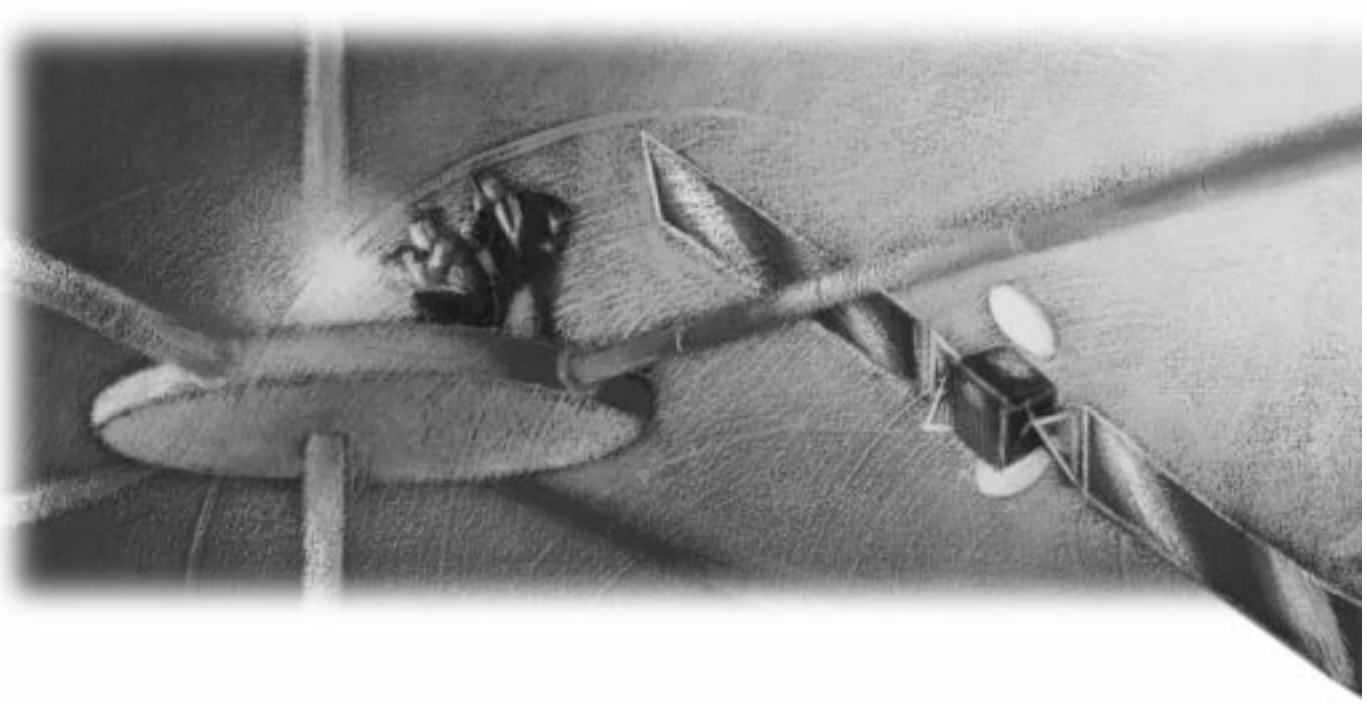


Objectivity for Java Guide

Release 6.0

Objectivity for Java Guide

Part Number: 60-JAVAGD-0

Release 6.0, September 19, 2000

Contents

About This Guide

This guide describes how to build Java applications to create and manipulate persistent objects. The guide introduces fundamental concepts and gives detailed descriptions, with examples, of the process by which you build an application.

Audience

This guide assumes that you are familiar with programming in Java.

Documentation Set

The *Objectivity for Java* documentation set consists of:

- The *Objectivity for Java Guide* (this book)
- The *Objectivity for Java Reference*, which contains complete descriptions of all classes and interfaces that constitute the Objectivity for Java programming interface

Organization

- Part 1 begins with the *Getting Started* chapter, which introduces the Objectivity / DB object-oriented database management system and the Objectivity for Java programming interface to Objectivity / DB. If you are new to Objectivity / DB or Objectivity for Java, you should start by reading *Getting Started*. The remaining chapters describe:
  - The Objectivity for Java objects responsible for the interaction between an application and Objectivity / DB.
  - How Objectivity / DB objects are represented, created and deleted, and accessed within an Objectivity for Java application.

  - The schema of a federated database and the mechanisms provided for managing and evolving the schema.
  - Objectivity for Java mechanisms that support fault-tolerant, distributed applications.
- Part 2 chapters contain reference material relevant to developers with specific needs in the areas of general query operations and interoperability.
- Part 3 chapters expand and collect definitions and examples that appeared throughout the guide or deal with an issue that pervades the Objectivity for Java programming interface. Chapters with names that are identical to chapters in Part 1 contain the complete source for the programming examples discussed in the chapters in Part 1. The files for the examples are in the *`chapter name`* subdirectory of the programming samples directory. See *Installation and Platform Notes* for the location of the samples directory for your platform.

Conventions and Abbreviations

Navigation

Table of contents entries, index entries, cross-references, and underlined text are hypertext links.

Typographical Conventions

`oobackup`	Command, literal parameter, code sample, filename, pathname, output on your screen, or Objectivity-defined identifier
`installDir`	Variable element (such as a filename or a parameter) for which you must substitute a value
Browse FD	Graphical user-interface label for a menu item or button
lock server	New term, book title, or emphasized word

Abbreviations

(administration)	Feature intended for database administration tasks
(*FTO*)	Feature of the Objectivity/DB Fault Tolerant Option product
(*DRO*)	Feature of the Objectivity/DB Data Replication Option product
(*IPLS*)	Feature of the Objectivity/DB In-Process Lock Server Option product
(*ODMG*)	Feature conforming to the Object Database Management Group interface

Command Syntax Symbols

[...]	Optional item. You may either enter or omit the enclosed item.
{…}	Item that can be repeated.
...|...	Alternative items. You should enter only one of the items separated by this symbol.
(…)	Logical group of items. The parentheses themselves are not part of the command syntax; do not type them.

Command and Code Conventions

In code examples or commands, the continuation of a long line is indented. Omitted code is indicated with the ellipsis (…) symbol. “Enter” refers to the standard key (labelled either Enter or Return) for terminating a line of input.

Getting Help

We have done our best to make sure all the information you need to install and operate Objectivity products is provided in the product documentation. However, we also realize problems requiring special attention sometimes occur.

Technical Support Web Site

You can find answers to frequently asked questions, supported platforms, known bugs, and bug fixes on the Objectivity Technical Support web site. Send electronic mail or call Objectivity Customer Support to gain access to the site.

How to Reach Objectivity Customer Support

You can contact Objectivity Customer Support by:

- **Telephone:** Call 1.650.254.7100 *or* 1.800.SOS.OBJY (1.800.767.6259) Monday through Friday between 6:00 A.M. and 6:00 P.M. Pacific Time, and ask for Customer Support.

  The toll-free 800 number can be dialed *only* within the 48 contiguous states of the United States and Canada.
- **Fax:** Send a fax to Objectivity at 1.650.254.7171.
- **Electronic Mail:** Send electronic mail to *help@objectivity.com*.

Before You Call

If you need help from Customer Support, please have the following information ready before you contact Objectivity:

- Your name, company name, address, telephone number, fax number, and email address
- Description of your workstation environment, including the type of workstation, its operating system version, compiler or interpreter, and windowing environment
- Information about the Objectivity product you are using, including the version of the Objectivity / DB libraries
- Detailed description of the problem you have encountered

Part 1 USAGE

1

Getting Started

This chapter provides an introduction to Objectivity for Java, the Java programming interface to the Objectivity / DB object-oriented database management system. The rest of this guide provides detailed conceptual information about Objectivity / DB features and how they are accessed through the Objectivity for Java programming interface. See the *Objectivity for Java Reference* for a complete description of all the classes and interfaces in Objectivity for Java.

In This Chapter

Objectivity/DB Architecture

This section gives an overview of the components of an Objectivity for Java application:

- The processes involved in an Objectivity for Java application.
- The types and function of Objectivity / DB objects accessible from within an application.
- Transactions: the mechanism for organizing operations on Objectivity / DB objects.
- The operations that applications can perform on Objectivity / DB objects.

Objectivity/DB Applications and Processes

An Objectivity / DB application works with objects stored in Objectivity / DB databases. In the Objectivity / DB architecture, Objectivity / DB applications have database services built directly into the application instead of relying on a back-end server process. This is accomplished by dynamically loading Objectivity / DB libraries into the same process space as the application.

Objectivity / DB provides simultaneous, multiuser access to databases that can be distributed across a network. A group of such databases is organized into a unit, called a *federated database*, by Objectivity / DB. All the logical entities in Objectivity / DB, including the federation, are called *Objectivity/DB objects*.

Applications do not work with Objectivity / DB objects directly; instead they work with local representations of objects, which must be retrieved from and written back to a federated database. To ensure that data maintained by Objectivity / DB objects remains consistent while being used by competing applications, Objectivity / DB uses a system of permissions, called read locks and write locks, to control access to the objects.

Locks are administered by a *lock server* that can run on any machine in the network. Before an operation can be performed on an Objectivity / DB object, an application must obtain access rights to the object from the lock server. In a standard configuration, the lock server runs as a separate process from the applications that consult it. If all lock requests originate from a single, multithreaded application, that application can optionally start its own internal lock server using a separately purchased option to Objectivity / DB, namely, Objectivity / DB In-Process Lock Server Option (Objectivity / IPLS). See Chapter 18, “In-Process Lock Server”.

