tool does not generate a new peer class for **X**. The application can use the existing peer class. You can specify this option multiple times.

@*input_file*

You can use the @*input_file* option to specify a file that contains arguments for the peer generator tool. The tool inserts the contents of the specified file in the command line before it begins to execute the command line. You can specify this option multiple times. An input file cannot itself include the @*input_file* option. If it does, the tool treats it as the name of a class, which is not found.

-leaf

Instructs the tool to generate a minimal definition for the peer classes of types specified after this option and before any **-full** option. When the tool generates a leaf peer class, the application cannot access C++ data or function members or any C++ base classes. The tool defines the Java peer class to inherit directly from **CPlusPlus**. It does not copy the inheritance structure from the C++ class. This option is useful when you want to prune a type graph to reduce the size of the interface code.

-map *C++_name Java_peer_name*

Allows you to specify the name of the Java peer class that you want to identify a particular C++ class. When you specify this option, you need not rely on the default name mapping rules. This option is particularly useful for naming template classes. You can specify the **-map** option as many times as you need to. Follow each specification with

1 The name of a C++ class. You must also specify the name of the C++ class in the list of classes for which you want to generate peer classes.

2 The name of the Java peer class that you want to represent the C++ class. This can be a fully qualified class name or an unqualified class name. If it is unqualified, the peer generator tool places the Java peer class in the package you specify on the command line.

-map_existing
C++_name
Java_peer_name

Allows you to specify the name of an existing Java peer class that you want to identify a particular C++ class. This option is the same as the **-map** option, except that the Java peer class already exists or will exist. Consequently, the peer generator tool does not generate any peer code for the Java peer type. In a separate run of the peer generator tool, or manually, you must create the Java peer type. See the documentation for **-map** for additional details.

-nosynchronize

Turns off automatic synchronization of peer methods. If you specify this option, your application is responsible for ensuring that only one thread at a time per session is accessing ObjectStore. Failure to prevent concurrent access to OSJI and peer method entry points can cause OSJI to fail.

-oldtemplates

Causes the peer generator to map some C++ characters to Java equivalents used in previous releases of ObjectStore. In *Developing ObjectStore Java Applications That Access C++*, Chapter 2, Rules for Template Name Flattening shows which C++ characters are invalid in Java and which Java equivalents they are mapped to.

-synchronize

Turns on automatic synchronization of peer methods. This is the default.

-suppress
package.*class*.*method*

Suppresses generation of the specified peer method. You should not need to specify this option frequently. However, if generated code for a particular method causes a problem for the compiler, this option allows you to prevent generation of that code.

You must specify the package name, the class name, and the method name. Use this option in the following manner:

1 Run **osjcgen** without specifying this option.

2 Try to compile the generated code.

A particular peer method causes a problem.

3 Run **osjcgen** again and specify the same classes, but suppress generation of the peer method that causes the problem.

4 Determine how to work around the lack of the suppressed peer method.

If it is not possible to work around the method that is causing the problem, you must redefine the C++ method into a form that does not cause a problem when it is mapped to a peer method.

Example of Running the Peer Generator Tool

The following example generates a Java peer class for the **CPerson** class. If it does not already exist, the tool creates the **MyPkg** subdirectory in the **MyDir** directory. The tool puts the Java peer class in the **MyPkg** package/subdirectory. The mapping schema database that the tool uses to generate the peer class is **CPerson.jadb**. The tool places the generated C++ files (and the generated Java files) in the **MyDir/MyPkg** directory.

```
osjcgen -package MyPkg
    -native_interface jni
    -schema CPerson.jadb
    -classdir MyDir
-   libdir MyDir
CPerson
```

If you specify

```
-libdir SomeOtherDir
```

the tool places the generated Java files in **MyDir/MyPkg** and the generated C++ files in **SomeOtherDir/MyPkg**.

Caution

If the tool generates files that have the same pathnames as your C++ source files, the tool overwrites the C++ source files without warning you. This can happen if you create your C++ source files in the package and directory that you specify with the **-package**, **-classdir**, and/or **-libdir** options to **osjcgen**.

