:

Java Interface to ObjectStore Tutorial

Release 6.1
February 2003

Java Interface to ObjectStore Tutorial

ObjectStore Release 6.1 for all platforms, February 2003

Contents

Preface

Purpose

Java Interface to ObjectStore Tutorial provides an overview of the basic concepts of Java interface to ObjectStore. It uses an example application to show you the way to define persistent classes and manipulate persistent objects. It also shows you the way to develop and run applications that use the Java interface to ObjectStore.

Audience

This book is for experienced Java programmers who are new to writing applications that use the Java interface to ObjectStore. If you are new to Java interface to ObjectStore, you should read the tutorial, and then look at the demonstration programs.

Scope

This book supports Release 6.1 of the Java interface to ObjectStore.

How the Tutorial Is Organized

The tutorial provides an introduction to ObjectStore interface for Java. The basic concepts are illustrated with a sample application called the Personalization application. In Chapter 5, you can find information that goes beyond the sample application.

- Chapter 1, Benefits of ObjectStore for Java, on page 1, introduces the concept of persistence and compares object serialization to the features provided by ObjectStore.
- Chapter 2, Description of the Personalization Application, on page 5, presents a small sample application to show the way to use ObjectStore.
- Chapter 3, Writing Your Application to Use ObjectStore, on page 13, introduces the core concepts of ObjectStore through explanation of code samples drawn from the Personalization application.
- Chapter 4, Compiling and Running an ObjectStore Program, on page 29, describes the compile and build phases of ObjectStore development using code from the Personalization application.

- Chapter 5, Using ObjectStore to Query a Database, on page 37, discusses queries and indexing, which are available in ObjectStore and PSE Pro. The tutorial application does not implement queries, but the information in this chapter is a basis for getting started.
- Chapter 6, Choosing PSE Pro or ObjectStore, on page 43, provides information to help you decide whether PSE Pro, or ObjectStore is the most appropriate solution for your requirements.
- Appendix A, Source Code, on page 47, provides a complete code example for the Personalization application.
- Appendix B, Sample Output, on page 63, provides sample output from running the Personalization application.

Notation Conventions

This document uses the following conventions:

Convention	*Meaning*
`Courier`	`Courier font` indicates code, syntax, file names, API names, system output, and the like.
`Bold Courier`	**`Bold Courier font`** is used to emphasize particular code, such as user input.
`Italic Courier`	*`Italic Courier font`* indicates the name of an argument or variable for which you must supply a value.
Sans serif	Sans serif typeface indicates the names of user interface elements such as dialog boxes, buttons, and fields.
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

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)`.

ObjectStore on the World Wide Web

ObjectStore has its own Web site (www.objectstore.net) that provides a variety of useful information about products, news and events, special programs, support, and training opportunities.

Technical Support

When you purchase technical support, the following services are available to you:

- You can send questions to support@objectstore.net. Remember to include your site ID in the body of the electronic mail message.
- You can call the Technical Support organization to get help resolving problems.
- You can access the Technical Support Web site, which includes
  - A template for submitting a support request. This helps you provide the necessary details, which speeds response time.
  - Frequently asked questions (FAQs) that you can browse and query.
  - Online documentation for all ObjectStore products.
  - White papers and short articles about using ObjectStore products.
  - Sample code and examples.
  - The latest versions of ObjectStore products, service packs, and publicly available patches that you can download.
  - Access to an ObjectStore product matrix.
  - Support policies.
  - Local phone numbers and hours when support personnel can be reached.

Education Services

Use the ObjectStore education services site (www.objectstore.net/services/education) to learn about the standard course offerings and custom workshops.

If you are in North America, you can call 1-800-477-6473 x4452 to register for classes. For information on current course offerings or pricing, send e-mail to classes@progress.com.

Searchable Documents

In addition to the online documentation that is included with your software distribution, the full set of product documentation is available on the ObjectStore Support Web server. The documentation is found at www.objectstore.net/documentation, and is listed by product. The site supports the most recent release and the previous supported release of ObjectStore documentation. Service Pack README files are also included to provide historical context for specific issues. Be sure to check this site for new information or documentation clarifications posted between releases.

Your Comments

ObjectStore product development welcomes your comments about its documentation. Send any product feedback to support@objectstore.net. To expedite your documentation feedback, begin the subject with `Doc:`. For example:

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

Chapter 1 Benefits of ObjectStore for Java

To introduce you to ObjectStore and the benefits of using ObjectStore, this chapter discusses the following topics:

Overview of ObjectStore Benefits

ObjectStore combines the simplicity of object serialization, and the reliability of a database management system (DBMS), together with in-memory-like performance for accessing persistent objects.

Although object serialization is easy to use, you quickly run into performance problems with large numbers of objects. Relational database management systems provide robust and reliable data storage, but mapping Java objects into rows and columns slows performance and increases the amount of code to be written.

ObjectStore improves upon some of the limitations of object serialization and relational databases, while providing performance that is better by orders of magnitude and still maintaining an easy-to-use interface.

Serialization and Persistence

The lifetime of a persistent object exceeds the life of the application process that created the persistent object. Persistent object management is the mechanism for storing the state of objects in a nonvolatile place so that when the application shuts down, the objects continue to exist.

Description of Serialization

Java provides object serialization, which supports a basic form of object persistence. You can use object serialization to store copies of objects in a file or to ship copies of objects to an application running in another Java virtual machine. Serialization enables you to flatten objects into a stream of bytes that, when read later, can recreate objects equivalent to those that were written to the stream.

Serialization provides a simple yet extensible mechanism for storing objects persistently. The Java object type and safety properties are maintained in the serialized form and serialization requires only per-class implementation for special customization. Serialization is usually sufficient for applications that operate on small amounts of persistent data and for which reliable storage is not an absolute requirement.

Disadvantages of Serialization

Serialization is not the optimal choice for applications that

- Manage tens to hundreds of megabytes of persistent objects
- Update objects frequently
- Want to ensure that changes are reliably saved in persistent storage

Because serialization has to read and write entire graphs of objects at a time, it works best for small numbers of objects. When the byte stream is a couple of megabytes in size, you might find that storing objects by means of serialization is too slow, especially if your application is doing frequent updates that need to be saved. Another drawback is the lack of undo or abort of changes to objects.

In addition, serialization does not provide reliable object storage. If your system or application crashes when objects are being written to disk by serialization, the contents of the file are lost. To protect against application or system failures and to ensure that persistent objects are not destroyed, you must copy the persistent file before each change is saved.

The Way ObjectStore Improves on Serialization

The three key improvements that ObjectStore provides over serialization are

- Improved performance for accessing large numbers of objects
- Reliable object management
- Queries

In addition, ObjectStore is easy to use, allows access to as little as a single object at a time, and allows multiple applications to read the same database at the same time.

Improved Performance for Accessing Large Numbers of Objects

While serialization reads and writes complete graphs of objects, ObjectStore provides the capability to read and write a few objects at a time. ObjectStore also provides the capability to access or fetch a smaller subset of objects from a larger number of objects. ObjectStore fetches related objects automatically when the application code refers to them.

Reliable Object Management

A primary difference between serialization and ObjectStore is in the area of transactions and recovery. With serialization, persistent stores are not recoverable automatically. Consequently, in the event of an application or system failure, a file can be recovered only to the beginning of the application session and only if a copy of the file is made before the application begins.

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

Queries

ObjectStore provides a mechanism for querying collections of objects. A query returns a subset of objects for which the query expression is true. To improve the performance of a query on a particularly large collection, you can build indexes on the collection. For more information on queries,see Chapter 5, Using ObjectStore to Query a Database, on page 37.

Ease of Use

As with serialization, ObjectStore provides an easy-to-use interface for storing and retrieving Java objects. You define persistence-capable Java classes and their fields and methods in the same way that you define transient Java classes.

You use standard Java constructs to create and manipulate both persistent and transient instances. Transparent object persistence through ObjectStore enables developers to make use of the full power of Java and to easily incorporate existing class libraries with ObjectStore.

The ObjectStore for Java API provides database features that allow you to

- Create, open, and close databases
- Start and end transactions
- Store and retrieve persistent objects

ObjectStore automatically generates the equivalent of serialization's `readObject` and `writeObject` for each persistence-capable class. As with serialization, you can override the implementation of these methods.