Transactions

An application's access to Objectivity / DB objects is controlled by a *transaction*. Transactions control the locks acquired on behalf of an application, and the transfer of data between the local representation of Objectivity / DB objects and the objects in a federated database.

A transaction is effectively a subsection of an application, the extent of which is designated by four operations: begin, commit, checkpoint, and abort. Once an application *begins* a transaction, the application can obtain access rights to, and local representations of, Objectivity / DB objects. From this point, the application is said to be *within* a transaction. A transaction ends when it is committed or aborted. The locks on any Objectivity / DB objects are released, the local representations of Objectivity / DB objects may no longer be consistent with the objects in the federated database, and invoking a system-defined operation on an Objectivity / DB object will throw a `TransactionNotInProgressException`.

When the application *commits* the transaction, any modifications to the objects are stored in the federated database. If the application *aborts* the transaction instead of committing it, the changes are discarded (rolled back), leaving the federated database in the logical state it was in before the transaction started. An application can also *checkpoint* a transaction, which saves modifications to the federated database but retains the local representations of objects and their locks.

Objectivity / DB guarantees that certain properties—atomicity, consistency, isolation, and durability (denoted by the acronym ACID)—are maintained when the operations within a transaction are applied to a database.

Atomicity means that all the operations within a transaction are performed on the database or none is performed. Thus, several operations, on one or more of the objects contained in a database, appear to all users as a single, indivisible operation.

Consistency means that the transaction takes the database from one internally consistent state to another, even though intermediate steps of the transaction may leave the objects in an inconsistent state. This property is dependent on the atomicity property.

Isolation means that until the transaction commits, any changes made to objects are visible only to other operations within the same transaction. When a transaction commits, the changes are made permanent in the database and henceforth visible to any other concurrent users of the database. If the transaction aborts, none of the changes are made permanent in the database.

Durability means that the effects of committed transactions are preserved in the event of system failures such as crashes or memory exhaustion.

Objectivity/DB Objects

There are four types of Objectivity / DB objects: basic object, container, database, and federated database.

A *basic object* is the fundamental unit stored by Objectivity / DB. An object whose class is defined by your application is maintained by Objectivity / DB as a basic object. Each basic object is contained within a container.

Containers serve a number of purposes within Objectivity / DB. They are used:

- To group basic objects. Basic objects within a container are physically clustered together in memory pages and on disk, so access to collocated basic objects in a single container is very efficient.
- As the unit of locking. When a basic object is locked, its container and all other objects in that container are also locked. This organization reduces the burden on the lock server in systems with a large number of objects.
- Optionally, to maintain application-specific data.

Each container is contained within a database.

A *database* consists of system-created containers and containers created by your application. A database is physically maintained as a file and is used to distribute related containers and basic objects to a particular physical location. Each database is contained within a federated database.

A *federated database* consists of system-created databases and databases created by your application. The federated database maintains the object model (or *schema*) that describes all the objects stored in the databases. The schema is language independent, which means that objects of classes defined using the Objectivity for Java programming interface can be accessed and managed from other Objectivity / DB programming interfaces.

A federated database is the unit of administrative control for Objectivity / DB. A federated database maintains configuration information (where Objectivity / DB files physically reside) and all recovery and backup operations are performed at the federated database level.

Objectivity / DB objects are organized into a containment or *storage hierarchy*:

federated database<databases<containers<basic objects.

In Objectivity / DB, the objects (federated database, database, and container) that *contain* other objects are called *storage objects*. Storage objects group other objects to achieve performance, space utilization, and concurrency requirements. The behavior of storage objects is defined solely by Objectivity / DB.

Objects that represent persistent data are called *persistent objects*. The behavior of persistent objects is defined by Objectivity / DB and your application. Basic objects are persistent objects. If you choose to associate application-specific data with a

container, such a container will function as a persistent object as well as a storage object.

Operations on Objectivity/DB Objects

Besides the basic object life-cycle operations of creation, deletion, and property management, Objectivity / DB objects provide additional operations to support their transfer between an application and a federated database.

In Objectivity for Java, persistent objects (including containers) are initially created as transient Java objects. An application must perform a specific operation called *clustering* that causes a transient object to be assigned a storage location in a database. Databases are always persistent and do not have to be clustered. For both persistent objects and databases, the assignment is not permanent until the transaction in which the assignment occurred commits.

Once an object exists within Objectivity / DB, you need to perform the following steps before you can access the object:

1. Create a local representation of the object. All subsequent operations on the object are directed to this local representation. This operation is also referred to as *retrieving* an object or *getting a reference to* an object.
2. Obtain a lock on the object. Objectivity / DB has two kinds of locks: read locks and write locks. A read lock indicates to Objectivity / DB that you need read-only access to an object. A write lock indicates that you intend to modify the object.
3. If the object has application-specific persistent data, copy the object's data into the local representation of the object. This operation is referred to as *fetching* the object's persistent data.

Because of the different role of storage and persistent objects, these operations are provided by slightly different mechanisms, and are subject to different policies for the different types of objects. The mechanisms and policies, as well as the basic object life-cycle management operations, are discussed in the following sections.

Storage Objects

Creating and Deleting Storage Objects

You can create and delete storage objects in two ways:

- Writing an application that uses one of the programming interfaces to Objectivity / DB.
- Using one of the operation-specific tools supplied with Objectivity / DB.

The following table summarizes the operations available for each kind of storage object:

	API (create and delete)	**Tool** (create and delete)
Container	X	
Database	X	X
Federated Database		X

Deleting a storage object deletes all of the objects contained in it.

Identifying Storage Objects

All storage objects are given identifiers when they are created. The federated database identifier is an integer, specified by the creator of the federated database.

Databases and containers also have integer identifiers. Typically these are assigned by Objectivity / DB when either type of object is created. The creator of a database, however, has the option of specifying its identifier.

Storage object identifiers are used to identify their respective objects to the lock server. The identifiers of a database and container appear as components of the identifiers of the persistent objects stored within them.

System Names

A *system name* uniquely identifies a storage object to Objectivity / DB. System names are mandatory for federated databases and databases and are optional for containers. System names have the following characteristics:

- Each federated database, database, or container can have only one system name.
- The system name is set when the object is created and cannot change for the lifetime of the object.
- The system name must be unique in the containing object.
- Valid system names follow the same rules as for file names within the operating system.

Retrieving Storage Objects

You can retrieve a preexisting storage object by:

- Looking up a storage object by its system name.
- Retrieving the storage objects contained in a storage object one level up in the storage hierarchy.

Locking Storage Objects

Whenever you perform an operation on a storage object, Objectivity / DB will *implicitly* obtain the locks needed by your application at the point at which they are needed. For example, when you create a new container in a database, Objectivity / DB will get a write lock on the new container and the database.

You may also *explicitly* request a lock in advance of actually using the object, to ensure that you will not be prevented from getting a lock because the object has been locked by another application.

When a federated database or database is locked implicitly, it can be shared. The role of the lock in this situation is not to control access, but to instruct Objectivity / DB to record modifications to the object in the transaction's *journal file*. Objectivity / DB uses the journal file to restore a federated database to its previous state if the transaction is aborted or terminated abnormally. In contrast, when a federated database or database is locked explicitly, it can be used only by the locking application. A write lock prevents all other concurrent access, and a read lock prevents another application from modifying the locked object.

Containers are the most visible unit of locking for users of persistent objects. When any basic object in a container is locked, the entire container is locked, effectively locking all basic objects in the container. Whenever a lock is requested for a container, Objectivity / DB applies a *concurrent access policy* to determine whether the requested lock is compatible with any existing locks.

Concurrent Access Policies

Whenever a lock on a container is requested, a concurrent access policy is applied to determine whether the lock should be granted. Objectivity / DB supports two concurrent access policies: *exclusive* and *MROW*.

The *exclusive* concurrent access policy rules are:

- If an application has a read lock on a container, any other application may also obtain a read lock on the same container.
- As long as a container is locked in read mode by at least one application, no other application can obtain a write lock on it.
- If an application has a write lock on a container, no other application may obtain a lock (read or write) on the same object.

The *multiple readers, one writer* (MROW) policy is useful for applications that would rather access a potentially out-of-date object than be prevented from accessing the object at all. MROW is used to allow access to the last committed or checkpointed version of a container being updated by another application.

The MROW concurrent access rules are:

- An application with MROW enabled can get a read lock for a container that is locked for write by another application.

- An application can get a write lock for a container that is locked for read by another application with MROW enabled.
- Write locks are still mutually exclusive. If an application has a write lock on a container, no other application may obtain a write lock on the same container.
- The application must not be in a MROW transaction because of the possibility of compromising referential integrity.

Persistent Objects

A persistent object is the representation of an object within an Objectivity / DB federated database. Both basic objects and containers can maintain application-specific data, and thus can serve as persistent objects.

A persistent object is an instance of a persistence-capable class.

Defining Persistence-Capable Classes

The first step in developing an object-oriented application is to define the classes that capture the structure and behavior of the fundamental entities in the application. Such definitions usually arise naturally out of the logical modeling phase of application development. With classes whose instances will be persistent, two additional steps must be added to the definition process:

1. The class definition must be modified so that instances of that class can be persistent as well as transient.
2. The class definition must then be added to or *registered with* the schema of a federated database before instances of the class can be stored in the database.

One of the defining characteristics of object-oriented databases is that persistent objects are accessible in a natural fashion from object-oriented programming languages. In order for this to be possible, persistence behavior must be associated with classes previously capable of producing only transient objects. In Objectivity for Java, the developer must explicitly add persistence behavior to application classes by deriving from a specific superclass, or by having the application class implement a simple interface. The classes with persistence behavior are said to be *persistence capable*. In addition to transient and persistence-capable classes, Objectivity for Java also supports *non-persistence-capable classes*, which are classes that cannot be used to create persistent objects directly. Objects of these classes are stored in the federated database when they are used as attributes of persistence-capable classes. An example of a non-persistence-capable class is the `String` class.