For example, suppose you have the **CPerson.cc** C++ file in the **/MyDir/MyPkg** directory. When you run **osjcgen**, you specify the following on one line:

```
osjcgen
    -package MyPkg
    -native_inteface jni
    -schema CPerson.jadb
    -classdir MyDir
    -libdir MyDir
CPerson
```

The peer generator tool generates these files:

```
/MyDir/MyPkg/CPerson.java
/MyDir/MyPkg/CPersonU.java
/MyDir/MyPkg/CPersonClassInfo.java
/MyDir/MyPkg/CPerson.cc
/MyDir/MyPkg/CPersonU.cc
```

An explanation of these files appears in *Developing ObjectStore Java Applications That Access C++*, Chapter 2, Overview of Tool Output. But as you can see, the tool would overwrite your **CPerson.cc** source file. On Solaris, you can work around this by specifying **.CC** instead of **.cc** in the C++ file name. On Windows, you can work around this by specifying **.cpp** instead of **.cc** in the C++ file name. Alternatively, you can either specify different directories for **-classdir** and / or **-libdir,** or you must not create C++ source files in the Java package subdirectory.

osjcheckdb: Checking References in a Database

The **osjcheckdb** utility or the **Database.check()** method checks the references in a database. This tool scans a database and checks that there are no references to destroyed objects. You can fix references to destroyed objects by finding the objects that contain the dangling references and overwriting the invalid references with something else, such as a null value. In addition to finding references to destroyed objects, the tool performs various consistency checks on the database.

If the tool does not find any problems, it does not produce any output.

Check paths

Before you invoke **osjcheckdb** from the command line, ensure that **tools.zip** or **tools.jar** is in your **CLASSPATH** variable. Also ensure that the distribution **bin** directory that contains **osjcheckdb** is in your **PATH** variable. The format for invoking this tool from the command line is

Command line

osjcheckdb *database_name1***.odb** ...

You can specify one or more databases. Separate multiple specifications with a space. If **osjcheckdb** cannot check a database that you specify, it displays a message about the inaccessible database and continues to the next database.

Be sure to specify the name of the **.odb** file of the database.

The tool displays messages about any errors that it finds.

API

The function signature for invoking the API for this tool is

Database.check(java.io.PrintStream)

When ObjectStore executes this method, it operates on the committed contents of the database and on any changes that have been saved as a result of **ObjectStore.evict()** or **ObjectStore.evictAll()**. ObjectStore does not operate on any changes that have been made but not committed or evicted.

The method writes any errors it finds to the argument stream. It also returns a Boolean value, which is **true** if the references are valid and **false** if there are any bad references.

The **osjgcdb** tool requires

- The **tools.zip** or **tools.jar** in your **CLASSPATH** environment variable.
- The ObjectStore **bin** directory in your **PATH** environment variable.

osjshowdb: Displaying Information About a Database

The **osjshowdb** utility displays information about one or more databases. This utility is useful when you want to know how many and what types of objects are in a database. You can use this utility to verify the general contents of the database.

This utility displays the following information:

- Name of the database
- Size of each segment
- Name and number of each type of object in the database
- Total size in bytes occupied on the disk by each type of object
- Number of destroyed objects

-showObjs option

If you specify the **-showObjs** option, the **osjshowdb** utility also displays the following information for each object:

- **oid**, which is an internal representation of its location in the database
- Type
- Number of bytes it occupies on the disk
- If it is an array, the number of elements in the array

-showData option

Specify the **-showData** option with **osjshowdb** to display string values as well as the references an object contains. When you specify the **-showData** option, it implies the **-showObjs** option.

Path variables

Before you invoke **osjshowdb** from the command line, ensure that **tools.zip** or **tools.jar** is in your **CLASSPATH** environment variable. Also ensure that the distribution **bin** directory that contains **osjshowdb** is in your **PATH** variable.

Command line

To execute the **osjshowdb** utility, use this format:

osjshowdb [**-showData**] [**-showObjs**] *db1*.**odb** [*db2*.**odb**]...

You can specify one or more databases.

When the utility displays **java.lang.String** objects, the number of elements is the number of characters in the string. The total bytes indicates the number of bytes that the data occupies on the disk.

There are some internal structures in the database that are not included in the calculations performed by the **osjshowdb** utility. Consequently, the total number of bytes as indicated in the output from **osjshowdb** is never equal to the actual size of a segment.

API

The API for the **osjshowdb** utility is **Database.show()**.

When ObjectStore executes this method, it operates on the committed contents of the database and on any changes that have been saved as a result of **ObjectStore.evict()** or **ObjectStore.evictAll()**. ObjectStore does not operate on any changes that have been made but not committed or evicted.

