OBJECTSTORE

DEVELOPING JAVA APPLICATIONS THAT ACCESS C++

RELEASE 3.0

October 1998

ObjectStore Developing Java Applications That Access C++

ObjectStore Java Interface Release 3.0, October 1998

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

Contents

Preface

Purpose

Developing ObjectStore Java Applications That Access C++ provides information needed to develop an ObjectStore Java application that accesses code written in C++.

Audience

This book is for ObjectStore programmers who want to develop an application in Java that uses classes written in C++. The information in this book assumes that you are knowledgeable about Java and that you are an experienced user of the C++ interface to ObjectStore. You also need the information in the *ObjectStore Java API User Guide.*

Scope

This book does not provide general information about the Java interface to ObjectStore (OSJI). For that information, see the *ObjectStore Java API User Guide.*

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

How This Book Is Organized

Chapter 1, Introduction to Developing ObjectStore Java Applications That Access C++, describes interoperability and peer classes, and summarizes the procedure for developing your application.

Chapter 2, Generating Peer Classes, provides information and instructions for using the tool that maps C++ classes to Java to create peer classes. It also describes how the tool maps some types.

Chapter 3, Writing the Application, discusses how to use peer objects in your application.

Chapter 4, Building the Application, shows how to link your application components for execution.

Chapter 5, Using ObjectStore Peer Collections provides information about storing both primary and peer objects in collections.You can also use queries and indexes with peer collections.

Notation Conventions

This document uses the following conventions:

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

Internet Sources of More Information

Internet gateway

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

Automatic email notification

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

Email discussion list

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

Support

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

Training

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

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

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

Your Comments

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

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

You can also fax your comments to 781.674.5440.

Chapter 1
Introduction to Developing ObjectStore Java Applications That Access C++

Interoperability between C++ and the Java interface to ObjectStore allows an application to access C++ classes from Java. To use the ObjectStore features that allow you do do this, you need an understanding of how ObjectStore maps C++ objects to Java.

Contents

This chapter discusses the following topics:

What Is Interoperability?

ObjectStore provides interoperability between C++ and the Java interface to ObjectStore. Interoperability allows you to write an ObjectStore Java application that can

- Create, populate, and update C++ databases
- Store Java and C++ objects in the same database
- Access ObjectStore C++ collections
- Use ObjectStore ObjectForms

In your application, the Java and C++ code must be executing in a single process.

How Does Interoperability Work?

In an ObjectStore Java application that accesses C++ code, there are two kinds of Java objects, called *primary* objects and *peer* objects.

What Are Primary Objects?

A primary object is a persistence-capable Java object. A persistence-capable object is an object that can be stored in an ObjectStore database. You can use primary objects just as if they were ordinary Java objects. The database correctly records their identity, class, and field values.

ObjectStore can store a primary object in a database in these ways:

- ObjectStore commits a transaction. If a transient primary object is reachable from a persistent object, ObjectStore migrates the transient object to the database.
- Your application calls the **ObjectStore.migrate()** method. This explicitly stores a specified primary object.
- Your application assigns the object as the value of a database root. ObjectStore immediately migrates the object to the database.

What Are Peer Objects?

Peer objects provide a way for Java applications to use C++ objects. A peer object acts as a proxy for a particular C++ object. It has no data fields and so it does not hold any state that is represented by the data members of the corresponding C++ object. A peer object provides object identity, which allows you to invoke a method on the corresponding C++ object. You can think of a peer object as a handle to a C++ object.

Each peer object identifies exactly one C++ object. Multiple peer objects can represent the same C++ object. The same C++ object has multiple Java peers only when the peers refer to members or base classes of the C++ object. A peer object is an instance of a Java peer class.

Peer objects can act as proxies for transient C++ objects or persistent C++ objects. In the case of persistent C++ objects, the

C++ code must be written to allocate objects in the usual way. For example, it must create new objects with calls to constructors that use ObjectStore's persistent **new**. As always, you must specify a location in which to allocate the new object. Unlike primary objects, peer objects are not subject to persistence through reachability.

Allocation and construction of new Java peer objects result in corresponding allocation and construction of C++ objects. If you call the **destroy()** method on a Java peer object, it causes the deletion of the corresponding C++ object. The typical way to access peer objects is the same as the way to access primary objects:

1 Open a database.

2 Obtain a database root.

3 Invoke methods on the root to traverse references to other objects.

To update the C++ object identified by a Java peer object, use the Java peer object methods. This propagates the changes to the corresponding C++ object.

Peer objects can become stale in the same way that primary objects can become stale. As you would expect, you cannot access stale peer objects.

You cannot extend Java peer classes. If you try to do so, the postprocessor displays an error such as the following:

```
Error:  Class COM.odi.omji.osmm.osmmSimpleType extends
COM.odi.jcpp.CPlusPlus but Java peer classes may not be
postprocessed or manually extended.
```

Peer pointer objects

A *peer pointer object* is a special kind of peer object. A peer pointer object is a Java object that encapsulates a location that is associated with a C++ object in a database. In other words, it is a pointer to a C++ object in a database.

The primary purpose of a peer pointer object is to transmit an address across the Java / C++ interface. Peer pointer classes extend **COM.odi.jcpp.PeerPointer**. ObjectStore also uses peer pointer classes to represent pointers to, references to, and arrays of nonclass objects.

Peer pointer classes do not have public methods. You cannot test instances of peer pointer classes for identity with the **==** operator. Use the **Object.equals()** method.

Peer pointers do not maintain identity. Multiple peer pointers can refer to the same address.

Using peer pointers

A peer pointer object never refers to data in the Java heap. It only refers to data in the C++ heap or in a C++ database. For example, an object of class **intP** represents a C++ object of type **int***. Pointers are not a part of Java, so the **int*** value typically originates in C++. When you have a C++ function that returns an object of type **int***, the Java peer method for that function returns an object of class **intP**.

Chapter 3, Writing the Application, on page 47 provides information about initializing peer pointer objects.

What Are Peer Classes?

A peer class has no state and no Java fields (instance variables) of its own.

The methods in a peer class are referred to as peer methods. Each method of a peer class is a stub method that corresponds to a C++ function member in the underlying C++ class. Peer methods provide the application-visible API for the peer class.

What peer objects do is sometimes called *method shipping* (or function shipping). When your application calls a peer method, the peer method

1 Converts its arguments from the Java to the C++ representation

2 Calls the corresponding C++ function member or accesses the C++ data member

3 Converts the result (if any) from the C++ representation to the Java representation

4 Returns the result, if there is one

For an object to be transmitted across the peer method boundary as an argument or return value, the object must be one of the following:

- Peer object
- Peer pointer object
- Persistent primary object

ObjectStore uses the **ClassInfo** class to maintain information about peer classes just as it maintains information about primary classes.

What Is the Interface Between Primary and Peer Objects?

You can store a reference to a peer object in a field of a primary object. If the primary object is persistent, or becomes persistent at the end of the transaction, it is your responsibility to ensure that the C++ object that the peer object identifies is persistent. Otherwise, the consequences are the same as storing a transient reference in an ObjectStore persistent object. If **check_illegal_pointers** is true, ObjectStore throws an exception.

You can pass a persistent primary object as an argument to a peer method. Persistent primary objects have a C++ representation that ObjectStore automatically generates. If you pass a transient primary object as an argument to a peer method, ObjectStore throws an exception as soon as the method is called. This is because a transient primary object has no underlying C++ representation, so it is impossible to translate it into a C++ pointer.

Identity

When the result of a method invocation is a C++ object, ObjectStore constructs a Java peer object (if the corresponding peer object does not already exist) and returns it. ObjectStore automatically handles object identity. If the same C++ object is returned to the Java run time more than once, ObjectStore detects that it has already handled this C++ object so it returns the same Java peer object. However, this does not apply to peer pointer objects.

How Are Peer Classes Defined?

ObjectStore provides a peer generator tool (**osjcgen**) to automatically write Java peer class definitions for specified C++ classes. The input to the tool includes an ObjectStore schema generated by the ObjectStore Schema Generator (**ossg**). For each class that you specify when you run the tool, the tool generates the items in the list that follows. The tool also generates these items for classes that are not explicitly specified if they are reachable from public data members, public function members, or public constructors of classes that are explicitly specified.

- Peer class definition.
- Unsafe peer class definition. An unsafe peer class provides static methods for operation on the corresponding peer class.
- **ClassInfo** subclass definition.
- C++ glue code that provides the interface between Java and C++ for the peer class.
- C++ glue code that provides the interface between Java and C++ for the unsafe peer class.
- Peer pointer class definition for each pointer type in the specified classes. For each peer pointer class that the tool defines, it also defines a corresponding unsafe peer pointer class and C++ glue code.

The tool defines all peer classes to extend the **COM.odi.jcpp.CPlusPlus** class itself or a class that directly or indirectly extends **COM.odi.jcpp.CPlusPlus**.

A peer class has the same inheritance structure as the corresponding C++ class.

What Is the Procedure for Using C++/Java Interoperability?

To use the C++ interoperability facility, follow the steps outlined below. Chapter 4, Building the Application, on page 65, describes these steps in more detail.

1 Define C++ classes.

2 Create an ObjectStore mapping schema database that contains the C++ classes. You create this schema with the ObjectStore Schema Generator (**ossg**) by specifying two special options. These options cause **ossg** to generate method interface information. See Generating the Mapping Schema Database on page 18.

3 Run the peer generator tool (**osjcgen**). The tool reads the mapping schema. Input to the tool and output from the tool are described in Chapter 2, Generating Peer Classes, on page 17.

4 Define Java classes that use the peer classes.

5 Run the Java compiler on your Java source files and the **.java** files generated by the tool. You should compile them all together to ensure that they are consistent with one another.

6 Run the **javah** tool on the compiled Java peer classes to generate headers.

7 Run the C++ compiler on the glue code generated by the peer generator tool.

8 Create the application schema database for the library used by your application.

9 Create the library that contains the application code, the glue code.

10 Postprocess the batch of **.class** files that result from compiling your Java source files in step 5. You do not need to postprocess the **.class** files for the generated peer classes. However, when you run the postprocessor on your Java application classes, be sure that the compiled peer classes are in your class path in a directory other than the postprocessor destination directory.

Example of Accessing C++ Classes

Here is an example of an application that stores information about professional golfers. The example shows three pieces of code:

- C++ code from which an ObjectStore schema can be generated
- Java code that is generated by the peer generator tool from the C++ code
- Java code that uses the tool-generated Java code

C++ Code

Here is the initial C++ code used to generate an ObjectStore schema.

```
class ProGolfer {
private:
    char* name;
    int age;
    int earnings;
    Agent* agent;
public:
    ProGolfer(char* n, int a, int e, Agent* ag);
    const char* getName() { return name; }
    void setName(char* _name);
    int getAge() { return age; }
    void setAge(int _age);
    int getEarnings() { return earnings; }
    void setEarnings(int _earnings);
    const Agent *getAgent() { return agent; }
    void setAgent(Agent *_agent);
}

class Agent {
private:
    char* name;
    int commission;
public:
    Agent(char* n, int c);
    const char* getName() { return name; }
    void setName(char* _name);
    int getCommission() { return commission; }
    void setCommission(int _commission);
}
```

Java Code and C++ Code Generated by Tool

The peer generator tool generates ten files when it generates peer classes for the **Agent** and **ProGolfer** classes. These files are described in Overview of Tool Output on page 26.

- **Agent.java**
- **AgentClassInfo.java**
- **AgentU.java**
- **Agent.cc**
- **AgentU.cc**
- **ProGolfer.java**
- **ProGolferClassInfo.java**
- **ProGolferU.java**
- **ProGolfer.cc**
- **ProGolferU.cc**

Agent.java

Here is the code in **Agent.java**:

```
package MyPkg;
public class Agent extends COM.odi.jcpp.CPlusPlus{
 /* required constructor for subtype constructors
    and unsafe makeArray methods */
 public Agent() {}

 /**
  * member function Agent::getName  char const*() declared at
"Agent.hh":10
  */
 public COM.odi.jcpp.charP getName() {
  COM.odi.jcpp.PeerReturnValue retValue =
    COM.odi.jcpp.PeerReturnValue.getPeerReturnValue(this);
  synchronized(retValue.om) {
  ntv_getName(retValue, retValue.thisRef);
  byte[] tmp_ref = retValue.getLinearObjectReference();
  if (COM.odi.jcpp.Utilities.isNullRef(tmp_ref))
   return null;
  tmp_ref = (byte[])  tmp_ref.clone();
   return new COM.odi.jcpp.charP(tmp_ref);
  }
 }
 final private static native void ntv_getName(
  COM.odi.jcpp.PeerReturnValue retValue,
  byte[] thisRef);

 /**
```

```
   * member function Agent::setName  void(char*) declared at
"Agent.hh":11
   */
  public void setName(COM.odi.jcpp.charP arg0) {
    COM.odi.jcpp.PeerReturnValue retValue =
      COM.odi.jcpp.PeerReturnValue.getPeerReturnValue(this);
    synchronized(retValue.om) {
    byte[] arg0Ref = COM.odi.jcpp.Utilities.getLinearRef(arg0);
    ntv_setName(retValue, retValue.thisRef, arg0Ref);
    COM.odi.jcpp.Utilities.recycle(arg0Ref);
    retValue.getVoid();
    }
  }
  final private static native void ntv_setName(
    COM.odi.jcpp.PeerReturnValue retValue,
    byte[] thisRef,
    byte[] arg0Ref);

  /**
   * member function Agent::getCommision  int() declared at
"Agent.hh":12
   */
  public int getCommision() {
    COM.odi.jcpp.PeerReturnValue retValue =
      COM.odi.jcpp.PeerReturnValue.getPeerReturnValue(this);
    synchronized(retValue.om) {
    ntv_getCommision(retValue, retValue.thisRef);
    return retValue.getInt();
    }
  }
  final private static native void ntv_getCommision(
    COM.odi.jcpp.PeerReturnValue retValue,
    byte[] thisRef);

  /**
   * member function Agent::setCommision  void(int) declared at
"Agent.hh":13
   */
  public void setCommision(int arg0) {
    COM.odi.jcpp.PeerReturnValue retValue =
      COM.odi.jcpp.PeerReturnValue.getPeerReturnValue(this);
    synchronized(retValue.om) {
    ntv_setCommision(retValue, retValue.thisRef, arg0);
    retValue.getVoid();
    }
  }
  final private static native void ntv_setCommision(
    COM.odi.jcpp.PeerReturnValue retValue,
    byte[] thisRef,
    int arg0);
```

```
  /**
   * static member function Agent::Agent  Agent(char*,int) declared
at "Agent.hh":9
   */
  public Agent(COM.odi.Segment segId, COM.odi.jcpp.charP arg0,
int arg1) {
    super();
    COM.odi.jcpp.PeerReturnValue retValue =
      COM.odi.jcpp.PeerReturnValue.getPeerReturnValue();
    synchronized(retValue.om) {
    byte[] arg0Ref = COM.odi.jcpp.Utilities.getLinearRef(arg0);
    ntv_Agent(retValue, (segId == null) ? -4
:((COM.odi.imp.Segment)segId).getSegmentId(), (segId == null) ? -4
:((COM.odi.imp.Database)(segId.getDatabase())).getDatabaseId(),
arg0Ref, arg1);
    COM.odi.jcpp.Utilities.recycle(arg0Ref);
    retValue.noteAsPersistent(this);
    }
  }
  final private static native void ntv_Agent(
    COM.odi.jcpp.PeerReturnValue retValue,
    int segId,
    int dbId,
    byte[] arg0Ref,
    int arg1);

  /**
   * required delete method
   */
  public void delete() {
    COM.odi.jcpp.PeerReturnValue retValue =
      COM.odi.jcpp.PeerReturnValue.getPeerReturnValue(this);
    synchronized(retValue.om) {
    ntv_delete(retValue, retValue.thisRef);
    COM.odi.imp.ObjectManager.destroyPeer(this);
    }
  }
  final private static native void ntv_delete(
    COM.odi.jcpp.PeerReturnValue retValue,
    byte[] thisRef);
  static {
    /* create a classinfo and register it */
    COM.odi.ClassInfo.register(new AgentClassInfo());

COM.odi.jcpp.CPlusPlus.noteJavaCppClassAssociation("MyPkg.A
gent", "Agent");
  }
}
```