Defining a persistence-capable class is very similar to defining a class used to create objects only within the application. In particular, classes whose instances are stored by Objectivity / DB may contain any of the following types of data:

- Primitive types
- Fixed-sized arrays
- Certain non-persistence-capable classes
- Persistence-capable classes
- Persistent-collection classes

Objectivity / DB also provides a number of capabilities for modeling links or *relationships* between objects, enabling a higher level of functionality than simply using references to related objects. You can specify the directionality and cardinality of relationships, how relationships are handled when objects are copied or versioned, and whether operations on objects propagate along relationships.

Once the class definitions are complete, they must be added to the federated database's schema. In Objectivity for Java, classes are added implicitly when objects are made persistent, or explicitly through schema management methods in the programming interface.

Creating Persistent Objects

At creation time, both containers and basic objects are transient and must be made persistent. There are two mechanisms for making a transient object persistent:

- By establishing an association between the transient object and a persistent object, or naming the transient object in the context of a persistent object. The transient object is made persistent and is stored or *clustered* in the same container (if the object is a basic object) or database (if the object is a container) as the already persistent object.
- By making an explicit call to cluster the object in a particular container or database.

Note that even though an object is persistent after one of these operations, the object is not visible to other transactions until the transaction in which it was made persistent commits or checkpoints.

The way you cluster basic objects into containers affects the concurrency and performance characteristics of your application and the storage characteristics of your federated database. Objectivity for Java provides the ability to trade off concurrency versus space utilization and runtime performance.

Deleting Persistent Objects

The way basic objects are deleted is determined by the type of container, *garbage-collectible* or *non-garbage-collectible*, in which the objects are stored.

In a *garbage-collectible container*, objects are deleted when they are no longer referenced by another object. Such containers are intended to store objects that are related to one another through references or relationships. Just as memory can contain objects that are no longer referenced, a garbage-collectible container can include invalid objects that were left over after some object references were deleted. Objectivity / DB provides a garbage collector that locates and deletes unreferenced objects in this kind of container. However, unlike the garbage collectors available for program execution environments, the Objectivity / DB garbage collector is run under the control of the database administrator.

In a *non-garbage-collectible container*, invalid objects must be explicitly tracked and deleted by an application. Such containers are primarily designed for use by an application that must interoperate with an application written in a non-garbage-collected language, such as C++. They could also be used to store objects that are not necessarily connected to one another by references. For example, you might create a non-garbage-collectible container to store all objects of a particular class. The default container of a database is non-garbage-collectible to provide interoperability with C++.

Identifying Persistent Objects

When an object becomes persistent, Objectivity / DB automatically assigns it a unique *object identifier* (OID). An OID is 64 bits in length and is composed of four 16-bit fields identifying the database (*D*), container (*C*), logical page number (*P*), and logical slot number (*S*) of the object. The string representation of an OID has the form *D-C-P-S*. Objectivity / DB uses OIDs instead of memory addresses to identify objects, because OIDS provide:

- Transparent access at runtime to objects located anywhere in a network.
- Full interoperability across all platforms.
- Access to more objects than a direct memory address permits.
- Integrity constraints and runtime type checking that are not possible through direct-memory addresses.

Although storage objects do not have OIDs, database and container identifiers can be cast in the same form as an OID and can be used to reference databases and containers. A database's OID is *D*-0-0-0; the OID for a container is *D-C-P*-1.

A persistent object's OID can change during the lifetime of the object only if the object is moved to a new container. When a persistent object is moved or deleted, its OID may be reused for a new persistent object. Application developers do not

need to manage or access OIDs. OIDs are reported in some exceptions to identify particular objects.

Naming Persistent Objects

Naming a persistent object helps you retrieve the object from the database by enabling you to look it up by name. It is also one way of making a transient object persistent.

Objectivity / DB provides several mechanisms to allow you to name persistent objects. You can:

- Assign a *root name* to a persistent object that is the root of a directed graph of persistent objects. Such an object is called a *named root*. Federated databases and databases maintain root names.
- Assign a *scope name* to an object that is unique within the context or *name scope* of a *scope object*. Any Objectivity / DB object can be a scope object.
- Create a name map using the Objectivity / DB name-map class.

Retrieving Persistent Objects

You can retrieve a preexisting persistent object using any of a variety of mechanisms:

- Look up an object by its (root or scope) name.
- Retrieve the objects which maintain a scope name for a given object, and retrieve the objects named within the scope of a given object.
- Retrieve the objects related to a given object.
- Retrieve the objects in a persistent collection.
- Retrieve the objects contained in a storage object.

Many of the retrieval methods allow you to filter for those objects that:

- Are of a particular class and its subclasses.
- Meet the conditions in a predicate. This allows you to search for objects by the value of one or more of their fields.

Indexes

Searching for objects that meet the conditions in a predicate is an expensive operation when the number of objects being searched is very large. To optimize such searches, Objectivity / DB supports the definition of *indexes*, which sort persistent objects according to the values in one or more of their fields.

Objectivity / DB indexes have the following characteristics:

- They can be created and deleted at runtime from any application.
- You can have as many indexes as you need.

- Indexes can reference any object of a given class in a given container.
- You can use many different indexes on the same objects.

Locking Persistent Objects and Fetching Their Data

Before you can access a retrieved persistent object, you must lock it and fetch its data. In contrast to storage objects, which in most cases are locked automatically by Objectivity/DB, persistent objects must always be locked by your application. Typically this happens when you fetch the object's persistent data. You can also reserve access to the object by explicitly requesting a lock on the object before you fetch its data.

Whenever a basic object is locked, the container in which it is stored is locked. As a consequence, the rules for concurrent access of basic objects are the same as the rules for containers.

Evolving Classes of Persistent Objects

At some point during the lifetime of your Objectivity for Java application, you may need to modify your class definitions and consequently, the underlying schema. Once you deploy a federated database, its schema, and database applications to your end users, it will not be practical for your end users to delete their federated databases and re-create them if the schema changes. A database must provide mechanisms to:

- Evolve a schema based on changes to the Java class definitions.
- Convert existing data in a federated database to new class definitions.

These two processes are known as *schema evolution* and *object conversion*, respectively.

Schema Evolution

Schema evolution is required if you make a change to an application-defined persistence-capable class contained in the schema of a federated database. The changes can include:

- Deleting, adding, or changing class contents, including persistent fields and relationships.
- Modifying the inheritance hierarchy.

Objectivity for Java migrates a schema implicitly when objects with a modified definition are made persistent or when an Objectivity/DB object is fetched, and Objectivity for Java detects that the Java class definition does not match the schema. You can also explicitly trigger schema migration through methods in the programming interface.

Object Conversion

When you change a schema, existing objects in a federated database that are based on the changed classes may need to be converted to reflect the schema changes. These objects are called *affected objects*.

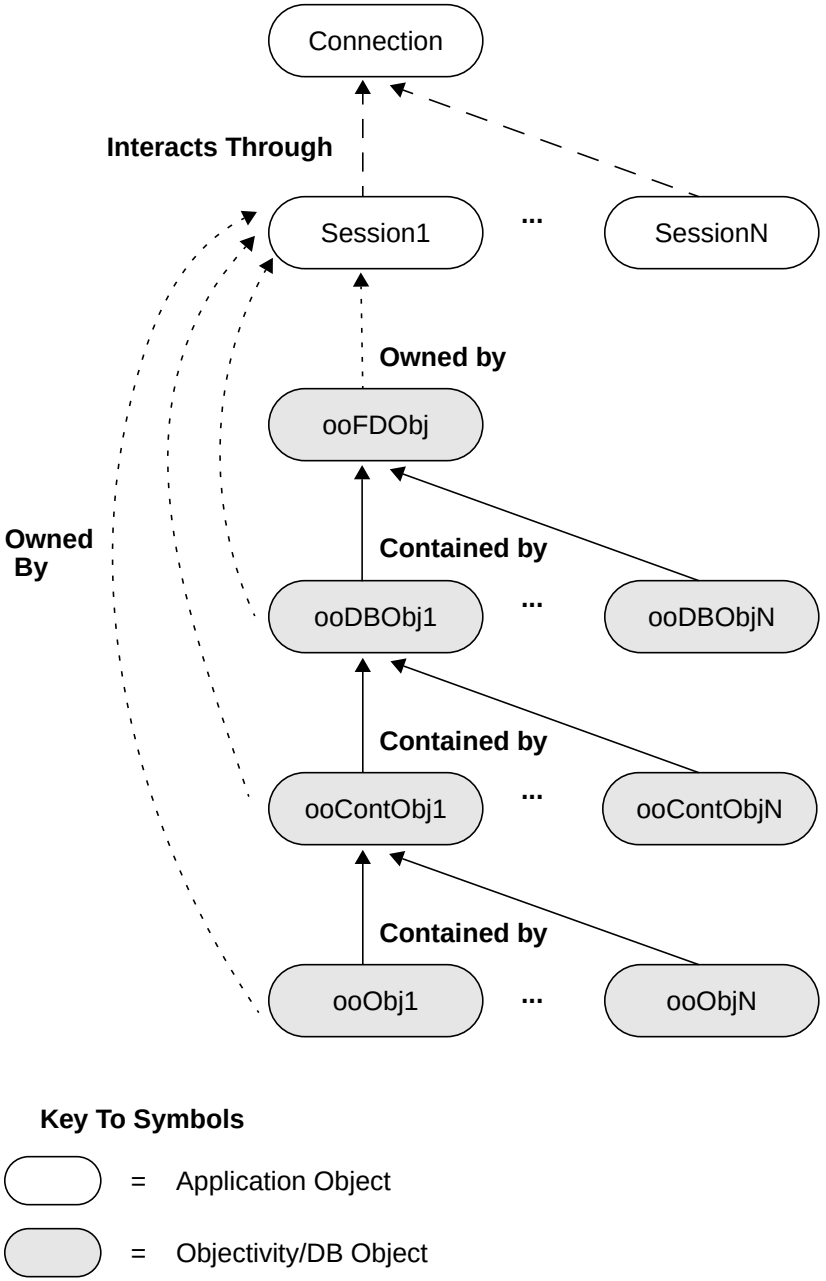
Objectivity for Java will implicitly perform *object conversion* on affected objects when they are accessed. You can also explicitly convert all affected objects in a particular storage object.