Sample output **customers.odb**

Here is some sample output from the **osjshowdb** utility for the **customers** and **problems** databases produced by the **tracker** demo.

osjshowdb -showObjs customers.odb
Name: customers.odb

There are 2 roots:

Name: _DMA_Database_header Type: _DMSS_Database_header
Name: customers Type: COM.odi.coll.imp.DictPeer_String

Destroyed Objects: 4

Segment: 0
Size: 48128 (47 Kbytes)

OID	Data Offset	Elements	Total Bytes	Type

Count	Tot Size (bytes)	Type

Segment: 2
Size: 8192 (8 Kbytes)

OID	Data Offset	Elements	Total Bytes	Type
<1|2|256>	256		24	COM.odi.coll.imp.DictPeer_String

<1|2|452>	452	18	32	java.lang.String
<1|2|520>	520	32	396	java.lang.Object[]
<1|2|928>	928		16	COM.odi.util.OSVectorEntry
<1|2|944>	944	1	20	COM.odi.util.OSVectorEntry[]
<1|2|1008>	1008	17	32	java.lang.String
<1|2|1040>	1040		36	COM.odi.demo.tracker.Customer
<1|2|1076>	1076	32	396	java.lang.Object[]
<1|2|1480>	1480		16	COM.odi.util.OSVectorEntry
<1|2|1496>	1496	1	20	COM.odi.util.OSVectorEntry[]
<1|2|1528>	1528	44	60	java.lang.String
<1|2|1588>	1588		36	COM.odi.demo.tracker.Customer

Count	Tot Size (bytes)	Type
1	24	COM.odi.coll.imp.DictPeer_String
2	56	COM.odi.demo.tracker.Customer
2	56	java.lang.String

Segment: 4
Size: 4096 (4 Kbytes)

OID	Data Offset	Elements	Total Bytes	Type

Count	Tot Size (bytes)	Type

problems.odb

osjshowdb -showObjs problems.odb

```
Name: problems.odb

There are 3 roots:

  Name: _DMA_Database_header   Type: _DMSS_Database_header
  Name: problems   Type: COM.odi.coll.imp.DictPeer_String
  Name: reportSegmentId   Type: java.lang.Integer

Segment: 0
Size: 54272 (53 Kbytes)

                                                Total
OID    Data Offset   Elements  Bytes   Type

   Count        Tot Size     Type
                (bytes)

Segment: 2
Size: 8192 (8 Kbytes)

                                                Total
OID    Data Offset   Elements  Bytes   Type

<2|2|256>    256                 24   COM.odi.coll.imp.DictPeer_String
<2|2|452>    452          18     32   java.lang.String
<2|2|484>    484                 36   COM.odi.demo.tracker.Problem
<2|2|520>    520          32    396   java.lang.Object[]
<2|2|928>    928                 16   COM.odi.util.OSVectorEntry
<2|2|944>    944           1     20   COM.odi.util.OSVectorEntry[]
<2|2|1008>  1008          17     32    java.lang.String
<2|2|1040>  1040                 36   COM.odi.demo.tracker.Problem
<2|2|1076>  1076          32    396   java.lang.Object[]
<2|2|1480>  1480                 16   COM.odi.util.OSVectorEntry
<2|2|1588>  1588                 36   COM.odi.demo.tracker.Problem

   Count        Tot Size     Type
                (bytes)

        1          24      COM.odi.coll.imp.DictPeer_String
        3         108      COM.odi.demo.tracker.Problem
        2          64      COM.odi.util.OSVector
        2          32      COM.odi.util.OSVectorEntry
        2          40      COM.odi.util.OSVectorEntry[]
        3         124      java.lang.String
Segment: 4
Size: 4608 (5 Kbytes)

                                                Total
OID    Data Offset   Elements  Bytes   Type

<1|4|1616>   1616                  8   java.lang.Integer

   Count        Tot Size     Type
                (bytes)

        1          8       java.lang.Integer
```

Segment: 6
Size: 9216 (9 Kbytes)