Chapter 2 Description of the Personalization Application

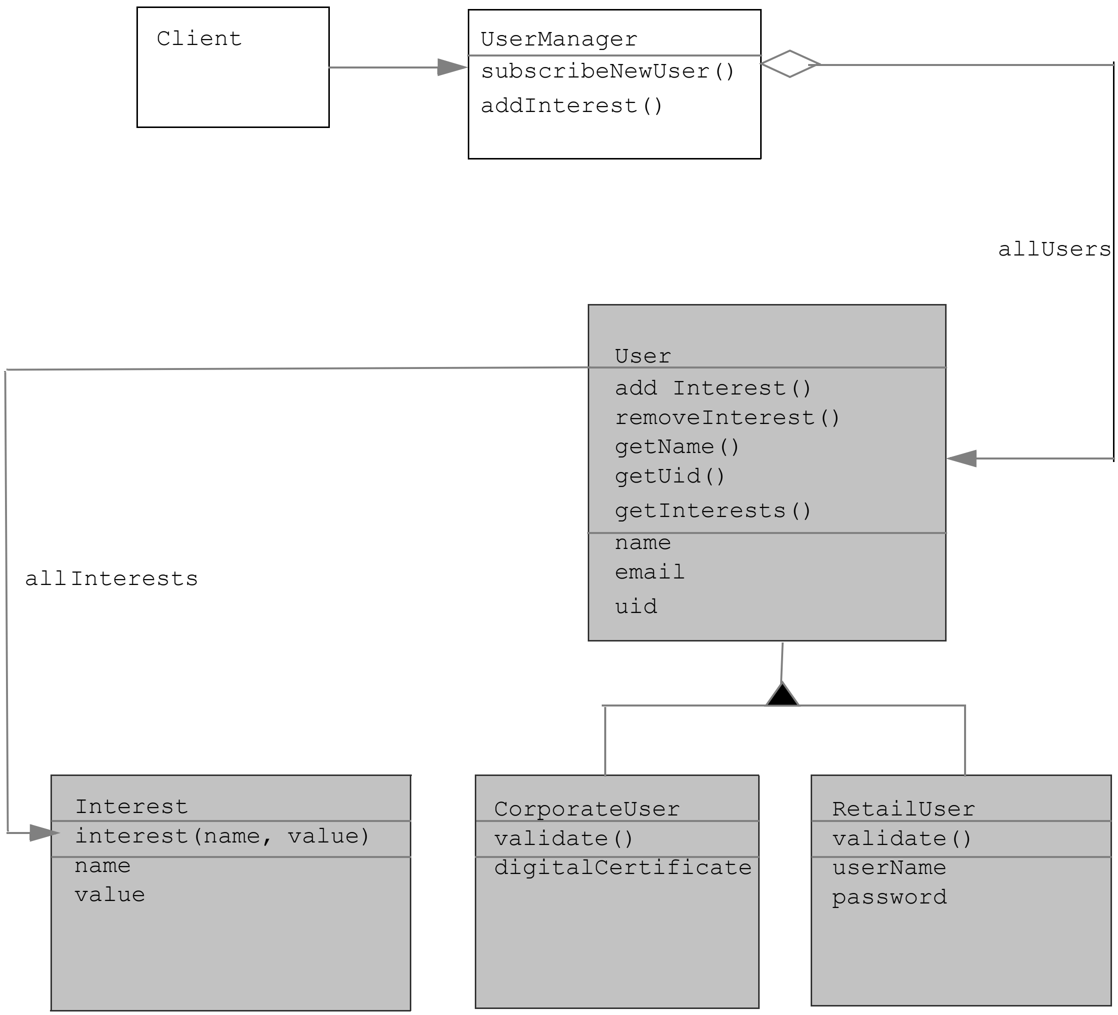
This tutorial refers to the *Personalization* application to help illustrate the key principles of ObjectStore. The following topics describe the Personalization application:

Overview of the Data Model

Consider the task of personalizing a web site to cater to an individual user's interests and access patterns. A personalized web site delivers customized content to the browser user. To achieve customization, a database of users and their interests must be maintained in the web architecture.

The personalization database tracks user interests so that web content can be generated dynamically based on those interests. Because web sites change rapidly, it must be easy to define new interests to the database. Each user's set of interests is different, and users can change their interests at any time.

The following figure shows the personalization object model with Rumbaugh Object Modeling Technique (OMT) notation. White classes are transient; that is, they are internal to the application's memory. Shaded classes are persistence capable; that is, instances of them can be stored in an ObjectStore database.



Personalization Application Architecture

The source code for the Personalization application is included with ObjectStore. By default, the files are installed in the `com\odi\tutorial` directory. The code is also listed in Appendix A, Source Code, on page 47.

Five files make up the Personalization application:

File	*Description*
`Interest.java`	Code for `Interest` objects
`User.java`	Code for `User` objects
`UserManager.java`	Code for the `UserManager` class which controls access to the objects in the database
`TestDriver.java`	Code for the user interface.
`PersonalizationException.java`	Code for handling exceptions specific to the application.

Description of the UserManager Class

The client application interfaces with the `UserManager` class, which controls access to `User` and `Interest` objects in the database. The `UserManager` also controls user access, registration, and session management. The `UserManager` might run as an application server that operates behind a web server and provides access to the personalization database.

The `UserManager` class has a number of static members that keep track of the database that is open, the set of registered users, and the set of interests defined to the database. It also has a number of static methods, each of which executes a transaction in ObjectStore.

The tutorial example reads screen input to generate requests, but requests can easily bypass the client application using the Remote Method Invocation (RMI), an Object Request Broker (ORB), or as Common Gateway Interface (CGI) environment variables if you are using Hypertext Transport Protocol (HTTP).

The `UserManager` class contains most of the database-specific code, such as starting and ending transactions. There are no `UserManager` objects stored in the database, which means that the `UserManager` class is not required to be persistence capable. Complete source code for this class is listed in Appendix A in Source Code for UserManager.java on page 51.

Description of the User and Interest Classes

There are two persistence-capable classes in the Personalization application: the `User` class and the `Interest` class. Only instances of persistence-capable classes can be stored in a database. Chapter 4, Compiling and Running an ObjectStore Program, on page 29, describes the way you use the postprocessor to make classes persistence capable.

`User` objects contain core information about a user, for example, name, email address, and personal identification number (PIN). `User` objects are an acquaintance of (that is, they are associated with, but do not own) `Interest` objects. `Interest` objects contain descriptions of the types of content the associated user is interested in. The `Interest`'s name (for example, `Food`) and value (`Pizza`) define the interest of the user. Users can change their profiles dynamically at run time by adding and removing `Interest` objects.

You define the `User` class for persistent use in the same way that you define it for transient use. The `Interest` class is also defined like any other Java class. Other than the `import com.odi.*` statement, there is almost no special code for persistent use of the `User` or `Interest` class. This is one of the key advantages of the transparent Java language binding that ObjectStoreprovides.

Adding New Interests

You can create new interests at run time because of the simple *metadata* aspect of the `Interest` class. That is, an `Interest` object has a name and a value. The name (for example, `Food`) and value (`Pizza`) can be defined at run time. New classes are not required to add or change interests.

Adding New User Types

As you define new types of users, you could add them as a specialized type of `User`. For example, `CorporateUser` and `RetailUser` might be specializations of the generic `User` class. These specialized user types might overload and perform various functions. For example, different user classes could apply different heuristics for authentication. For simplicity, the tutorial example implements only the generic `User` class.

Source Code for the Interest Class

Following is a portion of the source file for the `Interest` class. Complete source code is in Appendix A, in Source Code for Interest.java on page 47.

```
package com.odi.tutorial;
import com.odi.*;

/**
 * The Interest class models a particular interest that a user
 * might have. It contains a name and a value. For example, the
 * name of an interest might be "food" and the value might be
 * "pizza".
 */

public class Interest
{
  /* the name of the interest */
  private String name;
  /* the value of the interest */
  private String value;
  /* the user who has this interest */
  private User user;
  /* accessor functions */
  public String getName() { return name; }
  public String getValue() { return value; }
  public void setValue(String value) { this.value = value; }
  public User getUser() { return user; }

  /**
   * Constructs a new interest object given a name and a value
   */

  public Interest(String name, String value, User user)
  {
    this.name = name;
    this.value = value;
    this.user = user;
  }
}
```

Source Code for the User Class

Following is a portion of the source code for the `User` class. Complete source code is in Appendix A, Source Code for User.java on page 48.

```
package com.odi.tutorial;

import java.util.*;
import com.odi.*;
import com.odi.util.*;

/**
```

```
 * The User class models users. Users have names, email addresses,
 * and PINS. Each user also has a set of interests. The application
 * uses PINs to validate a user's identity.
 */
public
class User {

  /* The name of the user */
  private String name;

  /* The user's email address */
  private String email;

  /* The user's Personal Identification Number */
  private int PIN;

  /* The set of user's interests */
  private OSHashMap interests;

  /* accessors */
  public String getName() { return name; }
  public String getEmail() { return email; }
  public int getPIN() { return PIN; }
  public Map getInterests() { return interests; }

  /**
   * Constructs a user object given the name, email and PIN
   */
  public User(String name, String email, int PIN) {
    this.name = name;
    this.email = email;
    this.PIN = PIN;
    this.interests = new OSHashMap(5); /* initial hash table size */
  }

  /**
   * Add an interest to the User's list of interests.
   *
   * @param interestName the name of the interest
   * @param interestValue the value of the interest
   *
   * @exception PersonalizationException If the interest is
   * already there (the same name as another interest)
   */
  public Interest addInterest(String interestName, String interestValue)
       throws PersonalizationException {
    // Implementation ...
  }

  /**
   * Update an interest in the User's list of interests.
   *
```

```
 * @param interestName the name of the interest
 * @param interestValue the new value of the interest
 *
 * @exception PersonalizationException is thrown if the interest is
 * already not there.
 */
public Interest changeInterest(String interestName, String interestValue)
     throws PersonalizationException{
  // Implementation ...
}

/**
 * Remove an interest from the User's list of interests.
 *
 * @param interestName The name of the Interest to remove.
 *
 * @exception PersonalizationException if the interest is not
 * found in the user's list of interests
 */
public Interest removeInterest(String interestName)
     throws PersonalizationException {
  // Implementation ...
}
```

Chapter 3 Writing Your Application to Use ObjectStore

This chapter discusses the core concepts involved in writing an ObjectStore application. It uses the Personalization application to provide examples of the concepts. This chapter discusses the following topics:

Basic ObjectStore Operations

To store and access persistent data, all ObjectStore applications must implement the following basic operations:

- Create and join a session. See Creating Sessions on page 14.
- Create or open a database. See Creating, Opening, and Closing Databases on page 15.
- Start a transaction. See Starting Transactions on page 16.
- Create or retrieve a database root. See Creating Database Roots on page 18.

In addition, you must perform the following before running your application for the first time:

- Modify (or *annotate*) classes that will be stored in a database so they are *persistence capable* (or *persistence aware* for classes that need access to persistent objects). You do this by running the `oscfp` postprocessor after compiling your source code. See Running the Postprocessor on page 32.
- Set or modify the `CLASSPATH` environment variable to point to the annotated class files. See Adding Entries to Your CLASSPATH on page 30.

Getting Ready to Store Objects

Before you can create and manipulate persistent objects with ObjectStore, you must perform the following operations:

- Create a session.
- Create or open a database.
- Start a transaction.

In the Personalization application, all these operations are handled by methods of the `UserManager` class.

Creating Sessions

To use ObjectStore, your application must create a session. A session is the context in which ObjectStore databases are created or opened and transactions can be executed. Only one transaction at a time can exist in a session. Both ObjectStore and PSE Pro allow you to create multiple sessions and thus have multiple concurrent transactions in a single Java VM process.

Any number of Java threads can participate in the same session. Each thread must join a session to be able to access and manipulate persistent objects. To create a session, you call the `Session` constructor and specify the host and properties. The method signature is

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

A thread can join a session with a call to `Session.join()`. For example:

```
/* Create a session and join this thread to the new session. */
session = Session.create(null, null);
```

```
session.join();
```

ObjectStore ignores the first parameter in the `create()` method. You can specify null. The second parameter specifies null or a property list. See the *Java API User Guide*, Description of Properties.