Objectivity for Java API

This section gives an overview of the main classes in the Objectivity for Java programming interface. The classes are introduced in the order they would be used in implementing a simple application. The role of each class is explained and examples illustrate how to use the classes. A complete application combines all the examples.

The Objectivity for Java API provides some classes that directly map to Objectivity / DB features. These include `ooFDObj`, `ooDBObj`, `ooContObj`, `ooObj`, whose instances are local representations of Objectivity / DB objects. The API also includes classes, notably `Connection` and `Session`, whose instances represent facets of the interaction between an application and Objectivity / DB. Together these two groups of classes provide a rich and flexible interface for developing applications that take advantage of the capabilities of Objectivity / DB and the Java programming language.

The relationships between instances of the classes in an application are illustrated in the following figure.


Connection
Interacts Through
Session1
...
SessionN
Owned by
ooFDObj
Owned By
Contained by
ooDBObj1
...
ooDBObjN
Contained by
ooContObj1
...
ooContObjN
Contained by
ooObj1
...
ooObjN
Key To Symbols
= Application Object
= Objectivity/DB Object

The following sections give a brief introduction to the classes shown in the previous figure. The classes are introduced in the order they would be used in implementing a simple application. The sections explain, and illustrate with examples, how to use the classes. A complete application that combines all the examples is described at the end of this chapter.

Application Development

Objectivity for Java comprises four packages:

- `com.objy.db`—exception and support classes.
- `com.objy.db.app`—general application classes and interfaces.
- com.objy.db.iapp—interfaces for persistence and events
- `com.objy.db.util`—persistence-capable collection classes and interfaces, and miscellaneous utility classes.

Your Objectivity for Java application must either import these packages or whatever classes and interfaces are used by your application.

All the constants used by Objectivity for Java general application classes are defined in the `oo` interface. You can implement this interface in any of your classes; doing so allows you to refer to the constants with their unqualified names, such as `READ`. If your class does not implement this interface, it must use fully-qualified constant names, such as `oo.READ`.

Connection Class

An Objectivity for Java application must first establish a *connection* with the federated database it wishes to work with. It does this by calling the `Connection.open` static method to obtain an instance of the `Connection` class.

EXAMPLE This code fragment opens a connection to a federated database called `Vrc` for read / write access. The `Connection.open` method throws two checked exceptions, which the application must catch.

```
Connection connection;

try {
   connection = Connection.open("Vrc", oo.openReadWrite);
}
catch (DatabaseNotFoundException exception) {
   System.out.println("Federated database \"Vrc\" not found.");
   return;
}
```

```
catch (DatabaseOpenException exception) {
   System.out.println("Federated database \"Vrc\" not open.");
   return;
}
```

Session Class

A *session* is an extended interaction between an application and a connected federated database. You must create a session before you perform any operations that affect the objects maintained by the connected federated database. A session is an instance of the `Session` class and is created using the `new` operator.

Sessions own the local representations of all Objectivity / DB objects accessed by an application. Objectivity for Java does not allow objects that belong to one session to interact with objects that belong to a different session. Sessions also provide the transaction services—`begin`, `commit`, `checkpoint`, and `abort`–that guarantee consistency between the local representations of Objectivity / DB objects and the objects in a federated database. All persistent operations on Objectivity / DB objects must be performed within a transaction.

EXAMPLE This code fragment creates a session, begins a transaction, processes Objectivity / DB objects, and commits the transaction.

```
Session session = new Session();
session.begin();    // Begin a transaction
... // Perform processing on Objectivity/DB objects
session.commit();   // Commit a transaction
```

Federated Database and Database Classes

A federated database is represented by an instance of the `ooFDObj` class. Each session automatically creates such a representation, which you obtain by calling the session's `getFD` method.

Databases are represented by instances of the `ooDBObj` class. There are no public constructors for this class. You create a new database using one of the `newDB` methods and retrieve an existing database using the `lookupDB` method of an instance of `ooFDObj`.

EXAMPLE This code fragment creates a session and retrieves its associated federated database. It then calls the federated database's `hasDB` method to check whether a database named `VehiclesDB` exists and, if so, retrieves it from the federated database. You should check whether the database exists before doing the lookup, because the lookup operation will throw an exception if a database with the name `VehiclesDB` does not exist in the federated database. If the database does not exist, the example creates a new database in the federated database.

```
ooFDObj vrcFD;
ooDBObj vehiclesDB;

Session session = new Session();
vrcFD = session.getFD();
session.begin();
if (vrcFD.hasDB("VehiclesDB"))
   vehiclesDB = vrcFD.lookupDB("VehiclesDB");
else {
   vehiclesDB = vrcFD.newDB("VehiclesDB");
   System.out.println("Created database \"VehiclesDB\".");
}
session.commit();
```

Container Classes

Objectivity for Java has two container classes: `ooGCContObj` and `ooContObj`. Instances of these classes represent, respectively, garbage-collectible containers and non-garbage-collectible containers. Both of these classes may be subclassed to provide application-defined behavior.

Containers are created using the Java `new` operator, and are initially transient. You can make a container persistent by adding it to a database with the `addContainer` method. This method requires you to explicitly specify a name for the container (null if the container is to be unnamed), whether the container is hashed, its initial size, and how much it should grow when it needs to accommodate more basic objects. You must make a container persistent before you call any methods of `ooContObj`. When you make a container persistent, the container is automatically locked for write by Objectivity / DB.

Once you add a container to a database, you can later retrieve it by name with the database's `lookupContainer` method. When you retrieve a container, it is returned unlocked; when you call the container's system-defined methods, however, Objectivity / DB will automatically lock it for you.

EXAMPLE This code fragment calls a database's `hasContainer` method to check whether a container named `VehiclesContainer` exists in the database, and, if so, retrieves it from the database. If the container does not exist, the example creates a new container in the database.

```
... // Get session and database
ooContObj vehiclesContainer;

session.begin();
if (vehiclesDB.hasContainer("VehiclesContainer"))
   vehiclesContainer =
               vehiclesDB.lookupContainer("VehiclesContainer");
else {
   vehiclesContainer = new ooContObj();
   vehiclesDB.addContainer(vehiclesContainer,
                           "VehiclesContainer", 0, 5, 10);
   System.out.println("Created container
                       \"VehiclesContainer\".");
}
session.commit();
```

Basic Object Classes

Defining Persistence-Capable Classes

All persistence-capable classes must be derived from the Objectivity for Java class `ooObj` or implement interface com.objy.iapp.Persistent.

EXAMPLE This example illustrates the definition of a persistence-capable `Vehicle` class. The field types for this example include persistent fields of the Java `String` class, primitive types, a reference to a persistence-capable `Fleet` class, and a transient field that contains the daily rate for renting the vehicle.

```
public class Vehicle extends ooObj {
   // Persistent fields
   protected String license;
   protected String type;
   protected int doors;
   protected int transmission;
   protected boolean available;
   protected Fleet fleet;
```

```
    // Legal values for transmission field
    public static final int MANUAL = 0;
    public static final int AUTOMATIC = 1;

    // Transient field
    protected transient int dailyRate;

    public Vehicle(String license, String type, int doors,
                   int transmission, int rate) {
        this.license = license;
        this.type = type;
        this.doors = doors;
        this.transmission = transmission;
        this.available = true;
        this.dailyRate = rate;
    }
    ... // Method definitions
}
```

Creating Persistent Objects

Persistent basic objects are created using the Java `new` operator and are initially transient. A basic object becomes persistent when it is stored or clustered into a persistent container. This can occur implicitly when you create an association between the transient object and a persistent object or name the transient object in the context of an Objectivity / DB object, or you can explicitly request that it be clustered into a specific container with the `cluster` method of any persistent object or database. Most methods defined on `ooObj` can be called only after the transient object has been made persistent.

EXAMPLE This code fragment creates a `Vehicle` object and explicitly clusters it in the container `vehiclesContainer`.

```
... // Get session and container
session.begin();
// Create transient Vehicle object
Vehicle vehicle = new Vehicle("ca1234", "H", 4,
                              Vehicle.STANDARD, 40);
// Make vehicle persistent by clustering it in a container
vehiclesContainer.cluster(vehicle);
// Make vehicle visible to other sessions
session.commit();
```

Reading and Writing Persistent Objects

Before you can read the persistent fields of a retrieved object, the object must be locked and its data must be fetched. The `fetch` method obtains a read lock on the object and its container before fetching the object's data from the database.

To ensure that your persistent objects are always in the correct state when they are read, you should define access methods to get the value of every persistent field. Each method should call the object's `fetch` method before returning the field's value. After the first call to fetch, subsequent calls of the fetch method within a transaction do not incur any overhead.