Agent.cc

Here is the code in **Agent.cc**:

```
JNIEXPORT void JNICALL Java_MyPkg_Agent_ntv_
1getName(JNIEnv* _jni_env, jclass,
  jobject retValue,
  jbyteArray thisRef)
{
  EXCEPTION_HANDLERS_START {
  Agent* thisP = (Agent*)
    JCPlusPlus::getNonNullPointer(_jni_env, thisRef);
  JCPlusPlus::setLinearObjectReference(_jni_env, retValue, thisP-
>getName());
  } EXCEPTION_HANDLERS_END;
}
JNIEXPORT void JNICALL Java_MyPkg_Agent_ntv_
1setName(JNIEnv* _jni_env, jclass,
  jobject retValue,
  jbyteArray thisRef,
  jbyteArray arg0)
{
  EXCEPTION_HANDLERS_START {
  Agent* thisP = (Agent*)
    JCPlusPlus::getNonNullPointer(_jni_env, thisRef);
  char* arg0Ptr = (char*)JCPlusPlus::getPointer(_jni_env, arg0);
  thisP->setName(arg0Ptr);
  JCPlusPlus::setVoid(_jni_env, retValue);
  } EXCEPTION_HANDLERS_END;
}
JNIEXPORT void JNICALL Java_MyPkg_Agent_ntv_
1getCommision(JNIEnv* _jni_env, jclass,
  jobject retValue,
  jbyteArray thisRef)
{
  EXCEPTION_HANDLERS_START {
  Agent* thisP = (Agent*)
    JCPlusPlus::getNonNullPointer(_jni_env, thisRef);
  JCPlusPlus::setInt(_jni_env, retValue, thisP->getCommision());
  } EXCEPTION_HANDLERS_END;
}
JNIEXPORT void JNICALL Java_MyPkg_Agent_ntv_
1setCommision(JNIEnv* _jni_env, jclass,
  jobject retValue,
  jbyteArray thisRef,
  jint arg0)
{
  EXCEPTION_HANDLERS_START {
  Agent* thisP = (Agent*)
    JCPlusPlus::getNonNullPointer(_jni_env, thisRef);
  thisP->setCommision((int)arg0);
  JCPlusPlus::setVoid(_jni_env, retValue);
  } EXCEPTION_HANDLERS_END;
}
```

```
JNIEXPORT void JNICALL Java_MyPkg_Agent_ntv_
1Agent(JNIEnv* _jni_env, jclass,
  jobject retValue,
  jint segId,
  jint dbId,
  jbyteArray arg0,
  jint arg1)
{
  EXCEPTION_HANDLERS_START {
  os_segment* segIdPtr = JCPlusPlus::getSegment(dbId,segId);
  char* arg0Ptr = (char*)JCPlusPlus::getPointer(_jni_env, arg0);
  os_typespec tmp_tspec("Agent");
  Agent* tmp_this = ::new(JCPlusPlus::getSegment(dbId, segId),
&tmp_tspec) Agent(arg0Ptr, (int)arg1);
  JCPlusPlus::setObject(_jni_env, retValue, tmp_this, "Agent");
  } EXCEPTION_HANDLERS_END;
}
JNIEXPORT void JNICALL Java_MyPkg_Agent_ntv_
1delete(JNIEnv* _jni_env, jclass,
  jobject retValue,
  jbyteArray thisRef)
{
  EXCEPTION_HANDLERS_START {
  Agent* thisP = (Agent*)
    JCPlusPlus::getPointer(_jni_env, thisRef);
  delete thisP;
  JCPlusPlus::setVoid(_jni_env, retValue);
  } EXCEPTION_HANDLERS_END;
}
_DMA_ClassBind MyPkg_Agent_class_bind("MyPkg.Agent",
"Agent");
```

Java Code That Uses Generated Code

Here is Java code that uses the code generated by the tool:

```
public
class Example2 {

    public static void main() {
        ObjectStore.initialize(null, null);
        Database db = Database.create("PGA.odb",
            ObjectStore.ALL_READ | ObjectStore.ALL_WRITE);

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

        // Create instances of Agent and ProGolfer (and corresponding
        // C++ objects). Note: the Java peers are transient,
        // but the corresponding C++ objects are allocated in the
        // specified segment.
        Segment seg = db.getDefaultSegment();
        Agent pat = new Agent(seg, "Pat", 15);
        ProGolfer tiger = new ProGolfer(seg, "Tiger", 20, 1000000, pat);

        // Create a root that points to the C++ object corresponding to
        // tiger
        db.createRoot("Tiger", tiger);

        // End the transaction. This stores the C++ objects
        // in the database.
        tr.commit();
        db.close();
    }
}
```

Chapter 2
Generating Peer Classes

The peer generator tool takes an ObjectStore C++ mapping schema database as input and defines peer classes that represent C++ classes. This chapter provides information and instructions for running the peer generator tool. It also provides information about how the tool defines Java peer classes.

Contents

This chapter discusses the following topics:

Description of Inputs to the Tool

The peer generator tool uses an ObjectStore C++ mapping schema database to define the Java peer classes.

About the Mapping Schema Database

A mapping schema database is a C++ application schema database that contains additional information that is needed to define peer methods. A mapping schema database

- Represents all member functions in the schema
- Uses name reachability to represent all classes that are reachable

Generating the Mapping Schema Database

To generate the mapping schema database, specify the following two options when you run **ossg**.

- **-store_member_functions** or **-smf**
- **-store_function_parameters** or **-sfp**

Specify these options in addition to any options you normally specify. These options were new in ObjectStore Release 5.0. Complete information for using the schema generator is in *ObjectStore Building C++ Interface Applications*.

The mapping schema database is not the application schema database that the C++ portion of your code uses. The mapping schema database contains a lot of information that the C++ portion does not need. The recommended procedure is to run **ossg** twice:

- Once to generate the mapping schema database for the generation of peer classes. After you run the peer generator tool, you no longer need this version of the schema. You can delete it.
- Once to generate the application schema database for the library you create for your Java application that accesses C++. This is the application schema database that you link into your executable. You need this version of the schema for as long as you need your application to run.

Details for running **ossg** are in Chapter 4, Building the Application.

Specifying Classes for Which to Generate Peer Class Definitions

When you invoke the peer generator tool, you specify the classes for which you want the tool to define peer classes. Specify the core set of C++ classes that must be mapped. The peer generator tool maps the classes in this set. It also maps classes not explicitly in the set if they are reachable from public data members, public function members, or public constructors of classes in the root set.

Running the Peer Generator Tool

To map C++ classes to Java peer classes, run the peer generator tool (**osjcgen**).

Description of Command Line Format

The command line format appears below. It shows the different components in the command line on different lines only for clarity.

osjcgen -package *destination_package*
-native_interface *interface_type*
-schema *mapping_schema_database*
-classdir *target_dir* **-libdir** *target_dir* [*options*] *class_list*

-package *destination_package*
Identifies the package in which the peer generator creates peer classes. Required.

-native_interface *native_interface*
Specifies the virtual machine interface for which the tool should generate C++ glue code. Specify **jni** for the Sun VM or **ms_raw** for the Microsoft VM.

-schema *mapping_schema_database*
Identifies the mapping schema database from which the tool extracts type definitions. Required.

You must have specified the **-store_member_functions** (**-smf**) and **-store_function_parameters** (**-sfp**) options when you ran **ossg** to create this schema. If you did not, the tool cannot define peer methods. These options cause additional information to be stored in the schema database. **osjcgen** needs this information to correctly process member functions.

The mapping schema database you specify is not the application schema database that the application uses.

If the C++ code for which you want to define peer classes does not define methods, you need not specify the **-smf** and **-sfp** options when you run **ossg**.

-classdir *target_dir*
Specifies the directory relative to which the tool writes the peer classes and other generated Java files. For example, if the specified *destination_package* is **MyPkg** and the specified *target_dir* is **MyDir**, the tool creates the peer classes in **MyDir/MyPkg**. The directory you specify must exist. If the directory you specify does not contain a subdirectory with the name of the package, the tool creates this subdirectory. Required.

-libdir *target_dir*
: Specifies the directory in which the tool creates the C++ files. In the directory you specify, the tool creates a directory that has the name you specify for *destination_package*. The tool places the C++ files in this package subdirectory of the directory you specify with the **-libdir** option. The directory you specify must exist. Required.

options
: Any of the options described in Description of Additional Options on page 21. Optional.

class_list
: Lists the names of the classes and structs for which you want the tool to generate peer classes. The tool places each peer class in the specified destination package. You can intersperse class name specifications with the options described in Description of Additional Options on page 21. Required.

  Do not specify **enum**s. If a peer method accepts or returns an **enum**, the tool automatically generates the peer class for the **enum**.

Description of Additional Options

Here are descriptions of the additional options you can specify on the **osjcgen** command line. These options can precede, follow, or be interspersed with the required **osjcgen** options.

-boolean *typedef_name*
: Indicates that the specified typedef name identifies a **boolean** data type. You can specify this option multiple times.

-full
: Turns off the **-leaf** option. The tool generates complete peer classes for types specified subsequently. The **-full** option is in effect by default.

@*input_file*
: You can use the @*input_file* option to specify a file that contains arguments for the peer generator tool. The tool inserts the contents of the specified file in the command line before it begins to execute the command line. You can specify this option multiple times. An input file cannot itself include the @*input_file* option. If it does, the tool treats it as the name of a class, which is not found.

-leaf
: Instructs the tool to generate a minimal definition for the peer classes of types specified after this option and before any **-full** option. When the tool generates a leaf peer class, the application cannot access C++ data or function members or any C++ base classes. The tool defines the Java peer class to inherit directly from **CPlusPlus**. It does not copy the inheritance structure from the C++ class. This option is useful when you want to prune a type graph to reduce the size of the interface code.

-map *C++_name Java_peer_name*

Allows you to specify the name of the Java peer class that you want to identify a particular C++ class. When you specify this option, you need not rely on the default name mapping rules. This option is particularly useful for naming template classes. You can specify the **-map** option as many times as you need to. Follow each specification with

1 The name of a C++ class. You must also specify the name of the C++ class in the list of classes for which you want to generate peer classes.

2 The name of the Java peer class that you want to represent the C++ class. This can be a fully qualified class name or an unqualified class name. If it is unqualified, the peer generator tool places the Java peer class in the package you specify on the command line.

-map_existing *C++_name Java_peer_name*

Allows you to specify the name of an existing Java peer class that you want to identify a particular C++ class. This option is the same as the **-map** option, except that the Java peer class already exists or will exist. Consequently, the peer generator does not generate any code for the specified Java peer class. In a separate run of the peer generator, or manually, you must create the Java peer type. See the description of **-map** for more details.

-nosynchronize

Turns off automatic synchronization of peer methods. If you specify this option, your application is responsible for ensuring that only one thread at a time per session is accessing ObjectStore. Failure to prevent concurrent access to the Java interface to ObjectStore and peer method entry points can cause the Java interface to ObjectStore to fail.

-oldtemplates

Causes the peer generator to map some C++ characters to Java equivalents used in previous releases of ObjectStore. Rules for Template Name Flattening on page 35 shows which C++ characters are invalid in Java and which Java equivalents they are mapped to.

-synchronize

Turns on automatic synchronization of peer methods. This is the default.

-suppress
package.*class*.*method*

Suppresses generation of the specified peer method. You should not need to specify this option frequently. However, if generated code for a particular method causes a problem for the compiler, this option allows you to prevent generation of that code.

You must specify the package name, the class name, and the method name.

It is expected that you would use this option in the following manner:

1 Run **osjcgen** without specifying this option.

2 Try to compile the generated code. A particular peer method causes a problem.

3 Run **osjcgen** again and specify the same classes, but suppress generation of the peer method that causes the problem.

4 Determine how to work around the lack of the suppressed peer method.

If it is not possible to work around the method that is causing the problem, you must redefine the C++ method into a form that does not cause a problem when it is mapped to a peer method.

Example of Running the Peer Generator Tool

The following example generates a Java peer class for the **CPerson** class. If it does not already exist, the tool creates the **MyPkg** subdirectory in the **MyDir** directory. The tool puts the Java peer class in the **MyPkg** package/subdirectory. The mapping schema database that the tool uses to generate the peer class is **CPerson.jadb**. The tool places the generated C++ files (and the generated Java files) in the **MyDir/MyPkg** directory.

```
osjcgen -package MyPkg
   -native_interface jni
   -schema CPerson.jadb
   -classdir MyDir
   -libdir MyDir
CPerson
```

If you specify **-libdir SomeOtherDir**, the tool places the generated Java files in **MyDir/MyPkg** and the generated C++ files in **SomeOtherDir/MyPkg**.

Caution

If the tool generates files that have the same pathnames as your C++ source files, the tool overwrites the C++ source files without warning you. This can happen if you create your C++ source files in the package and directory that you specify with the **-package**, **-classdir**, and/or **-libdir** options to **osjcgen**.

For example, suppose you have the **CPerson.cc** C++ file in the **/MyDir/MyPkg** directory. When you run **osjcgen**, you specify the following on one line:

```
osjcgen -package MyPkg
-native_inteface jni
 -schema CPerson.jadb
 -classdir MyDir
 -libdir MyDir
CPerson
```

The peer generator tool generates these files:

```
/MyDir/MyPkg/CPerson.java
/MyDir/MyPkg/CPersonU.java
/MyDir/MyPkg/CPersonClassInfo.java
/MyDir/MyPkg/CPerson.cc
/MyDir/MyPkg/CPersonU.cc
```

An explanation of these files appears in Overview of Tool Output 26. But as you can see, the tool would overwrite your **CPerson.cc** source file. On Solaris, you can work around this by specifying

.CC instead of **.cc** in the C++ file name. On Windows, you can work around this by specifying **.cpp** instead of **.cc** in the C++ file name. Alternatively, you can either specify different directories for **-classdir** and / or **-libdir,** or you must not create C++ source files in the Java package subdirectory.

Overview of Tool Output

For each class specified in the **osjcgen** command line, the tool generates a Java peer class in the destination package. The tool also generates Java peer classes for classes not explicitly specified if they are reachable from public data members, public function members, or public constructors of classes that are explicitly specified.

For each Java peer class, the tool creates these files:

- A **.java** file that contains the standard Java peer class definition.
- Another **.java** file that contains the corresponding unsafe peer class definition. If the class being defined is a leaf class, the tool does not generate this file.
- A third **.java** file that contains the definition of the **ClassInfo** subclass for the peer class. This allows ObjectStore to manage instances of the peer class in the same way that it manages instances of primary objects. If the class being defined is a leaf class, the tool does not generate this file.
- A **.cc** file that contains the C++ glue code that ObjectStore uses as the interface between the Java peer methods in the safe class and the corresponding C++ methods. This is the C++ side of the peer class. Code in this file calls the C++ member functions.
- A **.cc** file that contains the C++ glue code that ObjectStore uses as the interface between the Java peer methods in the unsafe class and the corresponding C++ methods.
- If there are any pointer types in the specified class, the tool creates files for peer pointer classes. For each peer pointer class, the tool generates a
  - **.java** file that contains the peer pointer class definition
  - **.java** file that contains the peer pointer unsafe class definition
  - **.cc** file that contains the C++ glue code

For each enumeration, the tool creates only the **.java** file that contains the peer class definition that represents the **enum**. There is no C++ glue code for enumerations. See How Enums Are Mapped on page 36.