Creating, Opening, and Closing Databases

Before you begin creating persistent objects, you must create a database to hold the objects. In subsequent processes, you open the database to allow the process to read or modify the objects. To create a database, you 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)
```

The `initialize` method in the `UserManager` class shows an example.

```
public static void initialize(String dbName)
{

/* Other code, including creating a session and joining thread to session*/

/* Open the database or create a new one if necessary. */
  try {
    db = Database.open(dbName, ObjectStore.UPDATE);
  } catch (DatabaseNotFoundException e) {
    db=Database.create(dbName, ObjectStore.ALL_READ | ObjectStore.ALL_WRITE);
  }
}
```

The `initialize()` operation first creates a session, then joins the current thread to that session. Next, `initialize()` tries to open the database. If the database does not exist, `DatabaseNotFoundException` is thrown and is caught by `initialize()`, which then creates the database. `initialize()` also stores a reference to the database instance in the static variable `db`.

The `Database.create()` and the `Database.open()` methods are called with two parameters. In both methods, the first parameter specifies the pathname of a file. In the `create()` method, the second parameter is a UNIX-style protection number. In the `open()` method, the second parameter specifies the access mode for the database, that is, `ObjectStore.UPDATE` or `ObjectStore.READONLY`.

Shutting down

The `UserManager.shutdown()` method shows an example of closing a database and terminating a session.

```
/**
* Close the database and terminate the session.
*/
```

```
public static void shutdown() {
  db.close();
  session.terminate();
}
```

Starting Transactions

You create, destroy, open, and close a database outside a transaction. You access and manipulate objects in a database inside a transaction. Therefore, a program must start a transaction before it can manipulate persistent data. While the transaction is in progress, a program can read and update objects stored in the open database. The program can choose to commit or abort the transaction at any time.

Committing transactions

When a program commits a transaction, ObjectStore updates the database to contain the changes made to persistent data during the transaction. These changes are permanent and visible only after the transaction commits. If a transaction aborts, ObjectStore undoes (rolls back) any changes to persistent data made during that transaction.

Purpose of transactions

In summary, transactions do two things:

- They mark off code sections whose effects can be undone.
- They mark off functional program areas that are isolated from the changes performed by other sessions or processes (clients). From the point of view of other sessions or processes, these functional sections execute either all at once or not at all. That is, other sessions or processes do not see the intermediate results.

Creating transactions

To create a transaction, insert calls to mark the beginning and end of the transaction. 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

```
public static Transaction begin(int type)
```

The type of the transaction can be `ObjectStore.READONLY` or `ObjectStore.UPDATE`. Other transaction types are discussed in *Java API User Guide*, Description of Transaction Types.

Ending transactions

ObjectStore provides the `Transaction.commit()` method for ending a transaction successfully. When transactions terminate successfully, they commit and their changes to persistent objects are saved in the database. The `Transaction.abort()` method is used to end a transaction unsuccessfully. When transactions terminate unsuccessfully, they abort and their changes to persistent objects are discarded.

When an application commits a transaction, ObjectStore saves and commits any changes in the database. It also checks to see whether any transient objects are referred to by persistent objects. If any are, and if the referred-to objects are persistence-capable objects, ObjectStore stores the referred-to objects in the database. This is the process of transitive persistence, also called *persistence by reachability*.

The default commit operation makes all persistent objects inaccessible outside the transaction's context. After you commit a transaction, if you want to access data in the database, you must start another transaction and navigate to the object again from a database entry point.

ObjectStore also provides optional commit modes (or *states*) that allow you to retain the objects so that you can access them outside a transaction or in a different transaction.

These commit modes include

- `ObjectStore.RETAIN_HOLLOW`
- `ObjectStore.RETAIN_READONLY`
- `ObjectStore.RETAIN_UPDATE`
- `ObjectStore.RETAIN_STALE` (the default)
- `ObjectStore.RETAIN_TRANSIENT`

The Personalization application uses the `ObjectStore.RETAIN_HOLLOW` state when it commits transactions so that the application does not need to retrieve the database root and navigate to the object in subsequent transactions. For more information about retain states, see Retaining Objects or References to Objects22

Creating Database Entry Points

To access objects in a database, you need a mechanism for referring to these objects. In other words, you need an entry point. In a relational database system, the entry points are the tables defined to the database. The tables have names that you can use in queries to gain access to the rows of data. You cannot directly access a row by its table name.

In ObjectStore, the names or entry points are called *roots* and they are more flexible than in the relational database model.

Description of Database Roots

You can use a database root to name any object defined in the database. You can use a root to reference a collection object, which is ObjectStore's equivalent of a table; this is what the Personalization application does. You can, however, also choose to assign roots to individual objects.

A database root provides a way to give an object a persistent name. A root allows an object to serve as an initial entry point into persistent storage. When an object has a persistent name, any process can look it up by that name to retrieve it. After you retrieve one object, you can retrieve any object related to it by navigating object references or by a query.

Each database typically has a relatively small number of entry point objects, each of which allows access to a large network or collection of related objects. The Personalization application uses two roots: one for `User` objects and one for `Interest` objects.

Creating Database Roots

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

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

The name you specify for the root must be unique in the database. The object that you specify to be referred to by the root can be transient and persistence capable, persistent, or null. If it is not yet persistent, ObjectStore makes it persistent automatically when you call `createRoot()`.

Example of Creating Database Roots

In the remainder of the `UserManager.intialize()` operation, the Personalization application begins a transaction and looks for the database roots `allUsers` and `allInterests`. If they are not there, the application creates them, and then commits the transaction.

```
public static void initialize(String dbName)

//  database open code omitted

  /* Find the allUsers and allInterests roots or create them if not there. */
  Transaction tr = Transaction.begin(ObjectStore.UPDATE);
  try {
    allUsers = (Map) db.getRoot("allUsers");
```

```
    allInterests = (Set) db.getRoot("allInterests");
  } catch (DatabaseRootNotFoundException e) {
    /* Create the database roots and give them appropriate values */
    db.createRoot("allUsers", allUsers = new OSHashMap());
    db.createRoot("allInterests", allInterests = new OSHashSet());
  }
  /* End the transaction and retain a handle to allUsers and allInterests */
  tr.commit(ObjectStore.RETAIN_HOLLOW);
```

Most of the methods defined on `UserManager` access the root objects and use them to find a particular user, add a new user, or remove a user. The next section discusses these operations.

The Personalization application keeps track of all the users who register with the site, as well as the interests of each registered user. To track users, the application uses a persistent `Map` that is indexed on the user names. This allows quick look-up of a user in the database. To track interests, the application uses a `Set` of interests. These `Map`s and `Set`s are implementations of the JDK 1.2 collections. For more information, see Using Collections on page 26.

Storing Objects in a Database

Objects become persistent when they are referenced by other persistent objects. The Personalization application defines persistent roots; therefore, when a transaction commits, ObjectStore finds all objects reachable from these persistent roots and stores them in the database. This type of persistence is called *persistence by reachability* and helps preserve the automatic storage management semantics of Java.

Example of Storing Objects in a Database

For example, in the Personalization application consider the `UserManager.subscribe()` method, which adds a new user to the database.

```
public static int subscribe(String name, String email)
    throws PersonalizationException
  {
    Transaction tr = Transaction.begin(ObjectStore.UPDATE);
    /* First check to see if the user's name is already there. */
```

```
    if (allUsers.get(name) != null) {
      tr.abort(ObjectStore.RETAIN_HOLLOW);
      throw new PersonalizationException("User already there: " + name);
    }
    /* The user name is not there so add the new user;
       first generate a PIN in the range 0..10000. */
    int pin = pinGenerator.nextInt() % 10000;
    if (pin < 0) pin = pin * -1;
    User newUser = new User(name, email, pin);
    allUsers.put(name, newUser);
    tr.commit(ObjectStore.RETAIN_HOLLOW);
    return pin;
  }
```

The Personalization application checks whether the user's name is already defined in the database. If the name is defined, the Personalization application throws a `PersonalizationException`. If the name is not already defined, then the Personalization application creates a new user, adds that user to the `allUsers` collection, and commits the transaction. Because the `allUsers` collection is already stored in the database, ObjectStore stores the new user object in the database when it commits the transaction.

In the Personalization application, another example of storing objects in a database is the `addInterest()` method defined on the `User` class. To add an interest to a user's set of interests, the application calls

```
interests.put(interestName, interest);
```

This adds an `Interest` object to the user's interests, which are stored in a `Map`. When the transaction commits, ObjectStore makes the new `Interest` object persistent because the user is a persistent object and its `Map` is persistent. See Appendix A, Source Code, on page 47 for the complete code.

Definition of Persistence Capable

An object must be persistence capable before an application can store that object in an ObjectStoredatabase. Persistence capability is the capacity 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.

(ObjectStore also allows for classes that are persistence aware. Persistence-aware classes can access and manipulate instances of persistence-capable classes but cannot themselves be stored in a database. See the *Java API User Guide*, Creating Persistence-Aware Classes.)

To make a class persistence capable, you compile the class definitions as usual, then run the ObjectStore class file postprocessor on the class files. The class file postprocessor annotates the classes you define to make them persistence capable. This means that the postprocessor makes a copy of your class files, places them in a directory you specify, and adds lines of code (annotations) that are needed for persistence. Details about the way to run the postprocessor are in Chapter 4, Compiling and Running an ObjectStore Program, on page 29.

The annotations added by the postprocessor allow ObjectStore to understand the state of objects so ObjectStore can save their state in persistent storage. The annotations also allow ObjectStore to automatically ensure that

- Fields are always fetched before being accessed.
- Modified instances are always updated in the database at commit time.

Accessing Objects in the Database

After an application stores objects in a database, the application can use references to these objects in the same way that it uses references to transient objects. An application obtains initial access to objects in a database by navigating from a root or through an associative query. An application can retain references to persistent objects between transactions to avoid having to obtain a root at the start of each transaction.

Example of Using a Database Root

To access objects in a database, you must start a session, open the database, and start a transaction. Then you can obtain a database root to start navigating the database. For example, in the Personalization application (in `UserManager.java`), you obtain the `"allUsers"` root to obtain `User` objects.