Similarly, the persistent fields of an object should be modified only if the application has a write lock on the object. Also, once the object is modified, it must be marked as such or the changes will not be written to the database when the session commits. The `markModified` method performs both operations. To ensure that persistent objects are modified correctly, you should define access methods for all operations that set the value of a persistent field. These methods should call the object's `markModified` method before changing the field.

EXAMPLE This example illustrates how to define access methods that safely fetch and modify the persistent fields of a `Vehicle` object.

```
public class Vehicle extends ooObj {

   ... // Variable declarations and constructor definition

   public String getLicense() {
      fetch();
      return this.license;
   }

   ... // Get methods for rest of fields

   public String toString() {
      fetch();
      StringBuffer buffer = new StringBuffer();
      if (this.available)
         buffer.append("AVAILABLE ");
      else
         buffer.append("RENTED ");
      buffer.append("License:" + this.license);
      buffer.append("Class:" + this.type);
      buffer.append("Doors:" + this.doors);
      if (this.transmission == MANUAL)
         buffer.append("MANUAL");
```

```
      else
         buffer.append("AUTOMATIC");
      buffer.append(" Rate:" + dailyRate);
      String strng = new String(buffer);
      return strng;
   }

   public void rentVehicle() {
      markModified();
      this.available = false;
   }

   ... // Set methods for rest of fields

   public boolean isAvailable() {
      fetch();
      return this.available;
   }
}
```

Retrieving Persistent Objects

Objectivity / DB provides a variety of mechanisms for retrieving objects from a federated database. Objects can be retrieved by looking them up by name, traversing object references or relationships, or by scanning storage objects or relationships.

The scan method supports searching for objects based on the value of one or more persistent fields.

EXAMPLE This example constructs a predicate over one or more of the persistent fields of the vehicle. It then initializes an iterator to return all the vehicles that satisfy the predicate and prints a string representation of all the matching vehicles.

```
// VehiclesContainer initialized elsewhere
ooContObj vehiclesContainer;

public void listVehicles(String license, String type,
         int doors, int transmission, boolean available) {
   String predicate =
         new String("license == \"\"" + license + "\" && " +
                  "type == \"\"" + type + "\" && " +
                  "doors >= " + doors + " && " +
                  "transmission == " + transmission + " && " +
                  "available == 1");
```

```
   Iterator itr;
   Vehicle vehicle;
   session.begin();
   itr = vehiclesContainer.scan("Vehicle", predicate);
   // Check whether iterator has found any matching vehicles
   if (!itr.hasNext()) {
      System.out.println("Vehicle satisfying predicate: "
                         + predicate + " not found.");
      session.abort();
      return;
   }
   // Retrieve each element from iterator
   while(itr.hasNext()) {
      vehicle = (Vehicle)itr.next();
      System.out.println(vehicle.toString());
   }
   session.commit();
}
```

To improve the performance of scans performed over specific persistent fields, you can define indexes using a container's `addIndex` or `addUniqueIndex` methods.

EXAMPLE This example defines a unique index that sorts the objects in the container `vehiclesContainer` according to the values in the vehicle's `license` field. Before adding the index to the container, the example checks whether an index named `VehiclesIndex` exists in the container, because the `addUniqueIndex` method will throw an exception if the container already has an index named `VehiclesIndex`.

```
session.begin();
if (!vehiclesContainer.hasIndex("VehiclesIndex"))
   vehiclesContainer.addUniqueIndex("VehiclesIndex",
                                    "Vehicle", "license");
session.commit();
```

Terminating an Application

When you have finished interacting with the connected federated database, you should call the `close` method of the connection object. Applications typically close a connection only once, immediately before exiting. However, you may reopen the connection by calling the `reopen` method.

NOTE The `close` method terminates all sessions in your application. After you have closed the connection, you should not try to use those sessions or any of the objects that belong to them. If you later reopen the connection, you must create one or more new sessions.

Example Application

The example described in this section combines all the code fragments in this chapter into a complete application. The example application is intended to be used by the agents and managers of a vehicle rental company. The operations supported are:

- Add a vehicle to the rental company's fleet of vehicles.
- Delete a vehicle from the fleet.
- List:
  - All the vehicles in the fleet.
  - Only the vehicles that satisfy a predicate.
- Rent a vehicle.
- Return a rented vehicle.

The application is implemented by four classes:

- `Vehicle` (see page 343), which implements a vehicle that can be rented and returned.
- `Fleet` (see page 341), which implements a fleet of vehicles.
- `VrcInit` (see page 361), which initializes the rental company database.
- `Vrc` (see page 348), which implements the interactive application for accessing the database of the vehicle rental company.

To execute this example, you need to:

1. Compile the files `Vehicle.java`, `Fleet.java`, `VrcInit.java`, and `Vrc.java` located in the `GettingStarted` subdirectory of the programming samples directory. See the *Installation and Platform Notes* for your operating system for the location of the samples directory for your platform.
2. Start an Objectivity / DB lock server.
3. Create a federated database called `Vrc` in the `GettingStarted` sample directory.
4. Execute `VrcInit` to initialize the federated database.

See the Objectivity/DB administration book for information on the tools used to create a federated database and start a lockserver.

When you execute `VrcInit`, the output will be:

```
Created database "VehiclesDB".
Created container "VehiclesContainer".

Added vehicle: License:CA1234 Class:G Doors:4 MANUAL Rate:40
Added vehicle: License:CA7654 Class:H Doors:4 AUTOMATIC Rate:40
```

Then you can execute `Vrc`. At the `Vrc` command prompt you can try the following operations (user input is in boldface):

1. Enter `list` and then enter `no` when it asks for a predicate. You should see the following output:

   ```
   Enter request (add, list, rent, return, quit)
   list
   Do you want to enter a search field?
   no
   Searching for ALL vehicles.
   AVAILABLE License:CA1234 Class:G Doors:4 MANUAL Rate:40
   AVAILABLE License:CA7654 Class:H Doors:4 AUTOMATIC Rate:40
   ```

2. Enter `rent` and then enter `ca7654`. You will see the output:

   ```
   Enter request (add, list, rent, return, quit).
   rent
   Enter license.
   ca7654
   Looking for vehicle with license: CA7654
   Rented: License:CA7654 Class:H Doors:4 AUTOMATIC Rate:40
   ```

3. Enter `list` and then enter `yes` when it asks if you want to enter a search field. Enter `available` and then `true` to list vehicles that are available to rent. Enter `no` when it asks if you want to enter another search field. The output should be:

   ```
   Enter request (add, list, rent, return, quit)
   list
   Do you want to enter a search field?
   yes
   Enter field to search (license, class, doors, transmission,
   or available)
   available
   What value of AVAILABLE do you want to search for?
   true
   ```

```
Do you want to enter another search field?
no
Searching for vehicles with available = TRUE
AVAILABLE License:CA1234 Class:G Doors:4 MANUAL Rate:40
```

4. Try adding a new vehicle and then listing the database.

ODMG Application Classes

An ODMG application interacts with a federated database through instances of the Objectivity for Java `Database` and `Transaction` classes. An ODMG application uses a database object to maintain a connection to the federated database and to provide naming of root objects; it uses a transaction object to provide transaction services.

Although database and transaction objects are sufficient for writing an ODMG application, certain Objectivity / DB policies and properties are still maintained by Objectivity for Java connection and session objects, and a session is still responsible for all the local representations of Objectivity / DB objects. Thus, when you open an ODMG database and create an ODMG transaction, corresponding Objectivity for Java connection and session objects are created automatically. Default values for connection and session properties are used, with the following implications for application behavior:

- All persistent objects are stored in the default database of the federated database. This is because the ODMG standard does not support the concept of a federation of databases.
- The session's default clustering strategy clusters all named roots and the objects they reference into a single container.

If these constraints and default policies are not appropriate for your application, Objectivity for Java allows you to retrieve the connection and session from their database and transaction, and change the policies and properties. See Chapter 3, "ODMG Application Objects" for further information.

2

Application Objects

The interaction between an Objectivity for Java application and an Objectivity / DB federated database is controlled by a federated database *connection* and one or more *sessions*.

A federated database *connection* establishes an association between an application and a particular federated database. The connection maintains global policies that govern the interaction between the application and the connected federated database.

A *session* is an extended interaction between an application and a connected federated database. A session represents a set of objects under transactional control; the session owns the local representations of all Objectivity / DB objects accessed by an application, and provides the transaction services—begin, checkpoint, commit and abort—that guarantee consistency between the local representations and the objects in a federated database.

In This Chapter

Federated Database Connections

A federated database *connection* establishes an association between an application and a particular federated database. A federated database connection is an instance of the `Connection` class. Before you can interact with a federated database, you must first open a connection between your application and the federated database.

Opening and Closing a Connection

You open a connection by obtaining an instance of the `Connection` class with the static method `Connection.open` of the `Connection` class. This method returns the single instance of `Connection` allowed in an application. You can obtain that instance by calling the `Connection.current` static method.

The `open` method requires that you specify the federated database's boot file name and the connection's open mode. The connection's open mode initializes and limits the open mode of any sessions created by the application. A connection open mode of read / write permits session open modes of read / write or read-only; a connection open mode of read-only permits the session open mode read-only. You can change the open mode of a connection with the `setOpenMode` method.

When you have finished interacting with the connected federated database, you should close the connection with the `close` method of the connection object. Applications typically close a connection only once, immediately before exiting. However, you may reopen the connection instance by calling the `reopen` method.

NOTE The `close` method terminates all sessions in your application. After you have closed the connection, you should not try to use those sessions or any of the objects that belong to them. If you later reopen the connection, you must create one or more new sessions.