About Java Peer Class Definitions

When the peer generator tool generates a peer class, the definition includes

- A peer method for each public member function defined in the C++ class.
- Accessor functions that obtain and set each public data member in the C++ class. For array data members and for **const** data members, the **set** accessor is not generated.
- A constructor for each public constructor in the C++ class.
- A definition of a constructor with no arguments.

About Java Peer Pointer and Unsafe Class Definitions

The peer generator tool produces peer pointer classes and unsafe classes for classes you specify.

The pointer classes for the primitive types are part of the **jcpp** package. Multiple Java peer libraries can use the peer pointer classes. The peer pointer classes also have unsafe versions but they do not have corresponding **ClassInfo** subclasses.

The tool automatically generates an unsafe class for any peer class it defines. An unsafe class is an abstract class that is associated with a particular peer class. An unsafe class contains static methods for unsafe operations on instances of the associated peer class. Unsafe operations include array access and casting. The peer generator names the unsafe class by appending a **U** to the peer class name.

The peer generator tool defines a new overloading of the static method **makeArray()** in each unsafe peer class definition. This new overloading adds an argument to the **makeArray()** method signature. The additional argument is a clustering argument. For example, for the **Foo** class, the corresponding **FooU** unsafe class would contain the following array creation method:

```
public static Foo makeArray(COM.odi.Segment s, int n);
```

About C++ Output

The C++ output consists of the definition of the C++ side of the peer method. These are the **.cc** files that the tool generates. There is one **.cc** file for each of the following:

- Safe class definition
- Unsafe class definition
- Peer pointer class definition
- Peer pointer unsafe class definition

The **.cc** file contains C++ glue functions. The C++ glue function is the function that invokes the application's C++ member function corresponding to the peer method that was called. The typical call chain is as follows:

1 The Java application contains a call to a Java peer method.

2 The Java peer method calls the corresponding C++ glue function.

3 The glue function calls the corresponding C++ function member.

You must include (**#include**) the glue functions in one or more C++ source files that contain appropriate header file inclusion and class ordering.

How Classes and Structs Are Mapped

The peer generator tool generates Java peer classes for C++ classes and structs. The tool defines the generated peer class to inherit from **COM.odi.jcpp.CPlusPlus** either directly or indirectly.

Generated Names

The name of the generated Java peer class is the same as the C++ class name. However, for template class names, the tool must perform some conversions. See Rules for Template Name Flattening on page 35. Also, you can specify the **-map** option when you run the peer generator tool. This option allows you to specify a particular Java peer class name for a particular C++ class.

If a type **X** is defined in the scope of a class or namespace **Y**, the package in which the tool places the Java peer class for **X** has an additional name level. The package name for **X** is the destination package specified on the **osjcgen** command line with an additional package element equal to the name of the enclosing class or namespace followed by **_Pkg**..

When the peer generator tool generates a peer class for a nested type, it usually also generates a peer class for the enclosing type. In this case, the tool appends an underscore to the enclosing type name when it defines the Java peer name of the nested type.

Here are some examples of generated names when the value of *destination_package* is **MyPkg**.

C++ Name	***Java Name***
struct A	**MyPkg.A**
class B	**MyPkg.B**
class A::D	**MyPkg.A_Pkg.D**

Methods in Java Peer Classes

The Java peer method is the method that your ObjectStore application actually calls.

Minimum methods defined

The minimum methods provided in each Java peer class include

- Public constructors if the class is not an abstract class. The tool modifies each constructor's argument list by adding a segment specification as the first argument.
- A default constructor without arguments for use by subclass constructors. ObjectStore invokes this constructor from peer subclasses to ensure the initialization of superclasses. The tool does not generate this constructor if this peer class is a **final** class.
- Accessor methods that obtain and set each public data member in the corresponding C++ class. For example, for a field **X** of type **int**, the accessor methods would be
  - **int X()**
  - **void X(int)**
- Peer methods that correspond to public C++ functions. There might be fewer peer methods than C++ member functions if there are function overloadings.

Public constructors

The tool generates public constructors for the public C++ constructors in the C++ classes. In addition, nonabstract classes have constructors defined for instantiation purposes. In each Java public contructor that it generates, the tool inserts a new first parameter. This parameter takes **COM.odi.Segment** as an argument.

Private native peer methods

For each Java public peer method that the tool defines, it also defines a private native peer method.

The name of the private native peer method is the name of the public peer method with a prefix of **ntv_**. These private methods are for use only by the public peer methods.

Examples

Here is a peer method with a primitive return value:

```
int getId() {
    PeerReturnValue result =
        PeerReturnValue.getPeerReturnValue(this);
    ntv_getId(result);
    return result.getInt();
}
```

Here is an example of a peer method with a primitive argument:

```
void setId(int id) {
    PeerReturnValue result =
        PeerReturnValue.getPeerReturnValue(this);
    ntv_setId(result, id);
    result.getVoid();
}
```

Unsafe Peer Classes

Your application never instantiates unsafe peer classes. These classes provide static methods for operating on safe peer classes. In each unsafe peer class, the following public methods might be generated by the tool, where **X** is the name of the class.

- **static X makeArray(Segment, int); // create array of X**

  The tool generates this peer method if the corresponding safe peer class is not abstract and has a public constructor that takes no arguments.

- **static void deleteArray(X); // delete array of X**

  The tool generates this peer method if the corresponding safe peer class is not abstract and has a public destructor.

- **static X ref(X, int); // index into array of X**

  The tool generates this peer method if the corresponding safe peer class is not abstract.

- **static void set(X, int, X); // assign to an element of an X array**

  The tool generates this peer method if the class can be assigned to.

- **static X cast(COM.odi.jcpp.CPlusPlus);**

  The tool always generates this peer method.

The tool defines these methods in unsafe peer classes in addition to the methods described in the previous section.

Inheritance in Classes and Structs

Suppose a C++ class or struct (**X**) directly inherits from a single nonvirtual base class (**Y**). The tool defines the Java peer class for **X** so it inherits directly from the Java peer class for **Y**.

If a C++ class or struct does not inherit from any class, the corresponding Java peer class inherits directly from **COM.odi.jcpp.CPlusPlus**.

Multiple inheritance

Suppose a C++ class or struct (**X**) directly inherits from multiple base classes (**Y** and **Z**) either virtually or nonvirtually. The tool defines the Java peer class for **X** so it

- Inherits directly from **COM.odi.jcpp.CPlusPlus**
- Implements the interfaces **YI** and **ZI**

The **YI** and **ZI** interfaces provide the ability to convert between an **X** instance and its equivalent **Y** or **Z** instance, respectively.

Single virtual base class

Now, suppose a C++ class or struct (**X**) directly inherits from a single virtual base class (**Y**). The tool defines the Java peer class for **X** so it directly inherits from **COM.odi.jcpp.CPlusPlus** and implements the **YI** interface.

Single and multiple inheritance

It is possible for a class (**A**) to be the only inherited class for one class and to also be one of several inherited classes for some other class. For example, you might have these classes:

```
class A { int a; };
class B { int b; };
class C : A { };
class D : A, B { };
```

For these classes, the tool defines the following peer classes:

```
class A extends CPlusPlus {}
class B extends CPlusPlus {}
class C extends A {}
class D extends CPlusPlus implements AI, BI {
    A toA() {...} B toB() {...} }
interface AI { A toA(); }
interface BI { B toB(); }
```

The implementation of **D.toA()** returns a reference to the direct **A** superclass. **D.toB()** returns a **B** instance, which can be trivially cast to **A** to get the **A** superclass of **B**.

Unions

Unions are not supported in this release.

Handling of C++ Data Members

- For each public data member defined in a C++ type, the tool defines two Java peer methods. For a data member named **M**, the tool defines
- **M()** —This method obtains the value of the C++ data member.
- **M(***type_of_M***)** —This method sets the value of the C++ data member.

If a C++ data member is static, the tool defines these methods as static methods.

Handling of C++ Function Members

For some C++ function members, the tool does not define a corresponding Java peer method. This typically happens when the tool determines that another C++ function member of the same name is more desirable. The tool uses the algorithm below to pick the C++ function member for which to define a Java peer method.

1 A **const** method has precedence over a non-**const** method.

2 Formal parameters of type **X*** have precedence over **X&** and **X**.

3 Formal parameters of type **X&** have precedence over **X**.

4 **const** parameters have precedence over non-**const** parameters.

5 If no method is selected as most desirable based on the criteria above, the peer generator tool selects the method that appears earliest in the class declaration.

In the Java peer method definition, the tool inserts a comment that states which C++ function member corresponds to the peer method.

Handling of Operators

Normally, the name of the Java peer method is the same as the name of the corresponding C++ function member. For user-defined operators, the peer generator tool uses a standard translation process to convert the C++ operator name to a Java operator name. The following table shows the standard translations.

C++ Operator	*Java Operator*	*C++ Operator*	*Java Operator*
=	set	|=	orEq
+	plus	<<	out
-	minus	>>	in
* (binary)	times	<<=	shiftLeftEq
* (unary)	get	>>=	shiftRightEq
/	divide	==	equals
%	modulo	!=	notEquals
^	xor	<	isLess
++	next	>	isGreater
--	prev	<=	isLessEq
&	and	>=	isGreaterEq
|	or	&&	logAnd
~	not	||	logOr
+=	plusEq	!	logNot
-=	minusEq	()	call
*=	timesEq	[]	ref
/=	divideEq	->	deref
%=	moduloEq	,	comma
^=	xorEq	delete	opDelete
&=	andEq	new	opNew

Variable argument lists

Variable argument lists are not supported.

Rules for Template Name Flattening

There are characters that are not valid in Java, for example, some characters in C++ template class names. When a class name contains such characters, you can specify the **-map** option when you run the peer generator tool. This option allows you to explicitly map the name of a C++ class to a Java class name that you define.

If you do not use the **-map** option, the tool converts characters that are invalid in Java as shown in the following table.

C++ Character	*Java Equivalent*
<	_
>	_
:	_
,	_
*****	**P**
&	**R**
A blank space	Nothing

For example, suppose you have

foo<A::B*, int>

The tool maps this name to

foo_A__BP_int_

In previous releases, some C++ characters were mapped to different Java equivalents. If you want to use the old values, specify **-oldtemplates** when you run the peer generator.

To specify a template class when you run **osjcgen**, specify the template name with white space removed. If the template name requires white space, you must enclose it in quotation marks when you specify it on the command line. For example, suppose the name of the template class is **foo<const char*>**. Specify it as **"foo<const char*>"**.

How Enums Are Mapped

The peer generator tool translates **enum** types to Java peer classes that have a public static field for each enumerator in the enumeration. For example, here is some C++ code:

```
enum osmmResultEnum { OSMM_SUCCESS = 0,
    OSMM_FAILURE = 1};
```

The tool translates this to the following:

```
import java.util.Hashtable;
public class osmmResultEnum {
    private static Hashtable enumerators = new Hashtable();
    private int myValue;
    private osmmResultEnum(int value, boolean register) {
        myValue = value;
        if (register)
            enumerators.put(new Integer(value), this);
    }
    public static osmmResultEnum OSMM_SUCCESS =
        new osmmResultEnum(0, true);
    public static osmmResultEnum OSMM_FAILURE =
        new osmmResultEnum(1, true);
    public int intValue() {
        return myValue;
    }
    public static osmmResultEnum cast(int value) {
        osmmResultEnum e = (osmmResultEnum)
            enumerators.get(new Integer(value));
        if (e == null) e = new osmmResultEnum(value, false);
        return e;
    }
}
```

Here is an example of how to use the peer class:

```
osmmResultEnum myres1 = getData();
    if( myres1 == osmmResultEnum.OSMM_SUCCESS)
        {
        .
        .
        .
```

You could also use the peer class this way:

```
if( myres1.intValue() > 0 )
    return;
```

This representation of **enum**s diverges from the TwinPeaks model in that the tool does not generate any C++ glue code to support the representation. The enumeration peer classes

- Do not inherit from **COM.odi.jcpp.CPlusPlus**. They inherit from **Object**.
- Do not have unsafe peer classes.
- Do not have methods by which instances can be allocated in C++.

How Primitive Types Are Mapped

The peer generator tool maps C++ primitive types to Java types as described in the following table.

C++ Primitive Type	*Java Primitive Type*
void	**void**
bool	**boolean**
user boolean	**boolean**
char	**char**
signed char	**byte**
unsigned_char	**byte**
wchar_t	**char**
short	**short**
unsigned short	**short**
int	**int**
unsigned int	**int**
long	**int** (except on Digital UNIX)
unsigned long	**int** (except on Digital UNIX)
long	**long** (Digital UNIX only)
unsigned long	**long** (Digital UNIX only)
unsigned long long	**long** (Solaris only)
__int64	**long** (Windows NT only)
unsigned__int64	**long** (Windows NT only)
float	**float**
double	**double**
long double	**double**

The C++ primitive type **long long** is not yet supported.

How Pointers to Nonclass Types Are Mapped

The peer generator tool maps a pointer to class **X** in C++ to a reference to the corresponding Java peer class **X**. Pointers to other types of data in C++ do not have such useful representations in Java.

When the tool generates representations of pointers to nonclass types, it defines a class and an associated unsafe class, which can be used for many of the purposes that C++ pointers can be used for.

Pointers to Primitives

When the tool recognizes a C++ pointer to a primitive type, it maps the type to a Java peer class, as shown in the following table.

C++ Pointer	*Java Mapping*
void*	**COM.odi.jcpp.voidP**
char*	**COM.odi.jcpp.charP**
signed char*	**COM.odi.jcpp.signed_charP**
unsigned char*	**COM.odi.jcpp.unsigned_charP**
short*	**COM.odi.jcpp.shortP**
unsigned short*	**COM.odi.jcpp.unsigned_shortP**
int*	**COM.odi.jcpp.intP**
unsigned*	**COM.odi.jcpp.unsigned_intP**
long*	**COM.odi.jcpp.longP**
unsigned long*	**COM.odi.jcpp.unsigned_longP**
float*	**COM.odi.jcpp.floatP**
double*	**COM.odi.jcpp.doubleP**

Along with each of these **COM.odi.jcpp** pointer classes is a corresponding unsafe class.

Pointers to other primitive types

For pointers to primitive types other than those listed above, the tool uses the **COM.odi.jcpp.voidP** class. This includes

- **long double***
- **long long***
- **unsigned long long***
- **wchar_t***
- **__int64***
- **unsigned__int64***
- **bool***

Pointers to Pointer Types

For pointers to a pointer type, such as **foo**** and **int***, the peer generator tool creates a Java peer class as usual. The tool constructs the name of this peer class as follows:

- The first part of the name is the name of the Java peer class that represents the type of the ultimate target of the pointer.
- The tool appends a **P** to that name for each level of indirection.

For example, the Java peer class for **int*** would be **intPPP**.

For a C++ class type **foo**, there can be the **foo** and **fooPP** (and **fooPPP** and so on) Java peer classes. The tool never generates a **fooP** peer pointer class.

These Java peer classes define only a single constructor, which constructs a pointer from a *linear object reference*. A linear object reference is always a byte array that contains the information that describes the location and type of some object in C++.

For each pointer-to-pointer Java peer class, the tool generates a corresponding unsafe class. For example, the tool might generate **fooPP** and **fooPPU**.