```
allUsers = (Map) db.getRoot("allUsers");
```

Example of Using References

Consider again the `subscribe()` method in the Personalization application. The first part of this method protects against storing a duplicate name by checking whether the user's name is already in the database. For example:

```
/* First check to see if the user's name is already there. */
  if (allUsers.get(name) != null) {
```

```
      tr.abort(ObjectStore.RETAIN_HOLLOW);
      throw new PersonalizationException("User already there: " + name);
    }
```

Because the class variable `allUsers` references the `allUsers` collection, the application can use the standard Java `Map.get()` method to check if the name is already stored. The same code works for a persistent or a transient collection.

Retaining Objects or References to Objects

Each time the Personalization application commits a transaction, it specifies the `ObjectStore.RETAIN_HOLLOW` option. This option keeps references that were available during the transaction. The application can use the references in subsequent transactions.

After the Personalization application commits the initial transaction, the class variable `allUsers` continues to reference the `allUsers` collection. When it begins a new transaction, the application need not reestablish a reference to `allUsers` with the `getRoot()` method.

When an application commits a transaction, the default retain option is `ObjectStore.RETAIN_STALE`. This option makes all persistent objects inaccessible outside a transaction. To access any objects in the database, you must start a transaction, use a root to access an initial object, and navigate to other objects from the root.

For example, if the Personalization application specified `ObjectStore.RETAIN_STALE` when it committed a transaction, it could not access the `allUsers` collection outside a transaction. Also, to access the `allUsers` collection again, the application would need to start a new transaction and obtain a new reference with a call to the `getRoot()` method to obtain the `allUsers` `Map`.

If you want to access the contents of persistent objects outside a transaction, you can specify the `ObjectStore.RETAIN_READONLY` or `ObjectStore.RETAIN_UPDATE` option. These options allow you to read or update objects whose contents were available during the transaction. For example, the Personalization application specifies the `ObjectStore.RETAIN_READONLY` option in `validateUser()`.

```
public static User validateUser(String userName, int PIN)
  {
    Transaction tr = Transaction.begin(ObjectStore.READONLY);
    User user = (User) allUsers.get(userName);
```

```
    if (user == null) {
      tr.abort(ObjectStore.RETAIN_HOLLOW);
      throw new PersonalizationException ("Could not find user: " + userName );
    }
    if (user.getPIN() != PIN) {
      tr.abort(ObjectStore.RETAIN_HOLLOW);
      throw new PersonalizationException ("Invalid PIN for user: " + userName );
    }
    tr.commit(ObjectStore.RETAIN_READONLY);
    return user;
  }
```

If the `userName` and `PIN` passed to the `validateUser()` method denote a registered user, the `validateUser()` method returns a `User` object. Because `User` objects are persistent objects, if you want their contents to be accessible outside the transaction in which they were fetched from the database, you must specify the `ObjectStore.RETAIN_READONLY` or `ObjectStore.RETAIN_UPDATE` option when you commit the transaction.

If you use the default `ObjectStore.RETAIN_STALE` option, the receiver gets a stale `User` object. This causes ObjectStore to throw an exception when the application tries to access the `User` object. If you specify the `ObjectStore.RETAIN_HOLLOW` option, the `validateUser()` method returns a reference to a `User` object but not the contents of the `User` object. That is, no name, eMail, or PIN information is available. You can use the returned reference in a subsequent transaction.

Deleting Objects

When you delete objects in ObjectStore, you must

- Disconnect objects from their relationships and associations
- Destroy the object so that it is removed from the database

Example of Deleting an Object

To remove users from the database, for example, the Personalization application calls the `UserManager.unSubscribe()` method.

```
  public static void unsubscribe(String name)
    throws PersonalizationException
  {
    Transaction tr = Transaction.begin(ObjectStore.UPDATE);

    User user = (User) allUsers.get(name);
```

```
    if (user == null) {
      tr.abort(ObjectStore.RETAIN_HOLLOW);
      throw new PersonalizationException ("Could not find user: " + name);
    }

    /* remove the user from the allUsers collection, and
     * remove all of the users interests from the allInterests collection */
    allUsers.remove(name);
    Iterator interests = user.getInterests().values().iterator();
    while (interests.hasNext())
      allInterests.remove(interests.next());

    /* finally destroy the user and all its subobjects */
    ObjectStore.destroy(user);

    tr.commit(ObjectStore.RETAIN_HOLLOW);
  }
```

First, the Personalization application ensures that the user exists in the `allUsers` collection. If the user does not exist, the application throws an exception. Next, the application calls the `remove()` method to remove the user from the `allUsers` collection. This disconnects the user from the set of users, which means that this user is no longer reachable and can now be removed from the database. However, because there are still interests associated with the user, the application removes all the user's interests from the `allInterests` collection. Finally, to remove the user from the database, the application calls `destroy()` on the `User` object.

Destroying an Object

The `destroy()` method is an operation defined on the `ObjectStore` class. The signature is

public static void destroy(Object *object*)

The object you specify must be persistent or the `destroy()` method has no effect.

By default, when you destroy an object, ObjectStore does not destroy objects that the destroyed object references. In the Personalization application, the `User` object references two strings — `name` and `email` — as well as a map of `Interest`s. To destroy these objects along with the `User` object that references them, the application must define the `IPersistent.preDestroyPersistent()` hook method.

Destroying Objects Referenced by Destroyed Objects

When an application calls the `ObjectStore.destroy()` method, ObjectStore calls the `preDestroyPersistent()` method before actually destroying the specified object. A user-defined class should override this method to destroy any internal persistent data structures that it references. In the Personalization application, the `preDestroyPersistent()` method, as defined on the `User` class, looks like the following:

```
public void preDestroyPersistent()
{
  if (!ObjectStore.isDestroyed(name))
    ObjectStore.destroy(name);
  if (!ObjectStore.isDestroyed(email))
    ObjectStore.destroy(email);
  /* destroy each of the interests */
  Iterator interestIter = interests.values().iterator();
  while (interestIter.hasNext()) {
    Interest interest = (Interest) interestIter.next();
    ObjectStore.destroy(interest);
  }
  /* destroy the interests list too */
  ObjectStore.destroy(interests);
  }
```

When you call the `ObjectStore.destroy()` method on an object, it removes the primitive objects that are referenced by this object but it does not destroy fields in the object that are `String`, `Long`, or `Double` types. For `User` objects, then, the `PIN` attribute, which is an `int`, is deleted automatically. But the application must explicitly destroy the `name` and `email` strings. In addition, destroying a `Map` does not touch any of the objects referenced by that map. Therefore, the application must iterate over the map of interests and destroy each of the `Interest` objects before destroying the map itself.

Destroying Strings

Before deleting the `email` and `name` strings, the application checks whether they have already been destroyed. If the user’s `name` and `email` happened to have the same value, they could refer to the same `String` object in the database. This is because ObjectStore supports `String` pooling where only one copy of the `String` is stored in the database. Because Java `String`s are immutable, this presents a problem when you explicitly delete a `String` object. See the *Java API User Guide*, Destroying Strings.

About the Persistent Garbage Collector

ObjectStore provides a persistent garbage collector utility that automatically handles removing objects from the database. When using ObjectStore, you need not concern yourself with deleting objects. You are responsible for disconnecting your objects from relationships and associations, but ObjectStore can take care of removing all unreachable objects from the database.

The primary advantage of using the persistent garbage collector is that you do not have to write code to delete objects from the database. This allows you to avoid the problems associated with explicit deletion, such as dangling references and orphaned objects. See the *Java API User Guide*, Performing Garbage Collection in a Database.

Using Collections

The Personalization application uses collection objects to keep track of registered users, to store the related interests each user has, and to keep track of all defined interests.

Java collections

Sun’s JDK 1.2 has been enhanced to support collections. A *collection* (also known as a *container*) is a single object (such as Java's familiar `Vector` class) that represents a group of objects. The JDK 1.2 collections API is a unified framework for representing and manipulating collections and for allowing them to be manipulated independently of the details of their representation.

Collections, including `List`s, `Set`s, `Map`s, and others help improve reuse and interoperability. They allow you to implement generic, reusable data objects that conform to the standard `Collection` interfaces. They also enable you to implement algorithms that operate on `Collection` objects independently of the details of their representation.

ObjectStore collections

ObjectStore provides several collection implementations that implement the `Collection` interfaces defined in the JDK 1.2 and that provide improved scalability. These include

- `OSVectorList`
- `OSHashSet`
- `OSTreeSet`
- `OSHashBag`

- `OSHashMap`
- `OSTreeMap`

ObjectStore supports the hash table and vector representations, which are designed to provide good performance for persistent collections. `OSTreeSet` and `OSTreeMap` are based on a new B-tree implementation that is designed specifically for persistent collections with very large extents (hundreds of thousands of entries). Rather than causing your Java application to wait for all objects to be read into the collection from the database when the collection is first accessed, ObjectStore loads objects dynamically, several at a time, as your application navigates the elements in the collection.

The Personalization application uses a `Map` (`OSHashMap`) to keep track of all registered users. The user's name is the key for the map. In addition, the `User` class uses a `Map` to store the related interests that a particular user has.

`Map`s introduce a new requirement for classes of objects that will be stored as keys in persistent collections. These classes must provide a suitable `hashCode()` method. Objects that are stored as keys in `Map`s must provide hash codes that remain the same across transactions. The default `Object.hashCode()` method supplies an identity-based hash code. This identity hash code might depend on the virtual memory address or on internal implementation-level metadata associated with the object. Such a hash code is unsuitable for use in a persistent identity-based `Map` because it might be different each time an object is fetched from the database.

For your persistence-capable classes, you can override the `hashCode()` method and supply your own, or you can rely on the class file postprocessor to supply a `hashCode()` method suitable for storing instances in persistent hash tables. See the *Java API User Guide*, Storing Objects as Keys in Persistent Hash Tables, for more information about supplying your own `hashCode()` methods.

The Personalization application uses a set (`OSHashSet`) to keep track of all interests that are defined. Because the `Interest` objects are stored in `Map`s that are part of the `User` objects, the application does not need the `allInterests` set to make `Interest` objects persistent. The `allInterests` set is useful because it allows you to perform queries efficiently, such as "find all users who have a particular interest."

For information on querying and indexing collections, see Chapter 5, Using ObjectStore to Query a Database, on page 37.

Chapter 4
Compiling and Running an ObjectStore Program

To run an application, such as the Personalization application, you must perform the steps described in the following sections:

Installing ObjectStore

For information about installations, see the `README.htm` file in the top-level directory of your ObjectStore installation.

After the installation is complete, update your `PATH` environment variable to contain the `bin` directory from the installation. This allows you to use the Class File Postprocessor (`osjcfp`) and other tools.

For example, under Windows you would add the following entry to your `PATH` environment variable:

```
c:\Odi\osji\bin
```

Adding Entries to Your CLASSPATH

ObjectStore requires certain entries in the `CLASSPATH` environment variable before you can use it. When you want to develop ObjectStore programs as well as use ObjectStore, you must add additional entries to your `CLASSPATH`.

Entries Required to Run ObjectStore Applications

To use ObjectStore, you must set your `CLASSPATH` environment variable to contain

- The `osji.jar` file. This allows the Java VM to locate ObjectStore.
- The ObjectStore `tools.jar` file. This allows the Java VM to locate the ObjectStore files for the Class File Postprocessor and other ObjectStore database tools.

You must have these `.jar` files explicitly listed in your class path. You cannot list only an entry for the directory that contains them. For example, under Windows, you might have the following entries in your `CLASSPATH` variable:

```
c:\Odi\osji\osji.jar;c:\Odi\osji\tools.jar
```

Entries Required to Develop ObjectStore Applications

To develop and run ObjectStore applications, you must add entries to the `CLASSPATH` variable that allow Java and ObjectStore to find your

- Source directory
- Annotated class file directory

The source directory contains your Java source files and your class files (compiled source). As described in Definition of Persistence Capable on page 20, ObjectStore must postprocess those class files that you want to make into persistence-capable and persistence-aware classes. ObjectStore takes these `.class` files and produces new `.class` files that contain annotations that are needed for persistence.

You can decide whether to place these annotated files in the same directory as the source files or in a separate directory. For example, suppose that `c:\Odi\osji\com\odi\tutorial` is the directory that contains the source files for the `tutorial` package. You can decide to annotate your class files in place. That is, you can instruct the postprocessor to overwrite your original class files with the annotated class files. To do this, you must add the following entry to your `CLASSPATH` variable:

```
c:\Odi\osji
```

This entry allows the Java VM and ObjectStore to find the `com\odi\tutorial` directory, which contains both the source files and the annotated class files.

If you decide to place the annotated class files in a directory that is separate from that of your source files, for example, `c:\Odi\osji\com\odi\tutorial\osjcfpout`, you must add the following entries to your `CLASSPATH` variable:

```
c:\Odi\osji\com\odi\tutorial\osjcfpout;c:\Odi\osji
```

The first entry allows ObjectStore to find the annotated class files. The second entry allows ObjectStore to find your source files.

Background About Different Kinds of Class Files

In general, when you are developing an ObjectStore application, you are concerned about three kinds of `.class` files:

- Annotated class files that represent persistence-capable or persistence-aware classes.
- Superfluous class files that were input to the postprocessor and are now superseded by the annotated class files. These are the original class files created by the compiler.
- Unannotated class files that do not need to be postprocessed but that are still required by your application.

After compiling your application, ObjectStore has to find the original class files to annotate them. The postprocessor gives you the option of annotating the class files in place, which means that the superseded class files are replaced by the annotated class files. This is generally the easiest way to use ObjectStore because your `CLASSPATH` does not have to contain a separate entry for the annotated class files.

After postprocessing your application, you can run your application. When you run your program, Java has to locate the annotated class files and the unannotated class files that have not been superseded.

The location of the annotated class files must precede the location of your original class files in the `CLASSPATH`. This allows the Java VM to find the annotated class files before it finds your superseded (original) class files. You can find detailed instructions for doing this in the *Java API User Guide*, Chapter 8, Generating Persistence-Capable Classes Automatically.

Compiling the Program

You compile an ObjectStore application in the same way that you compile any other Java application. For example, to compile the Personalization application, change to the `c:\Odi\osji\com\odi\tutorial` directory and enter

```
javac *.java
```

You can use the asterisk (*) to compile all `.java` files or you can compile each file individually by specifying the file name. Case is significant for the file name, so you must specify, for example, `User.java` and not `user.java`.

When you compile the Personalization application, the `javac` compiler outputs the following run-time byte code files:

- `User.class`
- `Interest.class`
- `UserManager.class`
- `TestDriver.class`
- `PersonalizationException.class`

Running the Postprocessor

You must run the class file postprocessor (`osjcfp`) on the class files of the classes that you want to be persistence capable. The postprocessor generates new annotated files in place (overwrites original class files) or in a directory that you specify. When you run your program, you use the annotated class files and not your original class files.

Before you run the postprocessor, ensure that the `bin` directory that contains the postprocessor executable is in your path, as noted in Adding Entries to Your CLASSPATH on page 30.

Complete information about the postprocessor is in the *Java API User Guide*, Chapter 8, Generating Persistence-Capable Classes Automatically.

Description of Output from the Postprocessor

When you run the postprocessor to make a class persistence capable, the postprocessor creates an annotated class file for each class it postprocesses. For persistence-capable classes that are private, the postprocessor generates additional files named `xxxxClassInfo.class`, where `xxxx` is the class name.

Since none of the classes in the Personalization application are private, the `osjcfp` command given above generates the following annotated class files:

- `Interest.class`
- `User.class`

Example of Postprocessing Classes in Place

In the Personalization application, both the `Interest` and `User` classes are persistence capable. To postprocess these classes in place, change to the `c:\Odi\osji\com\odi\tutorial` directory and enter

```
osjcfp -inplace -dest . Interest.class User.class
```

When you specify the `-inplace` option, the postprocessor ignores the destination argument (`-dest`), but it is still required.

Specifying an Input File to the Postprocessor

If you have several class files to postprocess, you might find it easier to use a file to specify the arguments. To do this, create a text file that contains the arguments for the postprocessor and specify the text file name with the `@` sign when you run the postprocessor.

For example, suppose that in the `c:\Odi\osji\com\odi\tutorial` directory you create the `cfpargs` file with the following contents:

```
-inplace -dest
-pc User.class Interest.class
```

The `-pc` option instructs the postprocessor to make the specified classes persistence capable. You can then run the postprocessor with the following command:

```
osjcfp @cfpargs
```

Placing the Annotated Files in a Separate Directory

To put the annotated files in a separate directory, specify the `-dest` option followed by the name of the directory in which you want the postprocessor to put the annotated class files. For example, use the following command to place the annotated `User` and `Interest` class files in the `osjcfpout` directory:

```
osjcfp -dest osjcfpout Interest.class User.class
```

The `-dest` option specifies the destination directory for the annotated files. The argument for this option must be the same as the directory in your `CLASSPATH` environment variable that establishes the location of the annotated files. In this example, `CLASSPATH` should point to `c:\Odi\osji\com\odi\tutorial\osjcfpout`.

You must explicitly create the output directory before you run the postprocessor. The postprocessor creates a directory structure that matches your package structure, in this case, `c:\Odi\osji\com\odi\tutorial\osjcfpout\com\odi\tutorial`.

Additional Information About the Postprocessor

Under normal circumstances, you must postprocess together all classes in your application that you want to be persistence capable or persistence aware. Failure to do so can result in problems that are difficult to diagnose when you run your application. For example, objects might not be fetched automatically from the database when needed.

The postprocessor must be able to examine all class files in an application when it makes any class in the application persistence capable. There are postprocessor options that allow you to determine the classes that the postprocessor makes persistence capable. If it is inconvenient or impossible to postprocess all classes in your application together, you can postprocess separate batches of files. See Postprocessing a Batch of Files Is Important in the *Java API User Guide*.

It is good practice to provide accessor methods to encapsulate state in an object, as shown in the source code for the `Interest` class in Appendix A, Source Code, on page 47.

When using ObjectStore, accessor methods allow ObjectStore to fetch objects automatically from the database. It is easy to localize where ObjectStore must annotate your code to perform fetch and update checks, which occur only in the accessor methods and are not spread throughout your code.

Running the Program

Before you run an ObjectStore program, ensure that the ObjectStore `lib` directory is in your library search path. Run your ObjectStore program as a Java application.

For example, the following is a typical command line that runs the Personalization application:

```
java com.odi.tutorial.TestDriver test.odb
```

When you run an ObjectStore program, you specify the fully qualified class name to the Java VM. In this example, `com.odi.tutorial.TestDriver` is the fully qualified class name.

The `TestDriver` program expects an argument that contains the pathname of the database's `.odb` file, namely `test.odb`. If you are using PSE Pro, the application also creates `test.odt` and `test.odf`, and these three files (the `.odb`, `.odt`, and `.odf` files) form the database. You can specify any pathname you want, as long as the file name ends with `.odb`. This example uses a relative pathname, so ObjectStore creates the files in the directory in which you run the program.

Sample Output from the application is in Appendix B, Sample Output, on page 63.

Chapter 5
Using ObjectStore to Query a Database

ObjectStore provides a mechanism for querying `java.util.Collection` objects. 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 that contains all elements for which the expression is true.

To accelerate the processing of queries on particularly large collections, you can build indexes on the collection.

Although the Personalization application does not implement queries, this chapter discusses the following topics on queries.

For more information about creating and using queries and indexes, see Working with Collections in Chapter 7 of the *Java API User Guide*.

Querying Collections

Using queries is a two-step process:

1 Create the query.

2 Run the query against a collection.

Creating Queries

To create a query, you run the `Query` constructor and pass in a `Class` object and a query string. The `Class` object specifies the type of element contained by the collection you want to query. This element type must be a publicly accessible class, and any members (fields or methods) specified in the query string must be publicly accessible as well.

The query string is a predicate expression, which is defined with native Java. There is no SQL (Structured Query Language) or JDBC required. Query strings can include standard Java arithmetic, conditional, and relational operators, as well as simple methods.

The following example shows the creation of a query that finds all interests in which the interest name is `"wine"`:

```
Query query = new Query(Interest.class, "getName() ==
\"wine\"");
```

This example uses a public method instead of a field. This allows the `Interest` name field to remain private and preserves the encapsulation of the interest's state. If the name field was public, you could specify the query like this:

```
Query query = new Query(Interest.class, "name == \"wine\"");
```

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 or against different collections.

You can also use variables in query strings. See Specifying Variables in Queries on page 38.

Running Queries Against Collections

You run a query against a specific collection with a call to the `Query.select()` method. This call specifies the collection to be queried. For example, after you define a query as in the previous section, you can run that query like this:

```
Collection wine = query.select(allInterests);
```

In this example, the query tests the elements in the set of `allInterests` to find the elements whose name is `wine`.

Specifying Variables in Queries

You can use variables in queries instead of constants. In the previous example, you might want to substitute for different name values, depending on whether you are looking for wine, food, or some other interest. The following example shows the way to use a variable value in a query expression:

```
String interestName = "wine";
FreeVariables freeV = new FreeVariables();
freeV.put("x", String.class);
Query query = new Query(Interest.class, "getName() == x",
  freeV);
FreeVariableBindings freeVB = new FreeVariableBindings();
freeVB.put("x", interestName);
Collection queryResults = query.select(
  allInterests.values(), freeVB);
```

First, create a `FreeVariables` list and add a variable, `x` in this example, to it. When you add the variable, you specify the type of variable. In this case, the type of `x` is `String` because it is going to represent a `String` in the query string. When you create the query,

- Specify the type of element contained by the collection you want to query, as you would for a query that uses constants.
- In the query string, specify the variable added to the `FreeVariables` list, for example, replace the value `"wine"` with the variable `x`.
- Pass the `FreeVariables` list, `freeV` in this example, as an argument to the `Query` constructor.

Binding variables

Before you can execute the query, you must bind the variable in the query string to a variable in the program. To do this, create a `FreeVariableBindings` list. In this example, `freeVB` binds the variable `x` to the variable `interestName`.

Sample query with a variable

When you execute the query, pass the `FreeVariableBindings` list as an argument to the query `select()` method. For example, the following is a method that finds all users with a particular interest:

```
/**
 * Get all users with a particular interest.
 *
 * @param interestName: The name of the interest.
 *
 * @return An array of names of users with this interest.
 **/

public static String[] find Users(String interestName)
  throws PersonalizationException
{
  Transaction tr = Transaction.begin(ObjectStore.READONLY);
  String queryString = "getName() == \"" + interestName + "\"";
  Query interestQuery = new Query(Interest.class, queryString);
```

```
  Collection interests = interestQuery.select(allInterests);
  String[] users = new String[interests.size()];
  int index = 0;
  Iterator iter = interests.iterator();
  while (iter.hasNext())
    users[index++] =
      ((Interest)iter.next()).getUser().getName();
  tr.commit(ObjectStore.RETAIN_READONLY);
  return users;
}
```

Using Indexes to Speed Query Performance

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. You can add indexes to any collection that implements the `com.odi.util.IndexedCollection` interface. This interface provides methods for adding and removing indexes and for updating indexes when the indexed data changes.

While querying is supported on all `com.odi.util` collections, only `com.odi.util.OSTreeSet` already implements the `IndexedCollection` interface. This means that if you want to add an index to another type of collection (other than `OSTreeSet`), you must define a collection class that implements the `IndexedCollection` interface.

What an Index Does

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 field executes faster. This is because it is not necessary to examine each object in the collection to determine those elements that match the predicate expression. Also, ObjectStore does not need to fetch into memory the elements that do not match the query.

To use an index, you create it, and then specify the indexed field or method in a query. A query can include both indexed fields and methods and nonindexed fields and methods. ObjectStore evaluates the indexed fields and methods first and establishes a preliminary result set. ObjectStore then applies the nonindexed fields and methods to the elements in the preliminary result set.

Creating an Index

Use these methods, defined on `IndexedCollection`, to create an index:

```
addIndex(Class, String)
```

```
addIndex(Class, String, boolean, boolean)
```

The `Class` argument indicates the element type to which the index applies. The `String` indicates the element member to be indexed.

The optional `boolean` 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.

Example of Creating an Index

The following example shows the creation of an index on the return value from the `getName()` method on the `Interest` class. To execute this, the collection that contains the `Interest` objects, `allInterests`, must implement the I`ndexedCollection` interface. In this example, this is accomplished by specifying `allInterests` to be an `OSTreeSet` collection. For example, you could have the following code in the `UserManager.initialize()` method in the Personalization application:

```
db.createRoot("allInterests", allInterests =
  new OSTreeSet(db));
allInterests.addIndex(Interest.class, "getName()");
```

Maintenance Required After Changing Indexed Elements

After you add an index to a collection, ObjectStore automatically maintains the index as you add to or remove elements from the collection. However, it is your responsibility to update the index when indexed members change in instances that are already elements of an indexed collection.

For example, suppose you insert `Jones` into a collection called `userCollection`, and then you build an index for `userCollection` on the `email` field. If you remove `Jones` from the collection, ObjectStore updates the email index so it no longer includes the entry for the `Jones` object. However, if you leave `Jones` in the collection, but change Jones's `email` address, you must manually update the index to contain the correct email entry.

To update an index, you must

1 Remove the incorrect instance from the index. For example, remove `Jones` from the index.

2 Update the incorrect instance. For example, modify the `email` address for `Jones`.

3 Add the updated instance to the index. For example, add the updated `Jones` object to the index.

An example follows. For more information, see Enhancing Query Performance with Indexes in Chapter 7 of the *Java API User Guide*.

```
User jones = /* assume jones references Jones */
userCollection.removeFromIndex(User.class, "email", jones);
jones.setEmail("jones@objectdesign.com");
userCollection.addToIndex(User.class, "email", jones);
```

Chapter 6
Choosing PSE Pro or ObjectStore

This section compares PSE Pro and ObjectStore. It considers the following characteristics:

Overall Capability

PSE Pro supports large single-user databases and queries over large collections.

ObjectStore delivers high-performance object storage for both Java and C++ programs. It provides scalable concurrent access to very large databases in a distributed multitiered environment. ObjectStore provides complete database management system features that ensure reliability and high availability. This includes backup and recovery, security, roll forward, replication, and failover.

The ObjectStore Java API is a superset of the PSE Pro for Java API. It is easy to migrate applications from PSE Pro to ObjectStore when additional features are required.

Database Size

PSE Pro can handle databases that exceed the tens-of-megabytes range, and is capable of handling databases that contain millions of objects and databases whose sizes are in the hundreds-of-megabytes range. Beyond that, ObjectStore handles even larger databases, in the tens- to hundreds-of-gigabytes range. ObjectStore also provides object clustering to support finer control over the physical placement of objects, which is necessary in very large databases.

Concurrent Users

PSE Pro is not intended for large numbers of concurrent users. They can support multiple readers, but writers are serialized because locks are held at the database level. If you require that your application support a high volume of concurrent updates, you should consider ObjectStore. With ObjectStore, multiple users can concurrently access and update objects that are stored in several databases distributed around a network.

Collections

PSE Pro and ObjectStore both support high-performance, indexed queries over large collections. PSE Pro and ObjectStore provide robust collections libraries that support storage and indexed associative retrieval of large groups of objects. These libraries provide arrays, lists, bags, and sets, with B-tree and hash table indexing. PSE Pro and ObjectStore also provide query optimizers, which formulate efficient retrieval strategies and minimize the number of objects that must be examined in response to a query.

Integrity, Reliability, and Recovery

Both PSE Pro and ObjectStore ensure the integrity and reliability of your data. ObjectStore provides on-line backup and recovery, roll forward, replication, and failover to enable full support for administration of the database in a highly available 7x24 environment.

Multimedia Content Management

ObjectStore includes a comprehensive library of Object Managers that provide support for multimedia content management. Today’s Java applications make extensive use of data types such as image, audio, full text, video, and HTML. The Object Manager for each of these types helps you maintain this data. In addition, Object Managers are useful for specialized data types such as spatial and time series. Support for multimedia data types goes beyond storage management. An extended data type can also have sophisticated behavior defined by its methods, such as content-based retrieval of images.

Ease of Using Java

PSE Pro and ObjectStore provide powerful data management environments for Java applications. They provide a seamless binding to the Java language. The easy-to-use interface drastically reduces the amount of code required to manage persistent Java objects, but it still provides developers with the full power of Java to define, manipulate, and share important application data.

Whether you use PSE Pro or ObjectStore, creating persistent Java objects is as easy as creating transient Java objects. This means that the increased productivity of the Java environment is further enhanced with PSE Pro and ObjectStore.

Appendix A Source Code

This chapter provides the source code for the following Personalization application classes:

Source Code for Interest.java

```
package com.odi.tutorial;

import com.odi.*;

/**
 * The Interest class models a particular interest that a user may
 * have. It contains a name and a value. For example, the name of
 * an interest might be "food", and the value might be "pizza".
 */

public class Interest
{
  /* the name of the interest */
  private String name;

  /* the value of the interest */
  private String value;

  /* the user who has this interest */
  private User user;
```

```
  /* accessor functions */
  public String getName() { return name; }
  public String getValue() { return value; }
  public void setValue(String value) { this.value = value; }
  public User getUser() { return user; }

  /**
   * Constructs a new interest object given a name and a value
   */
  public Interest(String name, String value, User user)
  {
    this.name = name;
    this.value = value;
    this.user = user;
  }

  /*
   * Destroy the interest's associated objects
   */
  public void preDestroyPersistent()
  {
    if (!ObjectStore.isDestroyed(name))
      ObjectStore.destroy(name);
    if (!ObjectStore.isDestroyed(value))
      ObjectStore.destroy(value);
  }
}
```

Source Code for User.java

```
package com.odi.tutorial;

import java.util.*;
import com.odi.*;
import com.odi.util.*;