Connection Policies

A connection object has a number of policies that govern the interaction between the application and the connected federated database:

- A thread policy that controls interaction between the threads and sessions in the application.
- A schema policy that controls what types of modifications the application can make to the schema of the connected federated database.
- A predicate-scan autoflush policy that controls whether Objectivity for Java writes all changed objects to the cache (without commit) prior to performing a predicate scan.

- An AMS usage policy that controls the application's use of the Advanced Multithreaded Server (AMS). By default, AMS is used for remote data access if it is available; you can change the policy by calling the `setAMSUsage` method.

Thread Policy

An Objectivity for Java application can use multiple Java threads to execute concurrent *persistent* operations. However, *persistent* operations performed in a single session are treated serially by Objectivity/DB. To obtain truly concurrent operations, each of several concurrent threads must have its own session. Note that multiple threads can still access and manipulate the basic objects concurrently, as with any other Java objects. Objectivity for Java only serializes persistent operations and not the access to object values.

The interaction between threads and sessions is governed by the session's thread policy. A session's thread policy is initialized by the thread policy of the connection.

A thread policy can be restricted or unrestricted. When you open a connection, the thread policy of the connection is set to restricted by default; you can change the policy at any time by calling the connection's `setThreadPolicy` method. If you change the thread policy of a connection, the new thread policy will initialize the thread policy of sessions that are created after you changed the policy; the thread policies of existing sessions are not changed. Note that the thread policy of a session may be changed independently from the connection.

Schema Policy

A connection's schema policy controls what types of modifications the application can make to the schema of the federated database. By default, your application may add new class descriptions to the schema of the connected federated database and modify existing class descriptions in the schema. Changes to this policy are described in "Schema Policies" on page 260.

Predicate-Scan AutoFlush Policy

A connection's predicate-scan autoflush policy controls whether Objectivity for Java writes all changed objects to the cache (without commit) prior to performing a predicate scan. When autoflush is enabled, the results of the predicate scan will include any changes to values in any persistent fields. When autoflush is disabled, it is possible that the results of a predicate scan set may be out of sync with recently changed objects. By default, predicate-scan autoflush is enabled. You can change the policy by calling the `setPredicateScanAutoFlush` method.

Sessions

A *session* is an extended interaction between an application and a connected federated database. A session is an instance of the `Session` class. You must create a session before you perform any operations that affect the objects maintained by the connected federated database.

Sessions own the local representations of all Objectivity / DB objects accessed by an application. Objectivity for Java does not allow persistent objects that belong to one session to interact with persistent objects that belong to a different session. In addition to owning the local representations of Objectivity / DB objects, sessions also provide the transaction services that guarantee consistency between the local representations and the objects in a federated database.

Your application can create multiple sessions, each corresponding to a particular subtask that your application performs. Sessions serve to isolate the operations performed in the different subtasks of an application. If more than one session uses a given Objectivity / DB object, each session has its own local representation of that object.

A session also maintains the following policies and properties:

- Policies for managing locks.
- A policy that determines where transient objects are clustered when they are made persistent.
- A policy for interacting with Java threads.
- Policies for managing and using indexes.
- Objectivity / DB cache properties.

Creating and Terminating a Session

A session object is created simply by calling one of the two `Session` constructors provided. One constructor allows you to specify values for two Objectivity / DB cache properties; the other constructor takes no parameters and creates a cache with default values.

When a session is created, the session automatically creates a local representation of the federated database, a clustering strategy, and an ODMG transaction.

When a session object is no longer needed, you should release all the resources associated with the session, including its ODMG transaction, federated database, and any other Objectivity / DB objects that belong to the session, by calling its `terminate` method. Objectivity for Java terminates all sessions when a connection is closed. If a session is not explicitly terminated, it will be terminated as part of finalization by the Java garbage collector; however, explicit termination is recommended.

Once a session has been terminated:

- The *only* method that can be called on the session is `isTerminated`.
- All the local representations of Objectivity / DB objects become dead objects; the objects are no longer valid for persistent operations.

Sessions and Objectivity/DB Objects

Sessions are responsible for the local representations of Objectivity / DB objects accessed by an application. A session:

- Owns the local representations of Objectivity / DB objects.
- Prevents its objects from interacting with objects belonging to other sessions.
- Ensures that it has a single local representation for each Objectivity / DB object.

Object Ownership

Sessions *own* the local representations of all Objectivity / DB objects accessed by an application. The local representation of:

- The connected federated database is owned by the session that created it.
- A database is owned by the session that was in a transaction when the database was retrieved or created.
- A container is owned by the session that was in a transaction when the container was retrieved or made persistent.
- A basic object is owned by the session that was in a transaction when the object was retrieved or made persistent.

You can retrieve a local representation of the federated database with the session's `getFD` method. You can traverse to all other local representations of Objectivity / DB objects by starting from the session's federated database; however, you cannot retrieve any other local representation directly from the session itself. On the other hand, you can retrieve the session owning any given local representation with the object's `getSession` method.

Object Isolation

A session isolates the objects that it owns. If more than one session accesses a given Objectivity / DB object, each session uses its own local representation of that object.

Objectivity for Java does not allow the local representation of an object that belongs to one session to interact with the local representation of an object that belongs to a different session. For example, if `Session1`'s representation of a database is `D1` and `Session2`'s representation of a basic object is `B2`, you may not make `B2` a named root in `D1`. Instead, you must create another local representation (`D2`) of that same database in `Session2`. You can then make `B2` a named root in

D2, because both B2 and D2 belong to Session2. You can check if two objects, Obj1 and Obj2, belong to the same session by testing whether

```
Obj1.getSession() == Obj2.getSession()
```

is true.

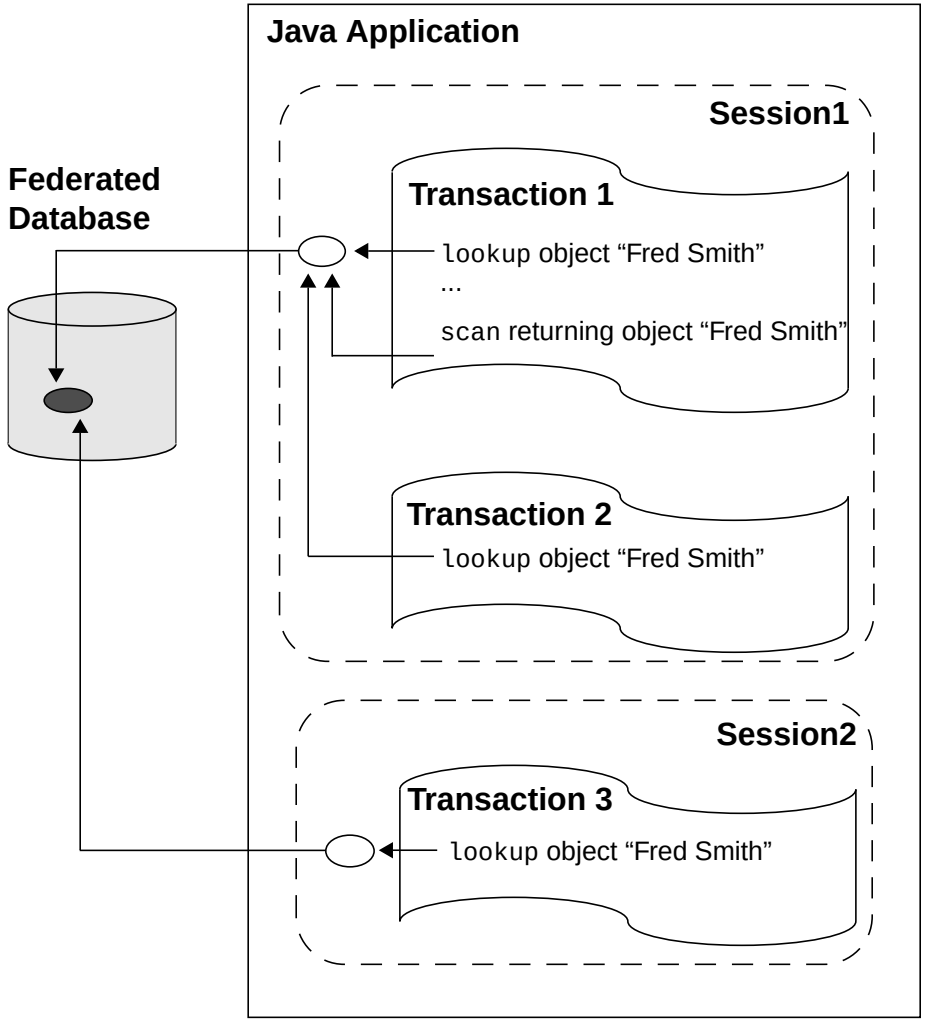
Any attempt to violate the restriction against crossing session boundaries causes an IsolationException to be thrown. Some violations can be detected immediately, while other violations are detected when you write the objects to the database. A violation is detected immediately when a persistent operation is performed on an object. Thus, in the example above, passing B2 to the bind method of D1 causes an immediate exception. On the other hand, you might set a persistent field of an object that belongs to one session to reference an object that belongs to a different session. When the referencing object is written to the database (usually at transaction commit), the violation will be detected and an exception thrown.

Object Identity

It is possible for an application to have more than one reference to the same object in the database. For example, an application could get one reference by looking up the object by name and another by scanning the container in which the object is stored. A session ensures that it has a *single* local representation of any particular object. If multiple references to an object are obtained within a session, all the references will point to the same local representation of the object. On the other hand, if a second session gets a reference to the same object, that reference will point to a different local representation of the object.