Representation of char* Types

Although the C++ **char*** data type and the Java **String** data type both represent character string data in their respective languages, their characteristics are too dissimilar to treat them as being equivalent. For example, the Java **String** data type

- Is not null terminated
- Uses the unicode character set

The C++ **char*** data type represents a character string that

- Is null terminated
- Might use a multibyte encoding for non-ASCII characters

In addition, the C++ **char*** data type is sometimes used for nonstring uses when the data that is referred to might not be a null-terminated sequence of characters.

By defatult, **char*** is represented as **COM.odi.jcpp.charP**. However, it is possible to have the peer generator tool perform conversion automatically for each of the following:

- **String -> char***
- **char* -> String**

When used as the type of a C++ function parameter, the **char*** string is allocated transiently on the heap.

There are several **typedef**s defined in **JCPlusPlus.hh** that the peer generator tool interprets in a special way. They are

- **typedef char* OS_EUCJIS_STR;**
- **typedef char* OS_EUCJIS_TMPSTR;**
- **typedef char* OS_EUCJISP;**
- **typedef char* OS_SJIS_STR;**
- **typedef char* OS_SJIS_TMPSTR;**
- **typedef char* OS_SJISP;**
- **typedef char* OS_STR;**
- **typedef char* OS_TMP_STR;**

For each of the **TMP** versions, the **char*** pointer is automatically deleted after return from the C++ function. For the non-**TMP** versions, the **char*** is not deleted, leaving the responsibility for deallocating to the called code. When used as the return type of a C++ function, the **TMP** versions automatically delete the **char*** after the conversion to **String** is completed, whereas the non-**TMP** versions assume that the **char*** string should not be deleted.

When **OS_STR** appears in a C++ definition, the tool defines the Java peer methods that manipulate the data to use **String** instead of **COM.odi.jcpp.charP**. The interpretation of the **char*** data will be determined by the current default **String** converter. See Using the charP Class on page 60.

When **OS_EUCJIS_STR** or **OS_SJIS_STR** is in a C++ definition, the peer generator defines the Java peer method that manipulates the data to use **String** instead of **COM.odi.jcpp.charP**. The implied character encoding is **EUCJIS** or **SJIS**, respectively.

Converting Strings

The abstract class, **COM.odi.jcpp.CharConverter**, defines the protocol for converting Java strings to and from C++ byte arrays. You can define subclasses of this class as needed. ObjectStore includes the following subclasses:

- **COM.odi.jcpp.StdCharConverter** uses the character conversion capabilities of the JDK **java.io** package.
- **COM.odi.jcpp.CppCharConverter** uses the Object Design native C++ Japanese string conversion library.
- **COM.odi.jcpp.UTF8CharConverter** uses the Java variant of UTF-8 character encoding.

How Reference Types Are Mapped

For a C++ reference type, the tool generates a Java peer class in the same way it would if it were for a C++ pointer type. Just as the tool maps **foo*** to **foo**, so it maps **foo&** to **foo**. The reason for this is that in Java, **foo** implies a pointer to an instance of **foo**. So **foo*** in C++ is semantically equivalent to **foo** in Java. For example, this is certainly true when **foo*** provides the type of a field. Here are examples of mapping reference types to peer classes:

C++ Reference Type	***Java Peer Class Name***
foo&	**foo**
int&	**intP**
char*&	**charPP**

How Embedded Class Type Return Values Are Mapped

When a C++ function member is defined to return a struct or class on the stack, the generated glue code transiently allocates an instance of the type and initializes it with the return value of the function. The Java peer method returns a reference to that transiently allocated instance and it is the responsibility of the caller to delete the storage when done with it. The tool inserts a comment in the generated Java code to this effect.

Restrictions

There are some restrictions with regard to peer classes.

Code for Which Peer Classes Cannot Be Generated

The peer generator tool cannot generate peer classes for C++ code that includes

- Friend functions
- Global functions and variables
- Function pointers
- Pointers to members
- Unions

Overloaded Functions

The peer generator tool might not be able to generate peer methods for all overloaded C++ functions. If a class has multiple functions of the same name that are candidates for peer method generation, and two or more of these generated peer methods would have the same method arguments, the tool generates only one of the peer methods that would have duplicate arguments. The criteria for how ObjectStore selects the functions for which to provide peer methods are in Handling of C++ Function Members on page 33.

For example, the following table shows how the tool would map some C++ functions to Java peer methods.

C++ Function	*Potential Java Peer Method*
void foo(double)	**void foo(double)**
void foo(int)	**void foo(int)**
void foo(long)	**void foo(int)**

The first overload, **foo(double)**, is distinguishable from the other two, so the tool generates the corresponding peer method. The other two functions map to identical Java method signatures. ObjectStore determines that each of the corresponding C++ functions is equally desirable according to the function selection criteria. Consequently, the tool maps the function whose definition appears first in the class definition.

Chapter 3
Writing the Application

After you run the peer generator tool, you can use the defined peer classes in your Java application. This chapter provides information about how to use peer classes in your application.

Contents

This chapter discusses the following topics:

Behavior of Peer Objects

A peer object identifies the associated C++ persistent or transient object. Peer objects exist in the Java VM as the result of one of these actions:

- Invocation of methods that return references to C++ objects
- Explicit creation of peer objects

Peer Objects Can Become Stale

A peer object that identifies a persistent C++ object can become stale when the application commits a transaction. Whether it becomes stale depends on whether or not the application retains any values after the transaction commits.

A peer object always becomes stale when the application

- Explicitly deletes it
- Evicts it without retaining anything
- Aborts a transaction in which the peer object identified a C++ object that was transiently allocated

As always, an application cannot use a stale object. An attempt to do so throws **ObjectException**.

If a peer object identifies a transient C++ object, the peer object does not become stale when the transaction commits.

What a Peer Object Can Represent

Each peer object represents one of the following:

- A top-level C++ object, that is, a C++ object that was explicitly allocated. This includes an object that was allocated in the transient segment.
- A base class of a derived class that uses multiple inheritance or virtual inheritance.
- An embedded data member object of some class type. (The data member is an instance of some other class. It is not a primitive type or a pointer type.)
- An element of an array of C++ objects of some class type. Each element of the array is represented by a distinct Java peer object. The array itself can be indexed by the index methods on the unsafe class instances for the class.

The last three possibilities in the previous list can introduce object aliases, which are multiple peer objects that correspond to a single C++ object allocation. If your application creates object aliases, you must be careful about trying to access peer objects that might have been deleted.

For example, suppose you have two peer objects named **A** and **B**. The C++ object identified by **B** is a data member of the C++ object identified by **A**. Both **A** and **B** identify the same C++ object. You call **delete()** on **A**. This effectively deletes **B** because the C++ object it corresponds to has been deleted. You cannot manipulate **B** in any meaningful way.

If you delete a peer object that is an array, ObjectStore deletes all peer objects that represent the elements of the array. However, the C++ / Java run-time system does not track subobject relationships. It is the responsibility of the application to ensure that this situation does not cause problems.

Initializing ObjectStore/Starting a Session

In calls that start a session, you must set the value of the **COM.odi.ObjectStoreLibrary** property to the name of the library you plan to create for your application. If you fail to specify this, you receive error messages at run time that claim that you did not mark types or that you have multiple schemas.

The calls that start a session are

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

Creating C++ Objects

To create a C++ object, specify a Java **new** expression with a constructor from the definition of the corresponding peer object.

When the peer generator tool defines a peer class, it adds an argument as the first argument in each constructor. This additional argument takes a specification of **COM.odi.Segment**. This argument is followed by the Java arguments that correspond to the arguments specified in the C++ class definition.

You use the constructors in Java **new** expressions to create persistent C++ objects. The **new** expression causes both the Java peer object and the persistent C++ object to exist. The location of the C++ persistent object is determined by the value specified for that first argument (**COM.odi.Segment**).

The value of the first argument in the constructor also determines whether the new object is transient or persistent. The C++ object remains transient or persistent until it is deleted. Peer objects are the same regardless of whether they identify persistent or transient objects.

It is possible to create peer objects that identify transient C++ objects. To do so, specify the transient segment as the location. To specify the transient segment, use the return value from **COM.odi.jcpp.CPlusPlus.getTransientSegment()**. For example:

```
charP charp = new charP(name, CPlusPlus.getTransientSegment());
```

This example creates

- A transient C++ **char*[]** object
- A peer pointer object that is an instance of the **COM.odi.jcpp.charP** class

When a transaction commits or aborts, there might be Java peer objects that identify transient C++ objects. ObjectStore retains these peer objects as well as the C++ objects they identify.

Deleting C++ Objects

To delete C++ objects, you can use **CPlusPlus.delete()** or **ObjectStore.destroy()**.

Using the CPlusPlus.delete() Method

When the peer generator tool defines a class, it always makes the class inherit from **COM.odi.jcpp.CPlusPlus**, either directly or indirectly. The **COM.odi.jcpp.CPlusPlus** class specifies a **delete()** method. Call this method to delete a C++ object.

Deleting a C++ object makes the corresponding peer object stale. Subsequent attempts to invoke methods on the deleted object in the same transaction throw ObjectException.

When you delete a C++ object, ensure that there are no objects with pointers or references to the deleted object. If an application tries to dereference a pointer or reference to a deleted C++ object, the results are unpredictable.

Also, you must not invoke **delete()** on an object that is not a top-level heap-allocated object. If you delete an embedded object, it can have unpredictable results.

Using the ObjectStore.destroy() Method

An alternative way to delete a C++ object is to call the **COM.odi.ObjectStore.destroy()** method on the corresponding peer object. The peer class implements a **destroy()** method so that the destructor for the object is invoked in C++. As a result, storage is reclaimed by the C++ destroy operation only after a Java garbage collection.

When you invoke **ObjectStore.destroy()** on a primary Java object, ObjectStore leaves a tombstone. If any objects try to access the destroyed object, the tombstone causes ObjectStore to throw **COM.odi.**ObjectNotFoundException. However, there is a bug that prevents ObjectStore from leaving a tombstone when you destroy a Java peer object. This will be fixed in a future release. For now, you must be careful that you do not destroy a Java peer object that is still referred to by another object and then try to use that reference. While doing so is always a mistake, in the current product there is no tombstone to flag the mistake for Java peer objects.

You can destroy Java peer objects that identify transient C++ objects when you are in a read transaction and when you are outside a transaction, as well as when you are in an update transaction.

If you invoke **ObjectStore.destroy()** on a Java peer pointer object, the method returns without destroying the object. Use the **deleteArray()** method of the appropriate unsafe peer pointer class if the peer pointer refers to a top-level allocation.

Invoking Peer Methods

When you write your application, you must

- Ensure that ObjectStore or your application synchronizes calls to peer methods
- Be aware of allowable arguments to and return values from peer methods
- Be aware of how ObjectStore handles return values from peer methods

Synchronizing Calls to Peer Methods

ObjectStore synchronizes calls from different threads to both peer and nonpeer methods. This ensures that only one thread at a time per session is accessing ObjectStore. If you want to, you can specify the **-nosynchronize** option when you run **osjcgen**. When you specify this option, ObjectStore does not perform this synchronization. However, failure to prevent concurrent access to the Java interface to ObjectStore and peer method entry points can cause ObjectStore to fail. If you do specify the **-nosynchronize** option, your application must synchronize calls to peer methods.

Allowable Arguments To and Return Values From Peer Methods

Objects of most types can be passed directly to and returned from a peer method. However, a primary Java object that is not persistent cannot be passed.

Handling Return Values from Peer Methods

When peer methods return values that are primitive values or pointers to class types, your application can handle the return values in the usual way.

When peer methods return values that are pointers to nonclass types, the values are instances of the appropriate pointer classes. For example, a return type of **int*** is represented as an instance of the **intP** class. A return value of **foo**** is specified by **fooPP**.

When peer methods return nonprimitive values, there are two issues of importance to consider:

- The identity (==) of Java peer objects
- Determination of the correct subclass for the instance of the Java peer object

Identity of peer objects

When a peer method returns an object, ObjectStore determines whether or not there is already an instance of the peer object that identifies the returned object. If there is such an instance, ObjectStore reuses it. If there is not such an instance, ObjectStore creates one and associates it with the returned persistent C++ object.

Identity is not maintained for peer pointers.

Determining the correct subclass for a peer object

When ObjectStore materializes an instance of a peer object, the peer object has the run-time type that corresponds to the run-time type of the C++ object that the peer object represents. For persistent C++ objects, this is a straightforward operation. ObjectStore always uses the stored type information associated with the persistent C++ type to determine the run-time type.

Return type of transient C++ objects

For transient C++ objects, however, ObjectStore must use the statically declared return type. This causes the caller to be unable to cast to a derived type.

Creating References to Peer Objects

There are different rules for cross-segment and cross-database references to Java peer objects and to ObjectStore collections (instances of **COM.odi.coll.Collection**), which are special kinds of Java peer objects. The reason for these rules is the benefit of keeping segments separate. The benefit is that ObjectStore administrative tools can operate on one segment without disturbing or delaying access to another segment.

References from Java Primary Objects

A Java primary object in one segment can refer to an ObjectStore collection object in another segment only if the collection object is exported. If you try to refer to an unexported collection object in another segment, ObjectStore throws **ObjectNotExportedException**. ObjectStore throws this exception when you try to commit the transaction or try to evict the object that contains the reference to the unexported object.

Except for instances of ObjectStore collections, you cannot export Java peer objects. If you try to, ObjectStore throws ObjectException. Consequently, you cannot create a reference from a Java primary object in one segment to a noncollection Java peer object in another segment. If you try to, ObjectStore throws ObjectNotExportedException.

References from ObjectStore Collections

Although you cannot export a Java peer object (except an ObjectStore collection), an ObjectStore collection in one segment can include a Java peer object that is in another segment.

An ObjectStore collection can also contain Java primary objects in other segments and ObjectStore collections in other segments. Such primary objects or collections must be exported. If they are not, ObjectStore throws ObjectNotExportedException when an attempt is made to insert them.

References from C++ Pointers

A C++ pointer in one segment can refer to a Java peer object, which might be an ObjectStore collection object, in another segment. If the object is a collection object and the collection object is not exported, you should specify during garbage collection that the collection object is a root. (Note that the C++ pointer is really pointing to a C++ object.)

A C++ pointer cannot refer to a Java primary object. This is always the rule. It does not matter whether or not the C++ pointer and the Java primary object are in the same segment. It is the responsibility of the application to enforce this restriction. If it is not enforced, the result might be database corruption when existing objects are exported or when the garbage collector or compactor is run.

If you use C++ to store a Java object in a segment, that Java object cannot be referred to by an object in another segment.

Summary of Cross-Segment and Cross-Database Rules

The following table reiterates the cross-segment and cross-database rules for references to Java peer objects. In this table, the reference source is in one segment and the reference target is in another segment.

Reference Source	***Java Primary Object Target***	***Collection Object Target (COM.odi.coll)***	***Java Peer Object Target***
Java primary object	Allowed. Must export target.	Allowed. Must export target.	Forbidden.
Collection element (**COM.odi.coll**)	Allowed. Must export target.	Allowed. Must export target.	Allowed.
C++ pointer	Forbidden.	Allowed.	Allowed.

Handling Exceptions

A function call might terminate because of an ObjectStore TIX exception or a C++ exception. When this happens, ObjectStore throws the corresponding Java exception. In all cases, ObjectStore tries to intercept TIX and C++ exceptions and throw an ObjectStore Java exception that does not cause the Java VM to exit.

For ObjectStore TIX exceptions, there is a clear mapping to ObjectStore Java exceptions.

For user-defined TIX exceptions, you can write handler functions. If invocation of a C++ member function by means of a Java peer method results in an unhandled TIX exception, ObjectStore intercepts it and invokes your handler function, or a default handler if you do not create your own.