OID	Data Offset	Elements	Total Bytes	Type
<1|6|288>	288	38	52	java.lang.String
<1|6|340>	340		20	COM.odi.demo.tracker.Report
<1|6|360>	360	32	396	java.lang.Object[]
<1|6|768>	768		16	COM.odi.util.OSVectorEntry
<1|6|784>	784	1	20	COM.odi.util.OSVectorEntry[]
<1|6|816>	816	100	116	java.lang.String
<1|6|932>	932		20	COM.odi.demo.tracker.Report
<1|6|952>	952	25	40	java.lang.String
<1|6|992>	992		20	COM.odi.demo.tracker.Report
<1|6|1012>	1012	12	28	java.lang.String
<1|6|1040>	1040		20	COM.odi.demo.tracker.Report
<1|6|1092>	1092	78	92	java.lang.String
<1|6|1184>	1184		20	COM.odi.demo.tracker.Report
<1|6|1204>	1204	32	396	java.lang.Object[]
<1|6|1608>	1608		16	COM.odi.util.OSVectorEntry
<1|6|1624>	1624		20	COM.odi.util.OSVectorEntry[]
<1|6|1656>	1656	80	96	java.lang.String
<1|6|1752>	1752		20	COM.odi.demo.tracker.Report
<1|6|1772>	1772	56	72	java.lang.String
<1|6|1844>	1844		20	COM.odi.demo.tracker.Report
<1|6|1864>	1864	76	92	java.lang.String
<1|6|1956>	1956		20	COM.odi.demo.tracker.Report
<1|6|2008>	2008	62	76	java.lang.String
<1|6|2084>	2084		20	COM.odi.demo.tracker.Report
<1|6|2104>	2104	32	396	java.lang.Object[]
<1|6|2512>	2512		16	COM.odi.util.OSVectorEntry
<1|6|2528>	2528	1	20	COM.odi.util.OSVectorEntry[]
<1|6|2560>	2560	109	124	java.lang.String
<1|6|2684>	2684		20	COM.odi.demo.tracker.Report

Count	Tot Size (bytes)	Type
10	200	COM.odi.demo.tracker.Report
3	96	COM.odi.util.OSVector
3	48	COM.odi.util.OSVectorEntry
3	60	COM.odi.util.OSVectorEntry[]
3	1188	java.lang.Object[]
10	788	java.lang.String

osjuphsh: Upgrading String Hash Codes in Databases

The **osjuphsh** utility is the command line utility for upgrading databases to use the correct **String** hash code. Invoke this tool with the following format:

osjuphsh [*options*] *database_name*

For example:

osjuphsh db.odb

Execution of this command is the same operation as a call to the **COM.odi.Upgrade.upgradeDatabaseStringHash()** method on the **db.odb** database with the **upgradeObjects** argument set to true and the **multipleTransactions** argument set to false.

The options you can specify are:

Option	***Description***
-skipobjects	Specifies that the database contains no objects that depend on **String.hashCode()**.
-multitrans	Specifies that each object should be upgraded in a separate transaction.
-verbose	Specifies that verbose information about the upgrade should be displayed. This includes displaying information about databases that do not require upgrading.

osjversion: Obtaining ObjectStore Version Information

You can use the **osjversion** utility to display the version number and the build date for the version of ObjectStore you are running. This command is in the **bin** subdirectory of the installation directory. You must include **bin** in your path to run **osjversion**. For example:

% **osjversion**

Java interface Release 3.0 to ObjectStore Release 5.1, 98-09-22 10:07:16 on buildhost

%

This command is useful when you want to ensure that you have the right version in your path.

You can also run the **unzip -z** command against any ObjectStore **.zip** file. Doing so allows you to see what version of ObjectStore the **.zip** file contains. For example:

% **unzip -z osji.zip**

Archive: osji.zip Java interface 3.0 to ObjectStore Release 5.1. 98-04-22 10:07:56 on buildhost,

%

The **osjversion** utility checks what is available through the **PATH** environment variable. To check what is available through the **CLASSPATH** variable, you can write a program like the following.

```
import COM.odi.ObjectStore;
class OSVersion {
    public static void main (String[] args) {
        ObjectStore.initialize(null,null);
        System.out.println(ObjectStore.releaseName());
    }
}
```

Appendix Packaging Your Application for End Users

When you package your application for delivery to end users, the package must include two class files for each persistence-capable class in your application:

- The annotated class file
- The corresponding **ClassInfo** class file

For example, if you have a persistence-capable class called **Person**, in the **App** package, you must provide users with

- The annotated **App\Person.class** file
- The **App\PersonClassInfo.class** file

There is no corresponding **ClassInfo** file for persistence-capable interfaces.

You can zip these files with the rest of your package. You do not need to send the unannotated version of your persistence-capable classes. The annotated version can be used in a transient context.

For persistence-aware classes, you must provide the annotated class file. There is no corresponding **ClassInfo** class file.