For example, in Transaction1 in the following figure, two different method calls retrieve the same object. The lookup method retrieves the object from the federated database and returns a local representation for it; the scan method returns the local representation created by the first call. The result is the same for the two method calls during different transactions of the same session. Thus, when Transaction2 looks up the same object, the local representation is the same representation that was retrieved by the earlier lookup.

Transaction3 looks up the same Objectivity / DB object. Because this is a transaction of Session2, this lookup retrieves the object from the database again and creates a different local representation for Session2.



Key To Symbols

= Session

= Objectivity/DB object

= Transaction

= Java object

Transactions

Sessions provide the transaction services that guarantee consistency between the local representations of objects in an application and the objects in a federated database. An application must be *within a transaction* to perform an operation that creates, reads, modifies, or deletes an Objectivity / DB object.

Transactions also group operations so that they appear as a single, indivisible operation. At the end of a transaction, the application is guaranteed that either all or none of the operations were performed. Thus, a federated database cannot be left in an inconsistent state that might result if some, but not all, of the operations had been performed.

A transaction is effectively a subsection of an application, the extent of which is designated by four operations: begin, commit, checkpoint, and abort. The application is said to be *within a transaction* after it *begins* and until it commits or aborts a transaction. While the application is within a transaction, it can obtain local representations of and perform processing on the objects for which it has the appropriate access rights. When the application *commits* the transaction, all modifications to the objects are saved to the federated database. If the application *aborts* the transaction instead of committing it, the changes are discarded (rolled back), leaving the federated database in the logical state it was in before the transaction started. Both operations signify the end of the transaction. The local representations of any Objectivity / DB objects are invalidated and any locks on the objects are released. You can also *checkpoint* a transaction, which saves modifications to the federated database but retains the local representations of objects and their locks.

Beginning a Transaction

You begin a new transaction by calling the `begin` method of a session. After `begin` is called, the state of the session is said to be *open* or *in a transaction* and the transaction is *in progress*. Once a session is in a transaction, you can work with objects that belong to that session.

Committing a Transaction

During a transaction, Objectivity / DB records all changes made to Objectivity / DB objects, but does not save the changes to the federated database. As a consequence, all such changes are visible only to the transaction in which they were made. To save changes and make them visible to other transactions, you can take either of the following actions:

- Commit the transaction.
- Checkpoint the transaction.

You commit a transaction by calling the session's `commit` method. Committing a transaction saves all newly created or modified Objectivity / DB objects to the federated database and:

- Ends the transaction and changes the state of the session to closed.
- Releases any locks acquired in the course of the transaction.
- Updates all applicable indexes according to the session's index mode.

The local representations of Objectivity / DB objects may no longer be consistent with the objects in the federated database and invoking a system-defined operation on an Objectivity / DB object will throw a `TransactionNotInProgressException`. However, the local representation is still owned by the session, and can be reused in subsequent transactions. Note that any values read into the Java objects during the transaction remain whether the transaction was committed, checkpointed, or aborted. In the cases of commit and abort, the values of these objects are still available but may no longer correspond to the objects' values in the database. Since checkpoint retains locks, the values are always in sync with those in the database and no other session can obtain a lock that would allow the objects to be modified.

None of the actions performed when a transaction is committed can be undone.

If you want to save the results of a computation, but retain access to objects, you can checkpoint the transaction by calling the session's `checkpoint` method. If you are finished updating the objects, the downgrading `checkpoint` method permits you to downgrade your session's write locks to read locks, thus permitting other processes to gain read access to those objects.

If many objects have been modified during a lengthy transaction, the cost of transferring all the data to the database at commit or checkpoint time can be high. During such transactions, you can call the `flush` method of the federated database, database, or container, which transfers the data of all modified objects of that storage object to the database but does not make the changes permanent or available to other processes. Flushing a federated database also frees up space in Java memory.

Aborting a Transaction

You abort a transaction by calling the session's `abort` method. When you abort a transaction, any changes are discarded (rolled back), leaving the federated database in the logical state it was in before the transaction started. The only exception to this rule is that deleting a database cannot be rolled back by aborting a transaction. In common with the commit operation, aborting a transaction also:

- Ends the transaction and changes the state of the session to closed.
- Releases any locks acquired in the course of the transaction.
- Invalidates the local representations of Objectivity / DB objects.

If a transaction is ended either by stopping the process within a debugger or by killing the process, the locks acquired during the transaction are not released. To clear these locks, you can enable automatic recovery for the session or use the `oocleanup` administration tool to perform manual recovery.

You can commit, checkpoint, or abort a transaction even if no Objectivity / DB objects were created or modified. The `TransactionNotInProgressException` is thrown, however, if you call any of these methods when the session is not in a transaction.

EXAMPLE This code fragment illustrates a sequence of operations for beginning and ending transactions.

```
// Open a federated database connection
Connection connection = Connection.open("bootFile",
                        oo.openReadWrite);
// Create session
Session session = new Session();
session.begin();  // Begin transaction
... // Perform processing on Objectivity/DB objects
session.checkpoint(); // Checkpoint session
... // Perform processing on Objectivity/DB objects
session.commit(); // End transaction
...
session.begin();  // Begin transaction
... // Perform more processing on objects that belong
   // to this session and other Objectivity/DB objects
session.commit(); // End transaction
```

Transaction Design Guidelines

A transaction can contain any number of persistent operations and other application actions. Deciding how to break up a given task into separate transactions involves performance, concurrency, and usability trade-offs. Here are some guidelines for making such decisions:

- Keep important transactions short. A short transaction is less likely to be aborted while waiting for a lock or for any other reason. If it is aborted, less work will be undone than with a long transaction. For example, consider a data-entry operator who has to fill out a lengthy on-line questionnaire while interviewing a customer. If the application waits until the questionnaire is completed before ending the transaction and committing the changes, the entire interview might have to be repeated if a problem prevents the data from being committed. As an alternative to short transactions, you can checkpoint a long transaction periodically, causing new and updated objects to be

committed without releasing their locks. You can also downgrade write locks to read locks at such junctures.

- Keep transactions involving high-traffic objects short. When a transaction involves an Objectivity/DB object that is frequently updated, minimize the waiting time for competing applications by keeping the transaction short. If only one user at a time needs to update an object, consider using an MROW session for applications that read it.
- Use long transactions when slow network access is involved. Each time a transaction ends, locks are released on the Objectivity/DB objects that were accessed in that transaction. If some or all of those objects are accessed in the next transaction, those locks must be reacquired, which involves network traffic to and from the lock server. In addition, objects may need to be refreshed each time a new transaction begins, because they may have been updated by a competing application between transactions. This refreshing activity involves transmitting the same objects across the network from the data server. Combining a series of short transactions into one long transaction can reduce repetitive lock and object refresh activity. Again, checkpointing and lock downgrading can be used to make changes visible incrementally.
- For very long transactions, flush changes incrementally. Changes accumulate in the local cache until the transaction is committed, so a very long transaction involving many objects or large objects can overflow the cache. When the cache overflows, objects that would otherwise be quickly accessible in local memory must be fetched from the data server, possibly across a slow network. Flushing the federated database writes all new and updated objects to the database without committing those changes, thus freeing space in the cache.

Session Properties

Sessions have a number of methods that get and set the value of properties that govern the interaction of sessions with Objectivity/DB. The following table lists the methods that set properties and notes when they can be called.

Instantiation	`Session(cacheInitialPages, cacheMaximumPages)`

Outside a Transaction	`setMrowMode`
	`setWaitOption`
	`setIndexMode`
	`setRecoveryAutomatic`

Anytime	`setUseIndex`
	`setThreadPolicy`
	`setLargeObjectMemoryLimit`
	`setHotMode`
	`setOpenMode`
	`setClusterStrategy`

Cache Properties

The Objectivity / DB cache is a portion of the process' virtual memory that is managed by Objectivity / DB for the purpose of providing fast access to persistent objects. Each session owns a portion of this cache for its exclusive use.

Each session's portion of the cache consists of buffer pages. Buffer pages are the same size as the storage pages in the federated database; you specify the storage page size when you create a federated database. Buffer and storage pages are the minimum units of transfer to and from disk and across networks. That is, when you access a persistent object, Objectivity / DB reads the storage page(s) containing the object into the cache. Conversely, when you commit a session containing a new or updated object, Objectivity / DB writes the buffer page(s) that contain the object as storage page(s) on disk.

The Objectivity / DB cache has separate sets, or pools, of buffer pages for handling small objects (objects that are smaller than a storage page) and large objects (objects that span multiple storage pages). In general, most persistent data is small, with the exception of very large arrays (including strings and data structures that support large relationships). The maximum size of the Objectivity / DB cache is the sum of the maximum sizes of the two buffer pools.

The number of buffer pages is set when the session is created. If the constructor with no parameters is used, the number of initial and maximum pages is set to the default values of 20 and 200 respectively.

Caching Small Objects

The `cacheInitialPages` parameter of the session constructor is used to set the initial number of buffer pages to be allocated for the small-object buffer pool. As small persistent objects are created or fetched, Objectivity / DB may add more buffer pages to this pool, up to the maximum number of pages specified by the `cacheMaximumPages` parameter of the session constructor. When this maximum is reached, Objectivity / DB swaps out unneeded buffer pages before adding new ones.

Caching Large Objects

The `cacheInitialPages` parameter also specifies the initial number of buffer pages to be allocated for the large-object buffer pool. When a large persistent object is created or fetched, Objectivity / DB reads just the object's header page into this pool and caches the object's remaining pages as a dynamically-allocated block of virtual memory. When the number of header pages in the buffer pool reaches the maximum specified by the `cacheMaximumPages` parameter of the session constructor, Objectivity / DB swaps out unneeded large objects before adding new ones.