When you write a handler function, it must map the TIX exception to a string that identifies the Java exception that ObjectStore should throw.

For user-defined C++ exceptions, it is not possible at run time to query the object thrown by C++ for its type. Consequently, ObjectStore throws a general Java exception that informs you that some C++ exception was thrown.

Initializing Peer Pointers

To initialize a peer pointer, you must obtain its value from the result of some other peer method call. For example, suppose you have a C++ class defined like this:

```
class data_block {
    private:
        int* my_storage;
    public:
        int* get_data_storage() { return my_storage; }
    ...
};
```

The Java peer class would contain a corresponding method:

```
public class data_block extends CPlusPlus {
    public intP get_data_storage() { ... }
    ...
}
```

A call to **data_block.get_data_storage()** returns a peer pointer object that contains the C++ address referred to by the **data_block.my_storage** field in the C++ class.

Using the **charP** Class

You must understand the conversion between C++ **char*** and Java **String** types. Unless you use the macros that perform conversions for you, it is your responsibility to convert these types.

To convert a Java **String** to a C++ **char** array, use the **charP(String, Segment)** constructor.

To convert a **charP** to a Java **String**, use **charP.toString()**.

The peer generator tool provides **charP** customizations for the following typedefs:

```
typedef char* OS_EUCJISP;
typedef char* OS_SJISP;
```

When the **OS_EUCJISP** typedef appears in a C++ definition, the Java peer method that manipulates the data is declared to use **COM.odi.jcpp.EUCJIScharP** instead of **COM.odi.jcpp.charP**.

When the **OS_SJISP** typedef appears in a C++ definition, the Java peer method that manipulates the data is declared to use **COM.odi.jcpp.SJIScharP** instead of **COM.odi.jcpp.charP**.

Specifying Peer Objects in Notifications

The object you specify when you construct a **Notification** object can be a Java peer object. When you specify a peer object, the peer object must identify a persisent C++ object. The C++ object cannot be in the transient segment or database.

When a session receives a notification for a Java peer object, the Java type of the peer object is based on the type of the outermost collocated peer object. This might not be the same as the Java type in the original notification. The received type and the original type can differ in situations where the original object is nested within a larger containing object, and collocated with the containing object.

For information about notifications, see the *ObjectStore Java API User Guide*, Chapter 11, Using the Notification Facility.

Restrictions When Using Peer Classes

When you are using Java peer classes, you must be aware of restrictions in the following areas.

Capability of C++ Functions

In your application, you cannot invoke Java peer methods that would ultimately call C++ functions that would

- Open or close a database
- Start or end a transaction

You must use the Java API to ObjectStore to perform these actions.

Inheritance from Java Peer Classes

You should not try to write new persistence-capable classes in Java that extend Java peer classes. Peer classes do not behave correctly when you try to make a persistent instance. This restriction stems from the following:

- All Java peer classes inherit from **COM.odi.jcpp.CPlusPlus**. Therefore, all classes that extend peer classes are by definition also peer classes.
- The postprocessor does not annotate peer classes.
- Peer classes cannot contain state.
- Peer classes cannot be made persistent by reachability.

Use of SegmentObjectEnumeration Objects

When your application invokes **Segment.getObjects()**, the returned enumeration includes ObjectStore collection objects, but it does not include any other Java peer objects.

Destruction of Segments

You should use a Java entry point to destroy a segment. Doing so ensures that you do not access objects in destroyed segments. For example, you might create an enumeration of the objects in a segment. If you then use a C++ entry point to destroy the segment, invocation of the **SegmentObjectEnumeration.nextElement()** or **SegmentObjectEnumeration.hasMoreElements()** method continues to return objects. After a while, ObjectStore stops returning objects and throws the correct SegmentNotFoundException. If you try to read or update the contents of the returned objects, ObjectStore throws ObjectNotFoundException.

C++ Pointers

C++ pointers to Java primary objects are not allowed. It does not matter whether the Java primary object is in the same segment as the C++ pointer or in a different segment.

Performing deepFetch() on a Peer Object

When you call **ObjectStore.deepFetch()** on a Java peer object, ObjectStore does not retrieve anything, since a peer object has no contents.

Retaining Collections

Peer objects, and therefore ObjectStore collection objects, have no data members and so, in Java, peer objects do not appear to be connected to other objects. Even if you explicitly iterate through the elements in a collection and then commit the transaction with **ObjectStore.RETAIN_READONLY**, you cannot access the collection outside a transaction. However, if the collection elements are not themselves peer objects or collections, you can manipulate them outside a transaction, but you cannot use the collection to access them. You must explicitly read them in the transaction, retain them, and then access them directly or through another nonpeer object.

Calling Java From C++

When accessing C++ from Java, a program might need to call back into Java from C++. When running on a platform that uses JNI, the C++ code must get the correct **JNIEnv*** value to pass to JNI calls. Calling **getJNIEnv()** returns the **JNIEnv*** that was passed to the C++ code from Java. This method is defined only on platforms that use the JNI interface. The signature is

static JNIEnv* JCPlusPlus::getJNIEnv()

Chapter 4
Building the Application

This chapter provides information and instructions for building an ObjectStore Java application that accesses C++ classes. See **COM.odi.demo.jcpp** for an example of building an application.

Contents

This chapter discusses the following topics:

Description of Files

When you build an ObjectStore Java application that accesses C++ classes, you must include certain C++ files and Java files.

C++ Files

The C++ portion of your application includes the following files:

- One or more header files that declare your C++ classes. For example, the **jcpp** demo contains **cprsn.hh**.
- One or more files that define methods for your classes. For example, the **jcpp** demo contains **cprsn.cc**.
- A schema source file that marks classes that you want to be persistent and / or one or more library schema databases. For example, in the **jcpp** demo, **schema.cc** marks the **CPerson** class.

When you compile these files, special options might be needed. This information is in Step 6: Compile C++ Glue Code on page 73.

A Java application that accesses C++ classes does not require the file that contains the **main** program for your C++ application.

Java Files

The Java portion of your application that is required for access to C++ includes several files.

Java source files

Create the files that define the Java classes that use the peer classes that represent your C++ classes. For example, the **jcpp** demo uses the **CPerson** Java peer class in the **Main.java** file.

In the call that starts a session, you must set the value of the **COM.odi.ObjectStoreLibrary** property to the name of the library you plan to create. You must specify a name that is acceptable to **SystemLoadLibrary()** and not an explicit path. The name should follow platform conventions for library names. If you fail to specify this, you receive error messages at run time that state that you did not mark types or that you have multiple schemas.

On Solaris, if the shared library is named, for example, **libCPerson.so**, set the property to **"CPerson"**. Also, ensure that the shared library is available in the **LD_LIBRARY_PATH** variable.

On Windows, if the DLL is named, for example, **CPerson.dll**, set the property to **"CPerson"**. Ensure that the DLL is available in the **PATH** variable.

For debuggable DLLs (used by **java_g** and **jdb**), the DLL should be named **libCPerson_g.so** on Solaris and **CPerson_g.dll** on Windows.

Glue file

Create one or more files that include the C++ glue code files that the peer generator tool creates. For example, the **jcpp** demo defines the **CPerson_glue.cc** file.

In the glue file, specify **#include** for the following files:

- C++ header files that declare the C++ classes. In the **jcpp** demo, this is **cprsn.hh**.
- A file that declares ObjectStore entry points for using Java and C++. This is **<ostore/dma/JCPlusPlus.hh>**.
- Java VM glue header files that declare native Java method for use by the Java VM to call native methods. Generate these files by compiling the Java class definition that the peer generator tool generates and then running the **javah** tool on the resulting classes. In the **jcpp** demo, **CPerson_stubs.h** is the Java VM glue header file.
- C++ files generated by the peer generator tool. These files contain definitions of the native C++ methods for the classes. These are the C++ glue code files that ObjectStore uses as the interface between the Java peer methods and the corresponding C++ methods. The tool stores these files in a directory determined by your specifications for the **-libdir** and **-package** options. In the **jcpp** demo, this file is **CPerson.cc**.

For example, in the **jcpp** demo, the contents of the **CPerson_glue.cc** file are

```
#include "cprsn.hh"
#include <ostore/dma/JCPlusPlus.hh>
#include "CPerson_stubs.h"
#include "CPerson.cc"
```

If your application uses the unsafe version of a peer class, you might also have, for example, **CPersonU_glue.cc,** or you might place **#include** statements for the unsafe class in the same file with the statement for the corresponding safe class. The glue file can contain **#include** statements for any number of classes.

Steps for Building the Application

When your C++ source files are ready, follow these steps to build an ObjectStore Java application:

1 Generate the mapping schema database for your C++ application.

2 Generate the Java peer classes and C++ glue code.

3 Write the Java classes that use the generated peer classes.

4 Compile the Java peer classes and Java source files.

5 Generate the C header.

6 Compile the C++ glue code.

7 Generate the application schema database for the library for your ObjectStore Java application.

8 Create the library. On Windows, you create a DLL. On Solaris, create a shared library.

Step 1: Generate the Mapping Schema Database for Your C++ Application

Run the schema generator (**ossg**) to create a mapping schema database. A mapping schema database is a special application schema database for the C++ classes you want to access. When you run **ossg** to generate a mapping schema database, you must specify two special options:

- **-store_member_functions** or **-smf**
- **-store_function_parameters** or **-sfp**

These options add information that is required by the peer generator tool. It is this additional information that makes the schema database a mapping schema database.

As input to the schema generator, specify a schema source file and/or one or more library schema databases.

Schema source file as input

If you specify a schema source file, pass the **-store_member_functions** and **-store_function_parameters** options to **ossg**.

If the input is a schema source file, the include options must specify the

- ObjectStore C++ **include** directory
- ObjectStore Java **include** directory
- Java **include** directory
- Java system-specific **include** directory

Library schema databases as input

If you specify one or more library schema databases, you must have specified the **-store_member_functions** and **-store_function_parameters** options to **ossg** when you created the library schemas. If you did not, the peer generator tool cannot correctly generate peer methods for classes in the library schemas. The tool might generate C++ glue code for nonleaf classes, which cannot be compiled.

Solaris example

For example, the **jcpp** demo uses this command line on a Solaris machine:

```
ossg -I/opt/ODI/ostore/include
   -I/opt/ODI/osji/include
   -I/usr/local/jdk11/include
   -I/usr/local/jdk11/include/solaris
   -I.
   -store_member_functions -store_function_parameters
   -assf jcgen_schema.cc
   -asdb CPerson.jcgen_adb
   schema.cc
```

Windows using the Sun JDK example

On a Windows machine, the command line would look like this:

```
ossg -Ic:\ODI\OSTORE\include
   -Ic:\ODI\OSJI\include
   -Ic:\jdk11\include
   -Ic:\jdk11\include\win32
   -I. \
   -store_member_functions -store_function_parameters
   -asof jcgen_schema.obj
   -asdb CPerson.jcgen_adb
   schema.cc
```

Windows using the Microsoft VM example

If you are building your application on a Windows machine for the Microsoft VM, you need to add **-DMicrosoft_VM**, as shown in the following example:

```
ossg -Ic:\ODI\OSTORE\include
    -Ic:\ODI\OSJI\include
    -Ic:\jdk11\include
    -Ic:\jdk11\include\win32
    -I. \
    -DMicrosoft_VM
    -store_member_functions -store_function_parameters
    -asof jcgen_schema.obj
    -asdb CPerson.jcgen_adb
    schema.cc
```

Digital UNIX example

On a Windows machine, the command line would look like this:

```
ossg -xtaso_short -vptr_size_short
    -I/usr/opt/ODI/OSTORE/include
    -I/usr/opt/ODI/OSJI/include
    -I/usr/include/java
    -I/usr/include/java/alpha
    -I.
    -store_member_functions -store_function_parameters
    -assf jcgen_schema.cc
    --asdb CPerson.jcgen_adb
    schema.cc
```

Step 2: Generate Java Peer Classes and C++ Glue Code

Run the **osjcgen** tool to create the Java peer class files and the C++ glue code.

Solaris example

For the **jcpp** demo, the command line looks like this on Solaris:

```
osjcgen  -package COM.odi.demo.jcpp
   -libdir /opt/ODI/osji
   -classdir /opt/ODI/osji
   -schema CPerson.jcgen_adb
   -native_interface jni
   CPerson
```

Windows using the Sun JDK example

On Windows, the command line would look like this:

```
osjcgen -package COM.odi.demo.jcpp
   -libdir c:\ODI\OSJI
   -classdir c:\ODI\OSJI
   -schema CPerson.jcgen_adb
   -native_interface jni
   CPerson
```

Windows using the Microsoft VM example

```
osjcgen -package COM.odi.demo.jcpp
   -libdir c:\ODI\OSJI
   -classdir c:\ODI\OSJI
   -schema CPerson.jcgen_adb
   -native_interface ms_raw
   CPerson
```

Digital UNIX example

For the **jcpp** demo, the command looks like this on Digital UNIX:

```
osjcgen -package COM.odi.demo.jcpp
   -libdir /usr/opt/ODI/OSJI
   -classdir /usr/opt/ODI/OSJI
   -schema CPerson.jcgen_adb
   -native_interface jni
   CPerson
```

Step 3: Write Java Classes

After you run the peer generator tool, you can write Java classes that use the generated peer classes.

Step 4: Compile Java Peer Classes and Java Files

Compile your Java source files and your Java peer classes. Be sure to compile all files together to assure consistency. For example, in the **jcpp** demo, these files are compiled together:

```
javac Main.java CPerson.java
```

This command line also compiles **CPersonClassInfo.java**. The **ClassInfo** subclass for a peer class is automatically compiled when you compile the peer class.

Step 5: Generate C Header File

Run the **javah** tool to generate the C header file for the generated and compiled Java peer classes. For example, in the **jcpp** demo, the commands look like this:

```
javah -o CPerson_stubs.h COM.odi.demo.jcpp.CPerson
```

Step 6: Compile C++ Glue Code

Run the C++ compiler on the C++ glue code.

Be sure to specify the correct include directories.

Solaris

On Solaris, compile the files with the **-PIC** (C++) option. This is required for the object files to be included in a shared library to be loaded by the Java run time.

Here are sample command lines for the **jcpp** demo:

Solaris C++ compile

```
CC -c -PIC -vdelx
   -I.
   -I/opt/ODI/ostore/include
   -I/opt/ODI/osji/include
   -I/usr/local/jdk11/include
   -I/usr/local/jdk11/include/solaris
   CPerson_glue.cc
```

Windows C++ compile using the Sun JDK

```
CC /c /GX /MD /Tp
   /Ic:\ODI\OSTORE\include
   /Ic:\ODI\OSJI\include
   /Ic:\jdk11\include
   /Ic:\jdk11\include\win32
   CPerson_glue.cc
```

Windows C++ compile using the Microsoft VM

```
CC /c /GX /MD /Tp
   /Ic:\ODI\OSTORE\include
   /Ic:\ODI\OSJI\include
   /Ic:\jdk11\include
   /Ic:\jdk11\include\win32
   /DMicrosoft_VM
   CPerson_glue.cc
```

Digital UNIX C++ compile

```
cxx -c -xtaso_short -vptr_size_short
   -I.
   -I/usr/opt/ODI/OSTORE/include
   -I/usr/opt/ODI/OSJI/include
   -I/usr/include/java
   -I/usr/include/java/alpha
   CPerson_glue.cc
```

Step 7: Generate the Application Schema Database for Your Library

Generate the application schema database for the library for your Java application. As input to the schema generator, specify a schema source file and/or one or more library schema databases. The **ossg** command line is the same as when you generated the mapping schema database, except that you

- Do not specify the **-store_member_functions** and **-store_function_parameters** options
- Must specify at least two ObjectStore library schemas

Required library schemas

You must specify the ObjectStore Java interface library schema database and the ObjectStore collections library schema database. The library schema database for collections is required even if your application does not use ObjectStore collections.