/**
 * The User class models users. Users have names, email addresses,
 * and PINS. Each user also has a set of interests. The application
 * uses PINs to validate a user's identity.
 */
public
class User {

  /* The name of the user */
  private String name;

  /* The user's email address */
  private String email;
```

```
/* The user's Personal Identification Number */
private int PIN;

/* The set of user's interests */
private OSHashMap interests;

/* accessors */
public String getName() { return name; }
public String getEmail() { return email; }
public int getPIN() { return PIN; }
public Map getInterests() { return interests; }

/**
 * Constructs a user object given the name, email and PIN
 */
public User(String name, String email, int PIN) {
  this.name = name;
  this.email = email;
  this.PIN = PIN;
  this.interests = new OSHashMap(5); /* initial hashtable size */
}

/*
 * Destroy the associated objects
 */
public void preDestroyPersistent() {
  if (!ObjectStore.isDestroyed(name))
    ObjectStore.destroy(name);
  if (!ObjectStore.isDestroyed(email))
    ObjectStore.destroy(email);

  /* destroy each of the interests */
  Iterator interestIter = interests.values().iterator();
  while (interestIter.hasNext()) {
    Interest interest = (Interest) interestIter.next();
    ObjectStore.destroy(interest);
  }
  /* destroy the interests list too */
  ObjectStore.destroy(interests);
}

/**
 * Add an interest to the User's list of interests.
 *
 * @param interestName the name of the interest
 * @param interestValue the value of the interest
 *
 * @exception PersonalizationException If the interest is
 * already there (the same name as another interest)
 */
public Interest addInterest(String interestName, String interestValue)
     throws PersonalizationException
```

```
  {
    Object previous = interests.get(interestName);
    if (previous != null)
      throw new PersonalizationException("Interest already there: " +
        interestName);

    Interest interest = new Interest(interestName, interestValue, this);
    interests.put(interestName, interest);
    return interest;
  }

  /**
   * Update an interest in the User's list of interests.
   *
   * @param interestName the name of the interest
   * @param interestValue the new value of the interest
   *
   * @exception PersonalizationException is thrown if the interest is
   * already not there.
   */
  public Interest changeInterest(String interestName, String interestValue)
       throws PersonalizationException
  {
    Interest interest = (Interest)interests.get(interestName);
    if (interest == null)
      throw new PersonalizationException("No such registered interest: " +
        interestName);
    interest.setValue(interestValue);
    return interest;
  }

  /**
   * Remove an interest from the User's list of interests.
   *
   * @param interestName: The name of the Interest to remove.
   *
   * @exception PersonalizationException is thrown if the interest is not
   * found in the user's list of interests
   */
  public Interest removeInterest(String interestName)
       throws PersonalizationException
  {
    Interest interest = (Interest)interests.remove(interestName);
    if (interest == null)
      /* did not find the interest */
      throw new PersonalizationException("Interest not found: " + name);

   return interest;
  }

}
```

Source Code for UserManager.java

```
package com.odi.tutorial;

import java.util.*;
import java.io.*;
import com.odi.*;
import com.odi.util.*;
import com.odi.util.query.*;

/**
 * The UserManager acts as the interface to the Personalization data
 * that is stored persistently. In real use, it might serve as an
 * application service that would receive requests (RMI or ORB
 * requests perhaps) and service those requests by accessing the
 * database.
 */
public
class UserManager
{
  /* The database that this UserManager is operating against. */
  private static Database db;

  /* The active session for this UserManager */
  private static Session session;

  /* The extent of all users in the database is held in a root.
     We use a map, whose key is the name of the user. */
  private static Map allUsers;

  /* The extent of all interests in the database is held in a root.
     We use a Set, which might have one or more indexes. */
  private static Set allInterests;

  /* This is used to allocate the Personal Identification Number (PIN) */
  private static Random pinGenerator;

  public static void initialize(String dbName)
  {
    /* initialize a random number generator to allocate PINs */
    pinGenerator = new Random();

    /* Create a session and join this thread to the new session. */
    session = Session.create(null, null);
    session.join();

    /* Open the database or create a new one if necessary. */
    try {
      db = Database.open(dbName, ObjectStore.UPDATE);
    } catch (DatabaseNotFoundException e) {
      db =
       Database.create(dbName, ObjectStore.ALL_READ | ObjectStore.ALL_WRITE);
```

```
    }
    /* Find the allUsers and allInterests roots or create them if not there. */
    Transaction tr = Transaction.begin(ObjectStore.UPDATE);
    try {
      allUsers = (Map) db.getRoot("allUsers");
      allInterests = (Set) db.getRoot("allInterests");
    } catch (DatabaseRootNotFoundException e) {

      /* Create the database roots and give them appropriate values */
      db.createRoot("allUsers", allUsers = new OSHashMap());
      db.createRoot("allInterests", allInterests = new OSHashSet());
    }

    /* End the transaction and retain a handle to allUsers and allInterests */
    tr.commit(ObjectStore.RETAIN_HOLLOW);
    return;
  }

  /**
   * Close the database and terminate the session.
   */
  public static void shutdown()
  {
    db.close();
    session.terminate();
  }

  /**
   * Add a new user to the database; if the user's name already
   * exists, throw an exception.
   *
   * @param name: The name of the user to be added
   * @param email: The email address of the user
   *
   * @return: The PIN of the new user.
   *
   * @exception PersonalizationException is thrown if the user
   * is already there.
   */
  public static int subscribe(String name, String email)
    throws PersonalizationException
  {
    Transaction tr = Transaction.begin(ObjectStore.UPDATE);

    /* First check to see if the user's name is already there. */
    if (allUsers.get(name) != null) {
      tr.abort(ObjectStore.RETAIN_HOLLOW);
      throw new PersonalizationException("User already there: " + name);

    }

    /* The user name is not there so add the new user;
```

```
     first generate a PIN in the range 0..10000. */
  int pin = pinGenerator.nextInt() % 10000;
  if (pin < 0) pin = pin * -1;
  User newUser = new User(name, email, pin);
  allUsers.put(name, newUser);

  tr.commit(ObjectStore.RETAIN_HOLLOW);
  return pin;
}

/**
 * Removes the user from the database.
 *
 * @param name: The name of the user to be removed.
 *
 * @exception PersonalizationException is thrown if the user is
 * not found.
 */
public static void unsubscribe(String name)
  throws PersonalizationException
{
  Transaction tr = Transaction.begin(ObjectStore.UPDATE);

  User user = (User) allUsers.get(name);
  if (user == null) {
    tr.abort(ObjectStore.RETAIN_HOLLOW);
    throw new PersonalizationException ("Could not find user: " + name);
  }

  /* remove the user from the allUsers collection, and
   * remove all of the user's interests from the allInterests collection */
  allUsers.remove(name);
  Iterator interests = user.getInterests().values().iterator();
  while (interests.hasNext())
    allInterests.remove(interests.next());

  /* finally destroy the user and all its subobjects */
  ObjectStore.destroy(user);

  tr.commit(ObjectStore.RETAIN_HOLLOW);
}

/**
 * Validates a username / PIN pair, and returns the user object.
 */
public static User validateUser(String userName, int PIN)
{
  Transaction tr = Transaction.begin(ObjectStore.READONLY);
  User user = (User) allUsers.get(userName);
  if (user == null) {
    tr.abort(ObjectStore.RETAIN_HOLLOW);
    throw new PersonalizationException("Could not find user: " + userName );
```

```
    }
    if (user.getPIN() != PIN) {
      tr.abort(ObjectStore.RETAIN_HOLLOW);
      throw new PersonalizationException("Invalid PIN for user: " + userName );
    }
    tr.commit(ObjectStore.RETAIN_READONLY);
    return user;
  }

  /**
   * Add an interest to an existing user's set of interests.
   *
   * @param userName: The name of the user.
   * @param interestName: The name of the interest to create.
   * @param interestValue: The value of the new interest.
   *
   * @exception PersonalizationException: thrown if the user is
   * not found or if the user already has this interest.
   */
  public static void addInterest(String userName, String interestName,
     String interestValue)
    throws PersonalizationException
  {
    Transaction tr = Transaction.begin(ObjectStore.UPDATE);
    User user = (User) allUsers.get(userName);
    if (user == null) {
      tr.abort(ObjectStore.RETAIN_HOLLOW);
      throw new PersonalizationException("User not found: " + userName);
    }
    else {
      try {
        Interest interest = user.addInterest(interestName, interestValue);
        allInterests.add(interest);
      } catch (PersonalizationException e) {
        tr.abort(ObjectStore.RETAIN_HOLLOW);
        throw e;
      }
    }

    tr.commit(ObjectStore.RETAIN_HOLLOW);
    return;
  }

  /**
   * Remove an interest from a user's set of interests.
   *
   * @param userName: The name of the user.
   * @param interestName: The name of the interest to remove.
   *
   * @exception PersonalizationException: thrown if the user is
```

```
 * not found or the interest is not found.
 */
public static void removeInterest(String userName, String interestName)
  throws PersonalizationException
{
  Transaction tr = Transaction.begin(ObjectStore.UPDATE);
  User user = (User) allUsers.get(userName);
  if (user == null) {
    tr.abort(ObjectStore.RETAIN_HOLLOW);
    throw new PersonalizationException("User not found: " + userName);
  }
  else {
    try {
      Interest interest = user.removeInterest(interestName);
      allInterests.remove(interest);
      ObjectStore.destroy(interest);
    } catch (PersonalizationException e) {
      tr.abort(ObjectStore.RETAIN_HOLLOW);
      throw e;
    }
  }

  tr.commit(ObjectStore.RETAIN_HOLLOW);
  return;
}

/**
 * Update an interest in an existing user's set of interests.
 *
 * @param userName: The name of the user.
 * @param interestName: The name of the interest to modify.
 * @param interestValue: The new value of the interest.
 *
 * @exception PersonalizationException: Thrown if the user is
 * not found or if the user does not already have this interest.
 */
public static void changeInterest(String userName, String interestName,
    String interestValue)
  throws PersonalizationException
{
  Transaction tr = Transaction.begin(ObjectStore.UPDATE);
  User user = (User) allUsers.get(userName);
  if (user == null) {
    tr.abort(ObjectStore.RETAIN_HOLLOW);
    throw new PersonalizationException("User not found: " + userName);
  }
  else {
    try {
      user.changeInterest(interestName, interestValue);
    } catch (PersonalizationException e) {
```