Library providers

If you are providing a library that contains persistence-capable classes, you must also provide the **PersistentTypeSummary** object that identifies those classes. ObjectStore uses the summary object to install schema information for your library at run time. This allows users to install a new version of a library without having to rebuild the application type summary.

Glossary

active persistent object

An active persistent object starts as an exact copy of the object that it represents in the database. ObjectStore initializes a hollow object so that it becomes an active object. This happens when an application calls the **ObjectStore.fetch()** or **ObjectStore.dirty()** method. If an application calls the **ObjectStore.dirty()** method (as opposed to the **ObjectStore.fetch()** method) on a hollow object, the resulting active object can be modified.

Consequently, an active object is not necessarily identical to the object in the database that it represents. An application can read or update an active persistent object; a persistent object must be active for an application to read or update it.

batch

A batch is a set of files that must be postprocessed together. Often, this is all the files in your application. In more complex applications, there might be multiple batches that each contain a library and a batch of files that you write, which reference the libraries.

database

Persistent storage is organized into databases. Before a persistent object can be created, the database in which it is to be stored must exist, and this database must be opened by the process performing the creation. The database must also be opened by any processes accessing the object. A single application can open several databases at once. A single database can be accessed by many applications at once.

deadlock

A simple deadlock occurs when one transaction holds a lock on a data item that another transaction is waiting to access, while at the same time the second transaction holds a lock on a data item that the first transaction is waiting to access. Neither process can proceed until the other does. ObjectStore detects and breaks deadlocks by aborting one of the transactions involved.

hollow persistent object

A hollow persistent object contains fields that are identical to the fields of the object in the database that the persistent object represents, but the fields have default values.

persistence-aware

If the methods of a class can operate on persistent objects but an instance of the class cannot itself be stored in a database, the class is persistence-aware.

When a method accesses fields in a persistent object, ObjectStore checks to ensure that the data has been read from the database. This checking is done by calls to the **ObjectStore.fetch()** and**ObjectStore.dirty()** methods. A persistence-aware class includes the annotations that call the **fetch()** and **dirty()** methods. It does not include the other annotations required for a class to be persistence-capable. Normally, you run the class file postprocessor to annotate a class so that it is persistence-aware. Occasionally, you manually annotate the class yourself.

persistence-capable

A persistence-capable object has the capacity to be stored in a database. If you can store the instances of a class in a database, the class is persistence-capable and the instances of the class are persistence-capable objects.

The definition of a persistence-capable class includes specific annotations required by ObjectStore. After you compile class definitions, you run the ObjectStore class file postprocessor on the compiled classes to add the annotations that make the classes persistence-capable.

Some Java-supplied classes are persistence-capable. Others are not persistence-capable and cannot be made persistence-capable. A third category of classes can be made persistence-capable, but there are important issues to consider when you do so. Be sure to read Java-Supplied Persistence-Capable Classes on page 360.

persistent object

A persistent object is a representation of an object that is stored in a database.

After an application retrieves an object from the database, the application works with the persistent object in the Java environment. A persistent object always exists in one of three states:

- Hollow
- Active
- Stale

session

A session allows the use of the ObjectStore API. ObjectStore uses the abstract **COM.odi.Session** class to represent sessions.

Your application must create a session. After a session is created, it is an active session. A session remains active until your application or ObjectStore terminates it. After a session is terminated, it is never used again. You can, however, create a new session.

A session consists of a set of persistent objects, and a set of ObjectStore API objects, such as a **Transaction**, **Database**s, and **Segment**s.

In a single Java VM,

- PSE allows one session at a time.
- PSE Pro allows multiple concurrent sessions.
- ObjectStore allows one session at a time. In a future release, ObjectStore will allow multiple sessions.

If you are using PSE or ObjectStore, separate Java virtual machines can each run their own session at the same time. If you are using PSE Pro, separate Java virtual machines can each run multiple sessions at the same time. See How Sessions Keep Threads Organized on page 28.

stale persistent object

A stale persistent object is no longer valid. Its fields have default values and it should not be used.

A persistent object might become stale after an application commits or aborts a transaction in which the active or hollow persistent object was accessible. When an application calls the **ObjectStore.destroy()** method, the target of the destroy method becomes stale.

An application must not try to read or update a stale object.

transitive persistence

When an application commits a transaction, it stores in the database any transient objects that can be transitively reached from any persistent objects. This is the process of transitive persistence.

Index

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W