On Solaris, the ObjectStore Java interface library schema database is **libosji.ldb**. The collections library schema is **liboscol.ldb**.

On Windows, the ObjectStore Java interface library schema database is **osji.ldb**. The collections library schema is **os_coll.ldb**.

On Digital UNIX, the ObjectStore Java interface library schema database is libosji.ldb. The collections library schema is liboscol.ldb and the schema evolution library schema is libosse.ldb.

Schema source file as input

If the input is a schema source file, the include options must specify the

- ObjectStore C++ **include** directory
- ObjectStore Java **include** directory
- Java **include** directory
- Java system-specific **include** directory

Solaris example

For example, the **jcpp** demo uses this command line on a Solaris machine:

```
ossg -I/opt/ODI/ostore/include
   -I/opt/ODI/osji/include
   -I/usr/local/jdk11/include
   -I/usr/local/jdk11/include/solaris
   -I. \
   -assf jcpp_schema.cc
   -asdb CPerson.jcpp_adb
   schema.cc \
   /opt/ODI/osji/lib/libosji.ldb
   /opt/ODI/ostore/lib/liboscol.ldb
```

Windows using the Sun JDK example

On a Windows machine, the command line would look like this:

```
ossg -Ic:\ODI\OSTORE\include
   -Ic:\ODI\OSJI\include
   -Ic:\jdk11\include
   -Ic:\jdk11\include\win32
```

```
-I.
-asof jcpp_schema.obj
-asdb CPerson.jcpp_adb
schema.cc
c:\ODI\OSJI\lib\osji.ldb
c:\ODI\OSTORE\lib\os_coll.ldb
```

Windows using the Microsoft VM example

On a Windows machine for the Microsoft VM, the command line would look like this:

```
ossg -Ic:\ODI\OSTORE\include
    -Ic:\ODI\OSJI\include
    -Ic:\jdk11\include
    -Ic:\jdk11\include\win32
    -I.
    -DMicrosoft_VM
    -asof jcpp_schema.obj
    -asdb CPerson.jcpp_adb
    schema.cc
    c:\ODI\OSJI\lib\osji.ldb
    c:\ODI\OSTORE\lib\os_coll.ldb
```

Digital UNIX example

On a Digital UNIX machine, the command line would look like this:

```
ossg -xtaso_short -vptr_size_short
    -I/usr/opt/ODI/OSTORE/include
    -I/usr/opt/ODI/OSJI/include
    -I/usr/include/java
    -I/usr/include/java/alpha
    -I.
    -assf jcpp_schema.cc
    -asdb CPerson.jcpp_adb
    schema.cc
    /usr/opt/ODI/OSJI/lib/libosji.ldb
    /usr/opt/ODI/OS5.0.sp3/lib/liboscol.ldb
    /usr/opt/ODI/OS5.0.sp3/lib/libosse.ldb
```

Step 8: Create the Library

Create a library that contains object files for all C++ code in your application. On Solaris, you create a shared library. On Windows, you create a DLL. You must include

- C++ methods. In the **jcpp** demo, this is **cperson.o**.
- **osjcgen** glue code. In the **jcpp** demo, this is **CPerson_glue.o**.
- Schema object file. In the **jcpp** demo, this is **jcpp_schema.o**.
- ObjectStore Java interface library that does not include schema (**osji**).
- On Solaris, you must also include the ObjectStore C++, collections, query, and thread libraries. For the threads library, be sure to include **libosths** and not **libosthr.**

Here are sample commands for creating the library.

Solaris

```
CC -G -o libCPerson.so
    cprsn.o CPerson_glue.o
    jcpp_schema.o -losji -los -loscol -losqry -losths
```

Windows using the Sun JDK

```
link /subsystem:console /DLL -out:CPerson.dll
    cprsn.obj CPerson_glue.obj
    jcpp_schema.obj c:\ODI\OSJI\lib\osji_jdk.lib
```

Windows using the Microsoft VM

```
link /subsystem:console /DLL -out:CPerson.dll
    cprsn.obj CPerson_glue.obj
    jcpp_schema.obj c:\ODI\OSJI\lib\osji_ms.lib
```

Digital UNIX

```
cxx -shared -taso -o libCPerson.so
    cprsn.o CPerson_glue.o jcpp_schema.o
    -losji -losdmadmut -losdmajcpp -los -loscol
    -losqry -losthr
```

Running the Postprocessor

You do not need to postprocess peer classes. However, you do need to postprocess Java classes that are not peer classes and that you want to be persistence-capable. If you have such classes, run the postprocessor on them and specify peer classes that you are generating and their **ClassInfo** implementations with the **-copyclass** option. This copies the peer classes to the postprocessor destination directory but does not modify the peer classes. Alternatively, ensure that these peer classes are in your **CLASSPATH** when you run your application.

You can run the postprocessor any time after you compile your Java source files.

For example, if you wanted a Java class called **Group** to be persistence-capable, you would specify something like the following:

```
osjcfp -dest osjcfpout Group.class -copyclass CPerson.class
```

When you run the postprocessor on your Java application classes, be sure that the compiled peer classes are in your class path in a directory other than the postprocessor destination directory. Complete information about the postprocessor is in the *ObjectStore Java API User Guide*, Chapter 8, Automatically Generating Persistence-Capable Classes.

UNIX: Using Additional Native Libraries with the Java Interface to ObjectStore

If your application has only one native library and you follow the instructions in this chapter, you should not encounter the problem described in this section.

On Solaris and Digital UNIX platforms, if your application loads additional native libraries, you might receive an error like the following:

```
Run-time error, libC:
        'delete[]' does not correspond to any 'new[]'
SIGABRT   6*   abort (generated by abort(3) routine)
    si_signo [6]: SIGABRT   6*   abort (generated by abort(3) routine)
    si_errno [0]: Error 0
    si_code [0]: SI_USER [pid: 12891, uid: 161]
        stackbase=EFFFEE00, stackpointer=EFFFCD60
```

The cause of this is that ObjectStore applications expect to find the ObjectStore library, **libos,** in the library search path before the C++ run-time library. To resolve this problem, do one of the following:

- Create an ObjectStore session and then load your additional native library.
- Relink the additional native library to link against **libos**, **libosths**, and **libthread**. Make sure that **libos** appears before the C++ run-time library in the link line.
- Create a new native library that links **libos**, **libosths**, **libthread**, and **libC**. Use **System.loadLibrary()** to load the new library before your additional native library.

If you have an application that uses **fork** to execute your ObjectStore application, the ObjectStore application automatically receives copies of the forking process's shared libraries. If the forking process uses the C++ run-time library, this causes a problem. To resolve the problem, use the middle option listed above or explicitly load the ObjectStore library.

Chapter 5
Using ObjectStore Peer Collections

ObjectStore peer collections can contain both primary objects and peer objects. Peer collections are designed for storing large lists, arrays, and other aggregation data structures. To achieve the best possible performance, these collections are implemented in a combination of C++ and Java. The more performance-critical aspects are implemented entirely in C++.

Contents

This chapter discusses the following topics:

For information about pure Java collections, see *ObjectStore Java API User Guide*, Chapter 7, Working with Collections.

Introduction to ObjectStore Java Interface Peer Collections

A collection is an object that serves to group together other objects. It provides a convenient means of storing and manipulating groups of objects, and supports operations for inserting, removing, and retrieving elements. ObjectStore Java interface peer collections also support set-theory operations, such as intersection, and set-theory comparisons, such as subset.

Collections form the basis of the ObjectStore peer query facility, which allows you to select those elements of a collection that satisfy a specified condition.

The ObjectStore peer collections facility allows you to create either ordered or unordered collections, and collections that either do or do not allow duplicates.

Collections are commonly used to model many-valued attributes, and they can also be used as class extents, which hold all instances of a particular class. Collections of one type, dictionaries, associate a key with each element or group of elements, and can be used to model binary associations or mappings.

Here is a simple example to help you understand how and when to use ObjectStore peer collections. First, here is an example of a program that does not use an peer collection. It is followed by code that does use a collection.

```
import java.util.Vector;
public class PartCatalog {
    Vector parts = new Vector();
    void insert(Part part) {
        parts.addElement(part);
    }
}
class Part {
    int partNumber;
    Employee responsibleEngineer;
    Part (int partNumber,
            Employee responsibleEngineer,
            PartCatalog catalog) {
        this.partNumber = partNumber;
        this.responsibleEngineer = responsibleEngineer;
        catalog.insert(this);
    }
}
class Employee {
    String name;
}
```

The following class uses an instance of **COM.odi.coll.Set**, one of the interfaces supplied by the peer collections facility. The definitions for the **Part** and **Employee** classes are the same as before.

```
import COM.odi.*;
import COM.odi.coll.*;
public class PartCatalog {
    Set parts;
    PartCatalog (Placement placement) {
        parts = NewCollection.createSet(placement, options);
    }
    void insert (Part part) {
        parts.insert(part);
    }
}
class Part {
    int partNumber;
    Employee responsibleEngineer;
    Part (int partNumber,
            Employee responsibleEngineer,
            PartCatalog catalog) {
        this.partNumber = partNumber;
        this.responsibleEngineer = responsibleEngineer;
        catalog.insert(this);
    }
}
class Employee {
    String name;
}
```

Creating New Peer Collections

Use the static methods defined in the **NewCollection** class to create new peer collections. These methods create Java peer objects that identify persistent C++ collections; you cannot use them to create transient collections.

Each kind of peer collection has default behavior. If you do not want the default behavior, pass a **CollectionOptions** object to the **NewCollection** method that you invoke to create your collection. You can reuse a **CollectionOptions** object as many times as you want. You can alter it between uses. The collection creation operation checks the content of the **CollectionOptions** object, but does not store any references to the **CollectionOptions** object.

You can invoke the **Database.createRoot()** method to create a root for a collection. If the object you assign to the root is the object that represents the collection, a call to the **Database.getRoot()** method returns a reference to the Java peer object that represents the collection. The **getRoot()** call does not return all objects in the collection.

Retaining collections not allowed

Peer objects, and therefore peer collection objects, have no data members and so, in Java, peer objects do not appear to be connected to other objects. Even if you explicitly iterate through the elements in a collection and then commit the transaction with **RETAIN_READONLY**, you cannot access the collection outside a transaction. However, if the collection elements are not themselves peer objects or collections, you can manipulate them outside a transaction but you cannot use the collection to access them. You must explicitly read them in the transaction, retain them, and then access them directly or through another nonpeer object.

Destroying collections

When you invoke **ObjectStore.destroy()** on a peer collection object, ObjectStore leaves a tombstone. If any objects try to access the destroyed object, the tombstone causes ObjectStore to throw COM.odi.ObjectNotFoundException.

Exporting collections	Peer collections are a special kind of Java peer object. While you cannot export other kinds of Java peer objects, you can export peer collection objects.
Calls to **deepFetch()**	When you call **ObjectStore.deepFetch()** on a peer collection, ObjectStore does not retrieve anything, since a peer object has no contents.

Which Objects Can Be Inserted in Peer Collections?

You can insert persistence-capable primary objects or null objects into peer collections. You can insert Java peer objects in peer collections if the peer objects represent persistent C++ objects. You cannot query Java peer objects, because they have no data members. You cannot insert into collections Java peer objects that identify transient C++ objects. If you try to, ObjectStore throws CollectionException.

If you insert a persistent object into a collection, at least one of the following must be true:

- The collection and the object being inserted must be in the same segment.
- The object being inserted must be exported.

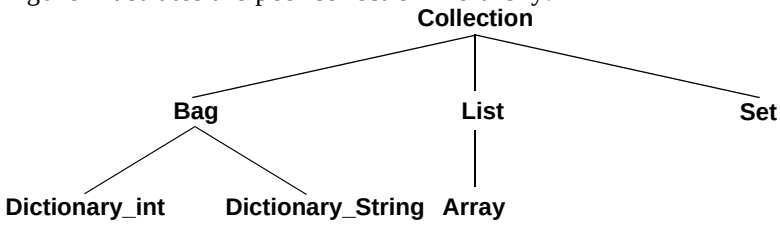
If neither is true, ObjectStore throws ObjectNotExportedException.

When you insert a transient persistence-capable object, ObjectStore immediately makes the object persistent. ObjectStore places this object in the same segment as the collection and does not make this object exported.

If your application inserts objects into peer collections during construction of the objects being inserted, you must specify the **-noinitializeropt** option when you run the postprocessor. Doing so avoids errors in the handling of modifications to the newly constructed objects.

Choosing a Peer Collection Interface

The **Collection** interface provides most of the methods that you can use to operate on peer collections. Three types of peer collections inherit directly from **Collection** — bags, lists, and sets, which are all interfaces. Bags and sets do not order elements, while lists do. Bags allow duplicates, sets do not, and lists can allow or not allow duplicates. There are two types of dictionaries that inherit from **Bag**. The additional feature of the dictionaries is that they maintain a key for each element. Finally, arrays inherit from **List**. Arrays allow null elements and provide for constant time retrieval based on an element's position. The following figure illustrates the peer collection hierarchy.



Along with the **Collection** interface hierarchy, ObjectStore provides the **NewCollection** class, which provides static methods for creating the various types of peer collections. ObjectStore also provides the **Cursor** and **ListCursor** interfaces for iterating over the elements in a collection.

This section contains a brief description of each type of peer collection.

Sets

Sets, as with familiar data structures like linked lists and arrays, have elements. Elements are objects that the set groups together. In contrast to lists and arrays, the elements of a set are unordered. You can use sets to group objects together when you do not need to record any particular order for the objects.

Besides lacking order, something that distinguishes sets from some other types of peer collections is that they do not allow multiple occurrences of the same element. This means that inserting a value that is already an element of a set leaves the set unchanged or throws a run-time exception (depending on the behavior you have specified for the set). In either case, sets do not allow duplicates.

Bags

Bags are similar to sets, except that they allow duplicates. Bags are collections that not only keep track of what their elements are, but also of the number of occurrences of each element. The **Bag** interface provides the methods available for sets, and also the **count()** method. This method returns the number of occurrences of a given element in a given collection.

Lists

In addition to sets and bags, the peer collections facility supports *lists*. Lists are collections that associate each element with a numerical position based on insertion order. Lists can either allow or disallow duplicates (by default they allow duplicates). In addition to simple insert (insert into the beginning or end of the collection) and simple remove (removal of the first occurrence of a specified element), you can insert, remove, and retrieve elements based on a specified numerical position, or based on a specified cursor position.

Dictionaries

Like bags, *dictionaries* are unordered collections that allow duplicates. Unlike bags, however, dictionaries associate a key with each element. The key can be either an integer or a string, as represented by the interfaces **Dictionary_int** and **Dictionary_String**, respectively. When you insert an element into a dictionary, you specify the key along with the element. You can retrieve an element with a given key.

Arrays

Arrays are like lists, except that they always provide access to collection elements in constant time. That is, for all allowable representations of an **Array**, the time complexity of operations such as retrieval of the *n*th element is order 1 in the array's cardinality. Arrays also always allow null elements, and provide the ability to automatically establish a specified number of new null elements.

Description of Peer Collection Behaviors

Each type of peer collection functions according to the behavior flags that are set for it. The following table describes the behavior flags. Most of these constants are defined in **COM.odi.coll.Collection**. The others are defined in **COM.odi.coll.Dictionary_int** and **Dictionary_String**.