```
        tr.abort(ObjectStore.RETAIN_HOLLOW);
        throw e;
      }
    }

    tr.commit(ObjectStore.RETAIN_HOLLOW);
    return;
  }

  /**
   * Get the interests for a particular user.
   *
   * @param userName: The name of the user.
   *
   * @return: a Set of interests.
   *
   * @exception PersonalizationException: Thrown if the user is
   * not found.
   */
  public static Collection getInterests(String userName)
    throws PersonalizationException
  {
    Transaction tr = Transaction.begin(ObjectStore.READONLY);

    User user = (User) allUsers.get(userName);
    if (user == null) {
      tr.abort(ObjectStore.RETAIN_HOLLOW);
      throw new PersonalizationException ("User not found: " + userName);
    }

    /* recursively fetch all of the objects accessible from the users
     * set of interests, so that they can be returned and accessed
     * outside of a transaction */
    ObjectStore.deepFetch(user.getInterests());

    /* Commit using RETAIN_READONLY so the interests can be accessed
     * outside of this transaction. */
    tr.commit(ObjectStore.RETAIN_READONLY);

    return user.getInterests().values();
  }

  /**
   * Retrieves all of the users.
   *
   * @return an array containing the names of all registered users.
   *
   * @exception Exception:
   */
  public static String[] getUserNames()
  {
    Transaction tr = Transaction.begin(ObjectStore.READONLY);
```

```
    String[] names = new String[allUsers.size()];
    Iterator userIter = allUsers.values().iterator();
    int userIndex = 0;

    while (userIter.hasNext())
      names[userIndex++] = ((User)userIter.next()).getName();

    tr.commit(ObjectStore.RETAIN_HOLLOW);
    return names;
  }

}
```

Source Code for TestDriver.java

```
package com.odi.tutorial;

import com.odi.util.*;
import java.util.*;
import java.io.*;

/**
 * The TestDriver class exercises the UserManager code. It
 * implements a simple UI that is driven by terminal I/O.
 */
public
class TestDriver
{
  /* Main: reads input commands from the terminal and exercises
   * the UserManager code.
   */
  public static void main(String argv[])
  {
    if (argv.length < 1) {
      System.out.println("Usage: java TestDriver <databaseName>");
      return;
    }

    /* the database to operate on is a command-line argument;
     * the file to read commands from is an optional second argument
     * (if no input file is specified, commands are read from System.in)
     */
    String dbName = argv[0];
    InputStream input = System.in;
    if (argv.length > 1)
      try {
      input = new FileInputStream(argv[1]);
    } catch (FileNotFoundException e){}

    /* initialize the UserManager, which opens the database */
```

```
    UserManager.initialize(dbName);

    /* read command input */
    BufferedReader instream =
      new BufferedReader(new InputStreamReader(input));

    /* print help message describing the legal commands */
    printHelp();

    while (true) {
      try {
        System.out.println();

        /* read a line of command input */
        String inputLine = instream.readLine();
        if (inputLine == null) { /* end of input */
          UserManager.shutdown();
          return;
        }

        /* tokenize the command input with a StringTokenizer */
        StringTokenizer tokenizer = new StringTokenizer(inputLine, "");
        if (!tokenizer.hasMoreTokens()) continue;
        String command = tokenizer.nextToken();
        System.out.println();
    /* ****************** */
    /*HELP*/
    /* ****************** */
        if ("help".startsWith(command)) {
          printHelp();
        }
    /* ****************** */
    /* SUBSCRIBE NEW USER*/
    /* ****************** */
        else if ("subscribe".startsWith(command)) {
          int PIN = UserManager.subscribe(readString(tokenizer) /*userName */,
            readString(tokenizer) /*userEmail */);
          System.out.println("Your personal identification number is " + PIN);
        }
    /* ****************** */
    /*UNSUBSCRIBE USER */
    /* ****************** */
        else if ("unsubscribe".startsWith(command)) {
          UserManager.unsubscribe(readString(tokenizer) /* userName */);
        }
    /* ****************** */
    /*VALIDATE USER PIN*/
    /* ****************** */
        else if ("validate".startsWith(command)) {
          User usr  =
            UserManager.validateUser(readString(tokenizer) /*userName */,
```

```
          readInt(tokenizer) /* PIN*/);
        System.out.println("User name " + usr.getName());
        System.out.println("PIN: " + usr.getPIN());
        System.out.println("email: " + usr.getEmail());
      }
  /* ****************** */
  /* LIST ALL USERS*/
  /* ****************** */
      else if ("listusers".startsWith(command)) {
        String[] names = UserManager.getUserNames();

        if (names.length == 0)
          System.out.println("There are no registered users.");
        for (int i = 0; i<names.length; i++) {
          System.out.println("" + names[i]);
        }
      }
  /* ****************** */
  /* ADD AN INTEREST */
  /* ****************** */
      else if ("addinterest".startsWith(command)) {
        UserManager.addInterest(readString(tokenizer) /* userName */,
          readString(tokenizer) /* interest name */,
          readString(tokenizer) /* interest value */);
      }
  /* ****************** */
  /* REMOVE AN INTEREST */
  /* ****************** */
      else if ("removeinterest".startsWith(command)) {
        UserManager.removeInterest(readString(tokenizer) /* userName */,
          readString(tokenizer) /* interest name */);
      }
  /* ****************** */
  /*CHANGE AN INTEREST */
  /* ****************** */
      else if ("changeinterest".startsWith(command)) {
        UserManager.changeInterest(readString(tokenizer) /* userName */,
          readString(tokenizer) /* interest name */,
          readString(tokenizer) /* interest value */);
      }
  /* ****************** */
  /* LIST USER INTERESTS */
  /* ****************** */
      else if ("interests".startsWith(command)) {
        String userName = readString(tokenizer);
        Collection interests =
          UserManager.getInterests(userName);
        Iterator iter = interests.iterator();
        if (!iter.hasNext())
```

```
            System.out.println("" + userName +
              " has no registered interests.");
          while (iter.hasNext()) {
            Interest i = (Interest)iter.next();
            System.out.println("" + i.getUser().getName() +
             " is interested in " + i.getName() + ": " + i.getValue());
          }
        }
    /* ****************** */
    /*   EXIT PROGRAM */
    /* ****************** */
        else if ("exit".startsWith(command)) {
          UserManager.shutdown();
          return;
        }
    /* ******************* */
    /* UNRECOGNIZED COMMAND */
    /* ******************* */
        else {
          System.out.println("Command not recognized.Try \"help\"");

        }
      } catch (PersonalizationException e) {
        System.out.println("" + e.toString());
      }
      catch (Exception e) {
        System.out.println("" + e.toString());
        UserManager.shutdown();
        return;
      }
    }
   }

  static void printHelp()
  {
    System.out.println();
    System.out.println("Each command consists of the command name,
      and a (possibly empty)");
    System.out.println("list of arguments, separated by spaces.");
    System.out.println();
    System.out.println("Legal commands are:");
    System.out.println("help // print this message");
    System.out.println("subscribe <username> <email>
      // enter a new user into the db");
    System.out.println("unsubscribe <username>
      // remove a user from the db");
    System.out.println("validate <username> <PIN>
      // validate PIN and display user data");
    System.out.println("listusers // list all users");
    System.out.println("addinterest <username> <interestname> <value>
```

```
        // register an interest ");
      System.out.println("removeinterest <username> <interestname>
        // unregister an interest ");
      System.out.println("changeinterest <username> <interestname> <value>
        // change an interest ");
      System.out.println("interests <username>
        //display all interests for a user");
      System.out.println("exit// exit the program");
    }

    static String readString(StringTokenizer tokenizer)
    {
      if (tokenizer.hasMoreElements())
        return tokenizer.nextToken();
      else
        throw new PersonalizationException("unexpected end of command input");
    }

    static int readInt(StringTokenizer tokenizer)
    {
      if (tokenizer.hasMoreElements()) {
        String token = tokenizer.nextToken();
        try {
          return Integer.valueOf(token).intValue();
        } catch (NumberFormatException e) {
          throw new PersonalizationException(
            "Number Format Exception reading \"" + token + "\"");
        }
      }
      else
        throw new PersonalizationException("unexpected end of command input");
    }
  }
```

Source Code for PersonalizationException.java

```
package com.odi.tutorial;

import java.util.*;

/**
 * The PersonalizationException is thrown when certain error
 * conditions arise. For example
 *   -- a uid is not found
 *   -- a user already exists in the database
 *   -- an interest is not found
```

```
 *   -- an interest already exists in the user’s set of interests
 */
public final class PersonalizationException extends RuntimeException
{
  public PersonalizationException (String message) {
    super (message);
  }
}
```

Appendix B
Sample Output

Following is some sample input and output from the Personalization application. Each command consists of the command name and a (possibly empty) list of arguments, separated by spaces. Legal commands are described in the following table:

Command	***Action***
`help`	Displays a list of commands with brief descriptions
`subscribe` *`username email`*	Enters a new user in the database
`unsubscribe` *`username`*	Removes a user from the database
`validate` *`username PIN`*	Validates PIN and displays user data
`listusers`	Lists all users
`addinterest` *`username interestname`*	Registers an interest
`removeinterest` *`username interestname`*	Unregisters an interest
`changeinterest` *`username interestname`*	Changes an interest
`interests` *`username`*	Displays all interests for a user
`exit`	Exits the program

Following is the output:

```
subscribe landis landis@objectdesign.com
  Your person identification number is 1489

subscribe obrien obrien@objectdesign.com
  Your person identification number is 7712

validate landis 1489
  User name landis
    PIN: 1489
    email: landis@objectdesign.com

addinterest landis wine burgundy
addinterest obrien wine bordeaux
addinterest obrien sports hockey
listusers
  obrien
  landis

interests obrien
  obrien is interested in wine: bordeaux
  obrien is interested in sports: hockey

changeinterest obrien sports marathon
interests obrien
  obrien is interested in wine: bordeaux
  obrien is interested in sports: marathon

exit
```

Index