Behavior	*Description*
Collection.ALLOW_NULLS	Controls whether null is allowed as an element of the collection. By default, sets, bags, and lists do not allow null elements, while arrays do allow null elements. If this flag is specified, the collection allows null elements.
Dictionary_int.DONT_MAINTAIN_ CARDINALITY **Dictionary_String.DONT_MAINTAIN_ CARDINALITY**	Indicate that the dictionary does not keep track of its own cardinality (the number of elements in the collection) by increasing and decreasing a counter when elements are inserted and deleted.
Collection.MAINTAIN_CURSORS	Required for a collection to have safe cursors. You might use safe cursors when you want to update a collection while you are iterating through it. A safe cursor ensures that the iteration visits elements added since the cursor was bound to the collection, and does not visit elements removed since the cursor was bound to the collection. Also, you can reposition a safe cursor if it becomes invalid.
Collection.PICK_FROM_EMPTY_ RETURNS_NULL	Controls what happens when a **pick()** method is invoked on an empty collection. If this flag is specified, the **pick()** method returns null. If this flag is not specified, NoSuchElementException is thrown.
Dictionary_int.SIGNAL_DUP_KEYS **Dictionary_String.SIGNAL_DUP_KEYS**	Indicate that duplicate keys are not allowed in the dictionary.
Collection.SIGNAL_DUPLICATES	Controls what happens if there is an attempt to insert a duplicate element into a set. This behavior is allowed only for sets because the other kinds of collections allow duplicates. If this flag is not specified, the insertion does nothing. If it is specified for the set, ObjectStore throws DuplicateEntryException if there is an attempt to insert a duplicate element.

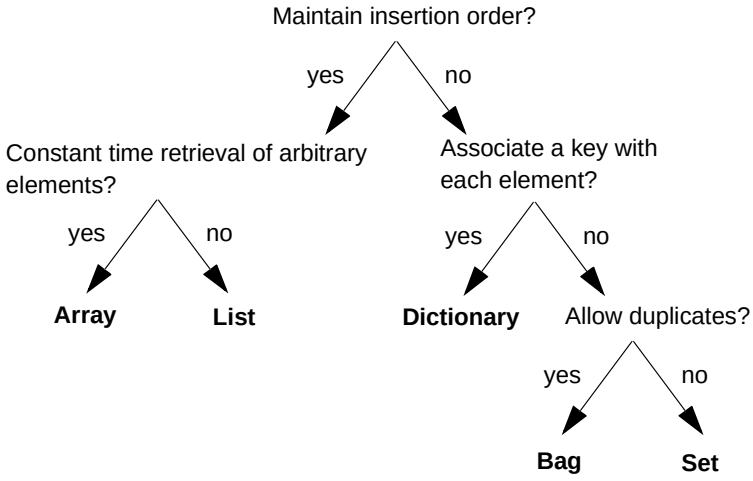
Default Behaviors for Each Kind of Peer Collection

There are default behaviors for each kind of collection. Some behaviors apply to only certain peer collections. Some behaviors can be changed with the **Collection.changeBehavior()** method. The following table shows which behaviors apply to which collection interface.

Interface	*Default Behavior*	*Possible Behaviors*	*Can Change with* **changeBehavior()**
Array	**ALLOW_NULLS**	**MAINTAIN_CURSORS** **PICK_FROM_EMPTY_RETURNS_NULL**	**MAINTAIN_CURSORS** **PICK_FROM_EMPTY_RETURNS_NULL**
Bag	None	**ALLOW_NULLS** **MAINTAIN_CURSORS** **PICK_FROM_EMPTY_RETURNS_NULL**	**ALLOW_NULLS** **MAINTAIN_CURSORS** **PICK_FROM_EMPTY_RETURNS_NULL**
Dictionary	None	**DONT_MAINTAIN_CARDINALITY** **PICK_FROM_EMPTY_RETURNS_NULL** **SIGNAL_DUP_KEYS**	Not allowed
List	None	**ALLOW_NULLS** **MAINTAIN_CURSORS** **PICK_FROM_EMPTY_RETURNS_NULL**	**ALLOW_NULLS** **MAINTAIN_CURSORS** **PICK_FROM_EMPTY_RETURNS_NULL**
Set	None	**ALLOW_NULLS** **MAINTAIN_CURSORS** **PICK_FROM_EMPTY_RETURNS_NULL** **SIGNAL_DUPLICATES**	**ALLOW_NULLS** **MAINTAIN_CURSORS** **PICK_FROM_EMPTY_RETURNS_NULL** **SIGNAL_DUPLICATES**

Decision Tree for Choosing a Peer Collection Type

Here is a simple decision tree to help you choose a peer collection type to suit particular behavioral requirements.



Navigating Peer Collections with Cursors

The **Cursor** and **ListCursor** interfaces help you navigate within a peer collection. A *cursor*, an instance of the **Cursor** or **ListCursor** interface, designates a position in a collection. You can use cursors to traverse collections, as well as to retrieve, insert, remove, and replace elements. This section discusses the following topics:

Creating Cursors

You create a cursor with the **Collection.newCursor()** or **List.newListCursor()** method. ObjectStore associates the cursor with **this** collection and creates it with the behaviors you specify. The method signatures are

public Cursor newCursor (int flags)

public ListCursor newListCursor(int flags)

The flags can include **Cursor.SAFE**, **Cursor.UPDATE_INSENSITIVE**, and **Cursor.ORDER_BY_ADDRESS**. You cannot specify **ORDER_BY_ADDRESS** for a dictionary.

When you create a cursor, ObjectStore positions the cursor at the collection's first element. You can then use **COM.odi.coll.Cursor** methods or **COM.odi.coll.ListCursor** methods to reposition the cursor and retrieve the element at which it is positioned.

The **ListCursor** class extends the **Cursor** class. The additional methods it provides allow you to insert and remove objects relative to the current position of the cursor. This is useful for ordered collections such as arrays and lists.

The **Cursor** class extends the standard Java **Enumeration** interface, so you can use cursors like you use any other **Enumeration** object. For example, you can use the **hasMoreElements()** and **nextElement()** methods from the **Enumeration** interface. You can pass a cursor to any Java method that accepts an **Enumeration** argument.

If your intent is to use only the **Enumeration** interface of the **Cursor** interface, consider using the **Collection.elements()** method instead of creating a cursor. The **elements()** method returns an implementation of **Enumeration** that is more efficient than the **Cursor** interface for iterating over the elements of a collection. However, **elements()** is more restrictive.

An application must destroy its cursors when it is done with them. This releases the transient C++ peer objects that are associated with the cursor. Not destroying a cursor causes a memory leak in the C++ heap.

OSJI cursors work the same way as cursors in the C++ interface to ObjectStore. For information that is not covered here, see the *ObjectStore C++ API User Guide*, Chapter 5.

Performing Collection Updates During Traversal

If you want to be able to update a collection while traversing it within the same session and transaction, you must use either an *update-insensitive* cursor, or a *safe* cursor.

If you update a collection while traversing it without using an update-insensitive or safe cursor, the results of the traversal are undefined.

Update-insensitive cursors

With an update-insensitive cursor, the traversal is based on a snapshot of the collection elements at the time the cursor was bound to the collection. None of the insertions and removals performed on the collection are reflected in the traversal.

To traverse a collection with an update-insensitive cursor, you must specify **Cursor.UPDATE_INSENSITIVE** when you create the cursor.

When you are done with an update-insensitive cursor, you should call **ObjectStore.destroy()** to destroy it before you commit the transaction. Update-insensitive cursors have large, transient, C++ data structures associated with them. This space is freed only when you destroy the cursor. If you do not destroy the cursor, you do not receive an exception, but it is a good idea to destroy the cursor to avoid performance problems.

Safe cursors

A safe cursor at a given point in a traversal visits any elements inserted later in the traversal order, and does not visit any elements that are later in the traversal order that are removed.

To traverse a collection with a safe cursor, you must specify **Collection.MAINTAIN_CURSORS** when you create the collection, and you must specify **Cursor.SAFE** when you create the cursor. If you try to create a safe cursor on a collection that does not have maintain cursors behavior, ObjectStore throws COM.odi.coll.CollectionException.

When you create a safe cursor, ObjectStore modifies the database. Consequently, the database must be open-for-update, there must be an update transaction in progress, and you must have permission to modify the database.

You must use the **ObjectStore.destroy()** method to explicitly delete every safe cursor before you commit the transaction in which the cursor was created. An attempt to commit a transaction in which there are outstanding safe cursors throws CollectionException. There is no finalizer for the **Cursor** class to do this. Since finalizers have no guarantee of running, they are not helpful in this circumstance.

You cannot create a safe cursor on any kind of dictionary. If you try to, ObjectStore throws COM.odi.coll.CollectionException.

Disadvantages of safe cursors

Safe cursors have some drawbacks that update-insensitive cursors do not:

- Updates to peer collections with safe cursors are slower. For each collection in a given segment that has **MAINTAIN_ CURSORS** behavior, there is an entry in a table that maps collections to their safe cursors. This table is stored in the same segment as the collections. An update to one of the collections requires a lookup in the table. Each cursor associated with the collection is checked and adjusted if necessary.
- Index maintenance for peer collections with safe cursors is slower. Whenever index maintenance is performed on an object in an indexed collection that has **MAINTAIN_CURSORS** behavior, the safe cursor table also has to be visited (because there might be safe ordered cursors that are pointing to the indexes).

Using safe cursors to implement recursion

One advantage of safe cursors is that you can use them to implement recursion without the use of recursive method calls.

Example of Using a Cursor

Here is a simple example that uses a cursor to visit each element in a set and insert the element into a vector.

```
Vector v;
    t = Transaction.begin(ObjectStore.UPDATE);
    s = NewCollection.createSet(db);
    s.insert("peter");
    s.insert("paul");
    s.insert("mary");
    cur = s.newCursor(0);
    v = new Vector();
    while (cur.hasMore()) {
        v.addElement(cur.retrieve());
        cur.next();
    }
    ObjectStore.destroy(cur);
    t.commit();
```

Restriction on Cursors

The **Collection** class has a number of methods that take a cursor as an argument. For these methods to work correctly, the cursor must be a safe cursor or a default cursor.

A default cursor is a cursor for which the **flags** argument equaled **0** when the cursor was created. In other words, peer collections methods that take cursors as arguments do not work correctly when **ORDER_BY_ADDRESS** or **UPDATE_INSENSITIVE** was specified when the cursor was created.

When the cursor is not a safe cursor or a default cursor, a call to a collections method that takes a cursor as an argument causes ObjectStore to throw **COM.odi.coll.**CollectionException.

Introduction to Using Peer Queries and Indexes

The Java interface peer query facility is based on the existing ObjectStore C++ query system. The peer query capability is a subset of the full ObjectStore C++ query system.

You can use a simple declarative syntax to issue peer queries from Java programs. The facility supports these comparison operators:

- **>**
- **<**
- **==**
- **!=**
- **>=**
- **<=**

You can combine these comparison operators with the following logical operators, as in C++ and Java:

- **&&**
- **||**
- **!**

Queries and indexes can refer to paths where the type of the last instance variable in the path is one of the following:

- **int**
- **java.lang.String**
- **long** - Queries on the **long** type must be bound queries.
- Object reference — Queries on object references must be bound queries.

Performing Navigational Queries

You can define peer queries that navigate through one or more objects. For example, a query can have the following meaning: "For this collection of employees, find all employees such that **employee.department.size** > 20". When you query a multistep path, all steps except the last step must be object references. (The last step can be an object reference, but it is not required.)

The peer query facility can define value-based indexes on objects of type **int**, **long**, and **java.lang.String,** and on object references. You can compare these values to a Java literal or a program variable.

You cannot call Java methods from queries.

The following sections discuss the APIs for the peer query facility. These methods are defined in the **Collection** interface.

Using Path Strings

In a peer query, a path specifies an order in which a collection should be traversed. ObjectStore uses the **COM.odi.coll.Path** class to represent a path. Instances of this class are transient objects that you define. Each **Path** object specifies

- The type of element in the collection to be traversed
- The name of a field of the element type
- A database that contains a description of the element type

A path string is one or more field names separated by periods. If there is only one field, it must be of type **int**, **long**, or **String**, or an object reference.

If there is more than one field in the path string, the first field must be a field of the element type specified in the relevant **Path** object, and it must be a class type. Call this class **A**. If the next field is the last field (that is, there are two fields in the string), it must be a field of class **A** that is of type **int**, **long**, or **String**, or an object reference. If the next field is not the last field (that is, there are more than two fields), it must be a class type. All fields in the string, other than the last one, must be class types. For example:

```
Path.create("mypackage.Employee", "dept.city.pollution", db);
```

You can use this path to create an index that speeds up a query, such as this one:

```
"dept.city.pollution > 3"
```

This query would find all employees where the employee's department's city's pollution level is greater than **3**.

A **Path** object is a transient Java peer object. When you are finished with the **Path** object, call **ObjectStore.destroy()** on it to free the C++ transient storage. The application accesses the **Path** object through a Java peer object, and the associated C++ object is not automatically garbage collected. You must explicitly delete it.

Querying a Peer Collection

Invoke **Collection.query()** to obtain a transient set of elements that comply with a query string. The method signature is

```
public Collection query(String elementType,String query)
```

This method returns a newly created transient set that contains all elements found. If the peer query finds an element more than once, it inserts the element into the result collection only once. You cannot modify a result set.

For **elementType**, specify a fully qualified Java class name. Every element of this collection must be an instance of (in the sense of **instanceof**) the named class.

For **query**, specify a query string.

When you are finished with the result set, call **ObjectStore.destroy()** on it to free the C++ transient storage. The application accesses this set through Java peer objects, and the associated C++ objects are not automatically garbage collected. You must explicitly delete them.

Here is an example of a peer query. Typically, this operation casts the result to a set.

```
Set sleepyHeads = (Set)allPeople.query("COM.research.Person",
    "sleepHours > 9");
```

Allowing Duplicates

To obtain a result set that includes duplicates, invoke the **query()** method with a third argument. The method signature is

```
public Collection query(String elementType,
        String query,
        boolean duplicates)
```

The **duplicates** argument specifies what to do if some element is found more than once by the peer query. If this is false, the method inserts an element into the result collection no more than once. If this is true, the method inserts each occurrence of the element into the result collection.

Obtaining One Element

To obtain one element that matches a peer query, invoke the **queryPick()** method on the collection. The method signature is

public Object queryPick(String elementType, String query)

If ObjectStore finds any elements, it arbitrarily returns one. If it does not find any elements, it returns null.

Do Any Elements Exist?

To determine if there are any elements in a collection that match a particular peer query, invoke the **exists()** method. The method signature is

public boolean exists(String elementType, String query)

If ObjectStore finds any elements, it returns true.

String Comparison

When ObjectStore performs a string comparison in a peer query, it uses the same ordering that is used by Java's **String.compareTo()** method. That is, ordering is based on the Unicode value of each character in the strings being compared.

Peer Query Example

The code in this section illustrates the use of the Java interface to ObjectStore peer query facility. The following sample code stores information about professional golfers. This sets up a framework for performing a peer query.

```
package COM.odi.test;
import COM.odi.*;
import COM.odi.coll.*;
// The ProGolfer classes

class ProGolfer
{
    String name;
    int age;
    int earnings;
    Agent agent;

    // Constructor:
    public ProGolfer (String name, int age, int earnings,
            Agent agent)
    {
        this.name = name;
        this.age = age;
        this.earnings = earnings;
        this.agent = agent;
    }
}

class Agent
{
    String name;
    int commission;

    // Constructor:
    public Agent (String name, int commission)
    {
        this.name = name;
        this.commission = commission;
    }
}
```

```
// Here is a program that creates a golfer, an agent, and a set
// to hold all golfers.

public class Example1
{
    public static void main (String argv[])
   {
       Session.create(null, null).join();
       Database db = Database.create (
           "PGA.odb",
           ObjectStore.ALL_READ | ObjectStore.ALL_WRITE);
       Transaction t = Transaction.begin(ObjectStore.UPDATE);
       Segment seg = db.getDefaultSegment();
       Agent pat = new Agent ("Pat", 15);
       ProGolfer tiger = new ProGolfer ("Tiger", 20, 1000000, pat);
       Set allGolfers = NewCollection.createSet (seg);
       allGolfers.insert (tiger);
       db.createRoot ( "AllGolfers", allGolfers);
       t.commit();
       db.close();
   }
}
```

What the code does

In this example, the program does the following:

1 Initializes the ObjectStore software.

2 Creates and opens a database named **"PGA.odb."**

3 Starts an update transaction.

4 Obtains the default segment.

5 Creates a persistence-capable **Agent** object and a persistence-capable **ProGolfer** object.

6 Creates a set to hold all golfers in the default segment.

7 Creates a database root with the name **"AllGolfers"** and assigns the **allGolfers** set to the root.

8 Commits the transaction, which stores **allGolfers**, **tiger**, and **pat** in **PGA.odb**.

The following code shows a peer query that retrieves all golfers who make less than $150,000 per year.

Adding a peer query

```
package COM.odi.test;
import COM.odi.*;
import COM.odi.coll.*;

class ExampleQuery
{
    public static void main(String argv[])
    {
        Session.create(null, null).join();
        Database db = Database.open
            ("PGA.odb", ObjectStore.OPEN_UPDATE);
        Transaction t = Transaction.begin(ObjectStore.READONLY);

        Set golfers = (Set)db.getRoot("AllGolfers");
        Set backToSchool = (Set)golfers.query
            ("COM.odi.test.ProGolfer", "earnings < 150000");
        Cursor golfer = backToSchool.newCursor(0);
        while (golfer.hasMoreElements())
        {
            ProGolfer g = (ProGolfer) golfer.nextElement();
            System.out.println(g.name +
                " has to attend qualifying school with earnings "
                + g.earnings);
            System.out.println(g.name +
                "s agent is " + g.agent.name);
        }
        t.commit();
        db.close();
    }
}
```

What the code does

In the preceding example, the program does the following:

1 Opens the **PGA.odb** database and starts a transaction.

2 Retrieves a database root that contains a reference to the **allGolfers** collection.

3 Searches members of the **allGolfers** collection to find the elements that satisfy the query **"salary < 150000".**

4 Displays the names, earnings, and agents of the golfers found. If no golfers in the database earn less than $150,000, the program does not display anything.

5 Ends the transaction and closes the database.

Postprocessing the classes

After you compile these four classes, you must run the postprocessor on all four classes together.

- Make the **ProGolfer** and **Agent** classes persistence-capable.
- You can make the **Example1** class persistence-aware, but it is not required because it only accesses transient instances of the classes.
- Make the **ExampleQuery** class persistence-aware. This is required because the **ProGolfer** and **Agent** classes have no accessor methods of their own. Although you never store an instance of the **ExampleQuery** class in a database, the **ExampleQuery** class does access persistenct instances of **ProGolfer** and **Agent** and therefore **ExampleQuery** must be persistence-aware.

Description of Peer Query Syntax

The syntax for the Java interface peer query facility is a simple declarative syntax. Every element of a collection must be an instance of (in the sense of **instanceof**) the same class.

Every clause of a peer query consists of three tokens:

- Field name
- Comparison operator
- Constant (or free variable name for a bound query)

Aclause that compares two fields does not meet this requirement, and is not allowed.

The legal tokens in a query string include

- Names of instance variables of type **int**, **long**, or **String**
- Constants of type **int**, **long**, or **String**. (Remember to delimit **String** constants with quotation marks.)
- Comparison operators: **<**, **>**, **==**, **<=**, **>=**, **!=**
- Parentheses
- Boolean operators: **&&**, **||**, **!**

Examples

Here are some sample queries:

- "**age == 3**"
- "**3 == age**"
- "**!(age==3)**"
- "**name == \"Jones\" && age < 30**"
- "**(gender == \"male\") && (! (salary >= 40000))**"
- "**gender == \"male\" && (age==4 || age == 5)**"

You can use variable values in queries by building the query string at run time. For example:

```
string gName = "Jones";
String query = “name == \"" + gname + "\"";
Set s1 = (Set) golfers.query("COM.odi.test.ProGolfer", query);
```

Another way to do this is to use bound queries. A bound query has less overhead when you plan to run the query with more than one different value.

Not supported

Queries that are not supported are

- Queries on types other than **int**, **long**, and **String**
- Nested queries (a query that contains a query)

Relative Values of String Fields

When performing a peer query, ObjectStore considers a null value to be less than any string value. This has meaning in the following situation:

- An application specifies an inequality query on a collection **C**, which contains objects of type **A**.
- **A** has an instance variable **B,** which is of type **String**.
- There is an instance of **A** in **C** that has a string value for **B**.
- There is an instance of **A** in **C** that has a null value for **B**.

If you perform an inequality query on **C**, ObjectStore considers the null value for **B** to be less than the string value for **B**.

For example, suppose you have a collection of three **Person** objects whose **name** fields contain **Scott**, **Zelda**, and null. Then you specify a query string like this:

```
"name <= \"Tolstoy\""
```

ObjectStore returns two **Person** objects: the ones with **Scott** and null in the name field. If you do not want the null values returned, you can rephrase your query as

```
"name <= \"Tolstoy\" && name != null"
```

Using Bound Queries

A bound query is a peer query plus values for the query's free variables. To help you use bound queries, this section discusses

About Free Variables

A free variable is a variable that is not an accessible instance variable of the class being queried. It is a new variable that you invent as a placeholder for the purpose of a query. A free variable differs from a regular variable in that you never explicitly declare a free variable. You refer to a free variable in two places:

- The query string
- The argument to the **BoundQuery.bind()** method

In both places, you must use exactly the same identifier.

Description of Query and BoundQuery classes

ObjectStore uses two abstract Java peer classes to represent bound queries.

- The **Query** class represents a query that can have zero or more free variables.
- The **BoundQuery** class represents a bound query.

Instances of these classes are always peer objects that identify transient C++ objects. Consequently, you can use them across transaction boundaries regardless of the value of the **retain** parameter. When you are done with instances of these classes, you must explicitly destroy them to free the C++ transient space.

Advantages of Bound Queries

After you create a **Query** object, you can create many different **BoundQuery** objects based on that **Query** object. Each **BoundQuery** object can have different values for the free variables. The advantage is that ObjectStore has to parse the query only once. For example, suppose you want to know how many employees are in their 20s, how many in their 30s, and how many in their 40s. It is faster to create a **Query** object and then repeatedly use it with different bound queries, than it is to perform the **Collection.query()** method each time.

Example of Using a Bound Query

Here is an example. Suppose there is a set called **"employees"** that contains objects of type **COM.odi.demo.company.Employee**. You create a peer query that uses two free variables, **"min"** and **"max"**. The **"salary"** field is in the **COM.odi.demo.company.Employee** class. After you create the peer query, you create a bound query that is based on that query. Then you bind the free variables to their values, run the query, and perform any required cleanup. Here is the code:

```
package COM.odi.demo.company;
import COM.odi.coll.*;

    class EmployeeDemo {
        Collection test(Set employees) {
        Query empSalRange =
            Query.create(
            "COM.odi.demo.company.Employee",
            "salary > min && salary < max",
            Database.of(employees));
        BoundQuery mediumSalQuery =
            BoundQuery.create(empSalRange);
        mediumSalQuery.bind("min", 10);
        mediumSalQuery.bind("max", 20);
        Collection mediumEmps =
            employees.query(mediumSalQuery);
        ObjectStore.destroy(mediumSalQuery);
        ObjectStore.destroy(empSalRange);
        return mediumEmps;
    }
}
```

Creating Peer Query Objects

Use the static method **Query.create(elementType, query, database)** to create a **Query** object. The **elementType** parameter is the name of the class of the elements in the collection being queried. The **query** argument specifies a query with free variables. The **database** argument specifies the database that contains the objects to be queried.

The syntax for the **query** argument is the same as for the **Collection.query()** method, except, the syntax here can include free variables. The name of a free variable must be a string that is

- Valid as a Java identifier
- Not the name of an accessible field in the **elementType** class

There are also **Query.createPick()** and **Query.createExists()** methods, which are similar to **Collection.pick()** and **Collection.exists()**. The **Query** methods allow you to specify free variables.

Creating BoundQuery Objects

Use the static method **BoundQuery.create()** to create a bound query object. Specify a **Query** object as an argument to this method. After you create the bound query, call the **BoundQuery.bind()** method once for each free variable in the original **Query** object.

The first argument to **bind()** is the name of the free variable. The second argument is the value to which it should be bound. This is the value the free variable has when you run the query. There are overloadings of **bind()** with different types for the second argument.

Running a Bound Query

To run a bound query, call **Collection.query(boundQuery)**. The method returns a transient collection that contains the results of the query. There are also overloadings of **Collection.queryPick()** and **Collection.exists()** that take a bound query argument. The query method must correspond to the kind of query you created. For example, if you call **Query.createPick()** to create the query, you must call **Collection.pick()** to run the query.

A bound query is considered to be fully bound after you specify it in a call to **Collection.query()**.

You can run a bound query over many different peer collections. You can run the query on databases other than the one it was originally created for. The query can extend across multiple databases.

Remember that the element type in the collection must match the type specified for the query. You can execute the same bound query over the same collection at many different times.

Querying on Object References

You can query on object references if you use a bound query. For example:

```
Department engineeringDept;
Set employees;

Query query = Query.create("COM.company.Employee",
        "dept ==eng", db);
BoundQuery boundQuery = BoundQuery.create(query);
boundQuery.bind("eng", engineeringDept);
Set engineeringEmployees = (Set)set.query(boundQuery);
```

You can use the comparison operators **==** and **!=**. The comparison operators that include **>** or **<** signs do not mean anything for objects. If you try to use relational operators, ObjectStore throws InvalidQueryStringException.

Bound queries search for actual object equality, like the **==** operator in Java. In other words, the object provided in the **bind()** method and the object found through the path must be the same persistent object. They cannot just be identical; they must be the same object.

This works for any Java class. The **String** class, however, is a special case. You can use all comparison operators on **String** classes. ObjectStore uses the **equals()** method to check for equality among **String** objects.

Wrapper classes (for example, **Integer** and **Short**) are another special case. Since ObjectStore does not maintain object identity for wrapper classes, do not specify them as object references in bound queries.

Using Indexes on Peer Collections

You can add one or more indexes to a peer collection to optimize the performance of queries over that collection. There are two steps to this process:

1 Flag a field in a persistence-capable class so that it is indexable.

2 Add an index to the collection.

Without indexing, ObjectStore performs a linear search to execute a query. When you add an index, you instruct ObjectStore to maintain an access method that allows efficient lookup. This access method consists of hash tables and/or B-trees.

You can also use an index to ensure that a collection does not have duplicate entries for a particular field. For example, you might want to ensure that a collection of **ProGolfer** objects does not contain any objects that have the same value for the **name** field. To do this, you specify the **Path.SIGNAL_DUPLICATES** option when you call **Collection.addIndex()**.

This section discusses the following topics:

Performance

The beneficial effects of indexes on searches become apparent only when the collection being searched is relatively large. If you are searching a collection with only a small number of elements, indexes probably do not speed things up. In fact, they might make performance slower than when not using indexes. A search of a small collection is so fast that the time needed to determine that an index exists and should be used is significant compared to the total search time.

In general, a few hundred elements constitute a small collection. Tens of thousands of elements constitute a large collection. For peer collections where the number of elements is somewhere in the middle, whether or not an index improves performance depends on the kind of collection, the data member the index is on, how much main memory is available, and other details peculiar to the collection.

Making Fields Indexable

To make a field indexable, specify **-indexablefield** for a particular field when you run the class file postprocessor to make the class persistence-capable. You must specify a fully-qualified field name, for example, **COM.odi.demo.people.name**. You can make one or more fields indexable.

Do not make a field indexable when you do not plan to add an index on that field. There is a small amount of overhead in execution time whenever you alter an object that has an indexable field. The overhead is the reason that all fields are not automatically indexable. There is also a fixed database storage overhead when a class has any indexable fields.

Adding Indexes

To add an index, invoke the **Collection.addIndex()** method on the collection to which you want to add the index. When you invoke this method, you pass a **Path** object, which you must have previously created. The **Path** object specifies

- The type of element the collection you are adding the index to can contain.
- A path that identifies the field to which you are adding the index. You must have previously made this field indexable. The field must be of type **int**, **long**, or **java.lang.String**, or it can be a Java object reference. If it is an object reference, the reference can be for a class or an interface. If the field is not indexable, or it is not a supported type, ObjectStore throws COM.odi.coll.InvalidQueryStringException when you try to create the **Path** object.

  To add a multistep index, such as an index on **"employee.department.size"**, each specified field must be indexable.

- A database whose schema describes the element type that the collection contains. The **Path** object never specifies the transient database.

A **Path** object is a Java peer object, so your application should destroy it after using it.

When you add the index, there are overloadings of the **addIndex()** method that allow you to specify

- Options that change the default behavior. The default is that the index is unordered and that the collection can contain duplicates.
- The location to store the index itself. The default is that it is stored in the same segment as the collection for which it is an index.

An application can add and remove indexes while it is running. The index remains in the database until you remove it with the **Collection.dropIndex()** method. The method signature is

```
public void dropIndex(Path path)
```

For example:

```
allGolfers.dropIndex (aPath);
```

You can test for the presence of a particular index. Invoke the **Collection.hasIndex()** method and pass the path that specifies the index you want to know about.

Example of Using an Index

For example, suppose you want to optimize looking up golfers in the **allGolfers** collection. You want to optimize this by name, as in the following query:

```
ProGolfer aGolfer = (ProGolfer)
    golfers.queryPick ("COM.odi.test.ProGolfer",
        "name == \"tiger\"");
```

When you run the postprocessor on the **ProGolfer** class, specify the fully qualified name of the field. For example:

```
-indexablefield COM.odi.test.ProGolfer.name
```

In your code, add an index to the **name** field in the **allGolfers** collection. First, create the **Path** object, and then, pass the path to the **addIndex()** method:

```
Path aPath = Path.create("COM.odi.test.ProGolfer", "name", db);
allGolfers.addIndex(aPath);
```

Performing a Query on Multiple Fields

You can perform a single query on several data members. When ObjectStore processes the query, it does so in this order:

1 Indexed fields in the order they appear in the query string

2 Nonindexed fields in the order they appear in the query string

Types of Indexes

The **options** argument to the **addIndex()** method specifies the type of index to create. Possible values are limited to

- **COM.odi.coll.Path.ORDERED**

  ObjectStore creates an ordered index. The advantage of an ordered index is that it can speed up inequality queries as well as equality queries. Inequality queries involve the comparison operators **<**, **>**, **<=**, and **>=**. An unordered index can speed up only equality queries.

  For equality queries

  - Looking up an element with an unordered index is faster than looking it up with an ordered index.
  - Looking up an element with an ordered index is faster than looking it up without any index.

- **COM.odi.coll.Path.SIGNAL_DUPLICATES**

  ObjectStore ensures that no two elements in the collection can have the same key value. If you try to insert an element into the collection with the same key value as some other element that is already in the collection, ObjectStore throws COM.odi.coll.DuplicateKeyException.

Restrictions

A particular index is restricted to a single data member.

Index

D

E

F

G

I

J

L

M

N

O

P

Q

R

S

T

U

V