You set the limit on the total amount of memory that can be dynamically allocated for large objects with the method `setLargeObjectMemoryLimit`. When this limit is reached, Objectivity / DB attempts to swap out the pages of large objects before opening additional large objects. However, if Objectivity / DB cannot find enough large objects to swap out, it ignores the specified limit and allocates additional pages as needed. Thus, the limit you specify is a soft limit that affects the amount of swapping performed on behalf of large objects.

Automatic Recovery

The automatic recovery property specifies whether automatic recovery on the local host should be performed when an application first interacts with a federated database. If automatic recovery is enabled, Objectivity / DB will roll back any incomplete transactions started by applications running on the same client host as the application. The automatic recovery property is disabled by default and can be set with the `setRecoveryAutomatic` method. The recovery attempt occurs when a transaction is started (by calling the `begin` method).

Checking whether automatic recovery is needed can be a time-consuming process. If you enable automatic recovery, it will be disabled immediately after the transaction ends to prevent the recovery process from being repeated for each subsequent transaction.

Clustering Strategy

When persistent objects are created, they are initially transient. They become persistent when assigned a storage location in the federated database. This can occur explicitly, when an object's `cluster` method is called, or implicitly, as a result of establishing an association with an object that is already persistent.

All operations that implicitly cluster objects call a session's clustering strategy to determine how the object should be clustered. You install a clustering strategy by calling the session's `setClusterStrategy` method. If you do not install a clustering strategy, a default strategy is installed.

EXAMPLE This code fragment shows how to install the clustering strategy defined in "Defining a Clustering Strategy" on page 242.

```
Session session = new Session();
ContainerPoolStrategy containerPoolStrategy =
   new ContainerPoolStrategy();
ClusterStrategy oldClusterStrategy =
   session.setClusterStrategy(containerPoolStrategy);
```

Locking Properties

The open and MROW modes of a session affect the type of locks a session can obtain for the objects it owns.

A session's open mode is a limit on the kinds of locks you can obtain for any objects contained within the federated database. The open mode can be either read / write, which allows objects to be locked for read or write, or read-only, which allows objects to be locked for read.

A session's open mode is set with the `setOpenMode` method. If this method is not called on a session, its default open mode is the same as the open mode of the connection to the federated database.

NOTE The open mode of the session is limited by the open mode of the connection to the federated database. If you try to set the session open mode to read / write when the connection open mode is read-only, an `ObjyRuntimeException` will be thrown when you begin the transaction.

The MROW mode relaxes the (default) exclusive concurrent access policy of a session. When MROW is enabled, a session is permitted to obtain a read lock for a container that is write locked by another session. You set the MROW mode with the `setMrowMode` method. See "Multiple Readers, One Writer (MROW) Policy" on page 82 for information on how MROW works.

The lock waiting property, set by the `setWaitOption` method, determines whether, and for how long, a session will wait to obtain the locks it needs to proceed if another process has already obtained a conflicting lock. By default, sessions do not wait for locks. See "Lock Waiting" on page 86 for information on how lock waiting works.

NOTE MROW-enabled sessions always wait for locks; any lock time-out value set with the `setWaitOption` method is ignored by an MROW session.

Indexing Properties

Sessions maintain two properties related to indexing: an index usage policy and an index mode.

The index usage policy controls whether indexes should be used when performing predicate scans. By default, indexes are not used; you can change the policy by calling the `setUseIndex` method.

The index mode specifies how and when indexes are updated relative to when indexed objects are updated. By default, indexes are updated when you commit the transaction in which indexed objects are modified. You can change the index mode by calling the `setIndexMode` method.

Chapter 13, "Optimizing Searches With Indexes," discusses how to create and use indexes and the issues you need to consider when designing indexes.

Sessions and Threads

An Objectivity for Java application can use multiple Java threads to execute concurrent persistent operations. Multiple threads may share a single session: one thread can even begin a session's transaction and another can commit or abort the session's transaction. Persistent operations performed in a single session are treated serially by Objectivity for Java. Thus, if truly concurrent operations are desired, each thread must have access to its own session.

The interaction between a session and a thread is governed by the session's *thread policy*. When the session is created, its thread policy is set to the current thread policy of the connection. You can change a session's thread policy at any time by calling the session's `setThreadPolicy` method. You can choose between two thread policies: restricted and unrestricted.

Restricted Thread Policy

The *restricted* thread policy requires that each thread use a particular session while it is interacting with a database. This policy is particularly suitable for applications, such as servers, that spawn a thread to handle each incoming request, and need to enforce a strict separation between the objects accessible while each request is being serviced.

The restricted thread policy is enforced by requiring that a thread be *joined to* a session. The `NotJoinedException` will be thrown if a thread that is not joined to the session:

- Calls the `begin`, `checkpoint`, `commit`, `abort`, or `isOpen` methods.
- Performs any persistent operation on any object that belongs to the session.
- Calls the `leave` method of the session.

This requirement holds both for threads that are created explicitly by the application and for threads that are created implicitly by a library imported by the application.

At any given time, a thread can be joined to only *one* session. A thread is automatically joined to a session it creates. In addition, you can call a session's `join` method to join a thread with an existing session. When a thread joins a session (by creating it or by an explicit call to the session's `join` method), the thread automatically leaves any session it was already joined to. You can also call

a session's `leave` method to make a thread leave a session. It is recommended that you call the `leave` method at some point before terminating, though it is not required. To obtain the session for the current thread, you call the `Session.getCurrent` static method.

EXAMPLE This code fragment illustrates what happens when a thread creates and uses more than one session.

```
// Thread joined with session1
Session session1 = new Session();
session1.begin();
…
session1.commit();
// Thread joined with session2 leaves session1
Session session2 = new Session();
session2.begin();
…
session2.commit();
```

If both sessions are created at the beginning of the sequence of operations, you must explicitly call the `join` method:

```
// Thread implicitly joined with session1
Session session1 = new Session();
// Thread implicitly joined with session2
Session session2 = new Session();
session1.join(); // Rejoin session1 and leave session2
session1.begin();
…
session1.commit();
session2.join(); // Rejoin session2 and leave session1
session2.begin();
…
session2.commit();
```

The `MultipleThreadsSP` programming example (see page 366) simulates a server application handling requests to look up or list root objects in a database. The server creates a new thread to handle each request. Instead of having each thread create a new session, the example shares a pool of sessions among the threads. Access to the pool of sessions is synchronized. The session pool is implemented in the `SessionPool` class (see page 372). The example also illustrates the use of a restricted thread policy: before a thread uses a session that it retrieves from the pool, it executes a join operation.

This example can be executed after you compile the files and create a federated database named `"Objects"` in the `Application` subdirectory of the samples directory.

Unrestricted Thread Policy

The requirement that a thread running a transaction be joined with the session providing the transaction services becomes unworkable if the application, or a library used by the application, makes extensive and implicit use of Java's lightweight thread capability. Consider an event handler that invokes one of a set of methods when an event occurs. The event handler may or may not spawn a new thread to handle each event. In such an environment, it would be very difficult to determine whether and when a given thread should attempt to join with the session in charge of the database access.

The *unrestricted* thread policy is designed for this type of application. The only requirement for this policy is that when a persistent operation is messaged, the object's owner session must be in an active transaction.

3

ODMG Application Objects

Objectivity for Java supports ODMG applications with its `Database` and `Transaction` classes. An ODMG application uses an *ODMG database* to interact with an Objectivity / DB federated database and to provide naming of root objects; it uses *ODMG transaction*s to provide transaction services.

In This Chapter

ODMG Applications

An *ODMG application* can perform the following operations:

- Open a database.
- Create a transaction.
- Begin a transaction.
- Operate on graphs of persistent objects:
  - Create an object and make it a named root. Make persistent all objects transitively referenced from the named root.
  - Look up a named root and traverse to objects transitively referenced from the named root.
- End the transaction.
- Close the database.

Although ODMG database and transaction objects are sufficient for writing an ODMG application, certain Objectivity/DB policies and properties are still maintained by Objectivity for Java connection and session objects, and a session is still responsible for all the local representations of Objectivity/DB objects. Thus, when you open an ODMG database and create an ODMG transaction, corresponding Objectivity for Java connection and session objects are created automatically. Default values for session and connection properties are used, with the following implications for application behavior:

- All persistent objects are stored in the default database of the federated database. This is because the ODMG standard does not support the concept of a federation of databases.
- The session's default clustering strategy clusters all named roots and the objects they reference into a single container.

If these constraints and default policies are not appropriate for your application, Objectivity for Java allows you to retrieve the connection and session from their database and transaction, and change the policies and properties. In particular, you can redefine the clustering strategy of a session to:

- Use multiple containers. Because Objectivity/DB locks at the container level, this would probably be the minimum amount of storage reconfiguration you would want to do. Otherwise, the concurrency of your application would be severely restricted. See Chapter 4, "Locking and Concurrency," for more information about Objectivity/DB locking mechanisms and policies.
- Use multiple databases.
- Use one database or container pool for object graphs of a particular type, and a different database or container pool for object graphs of a different type.

Databases

An instance of the `Database` class represents an ODMG database. The ODMG standard does not include the concept of a federation of databases; as a consequence an ODMG database corresponds to an Objectivity/DB federated database in which all persistent objects are stored in the default database.

Opening and Closing a Database

An ODMG application opens an ODMG database by calling the `Database.open` static method of the `Database` class. The method creates the ODMG database object that corresponds to the connected federated database and a connection object that interacts with the federated database.

An Objectivity for Java application can have only *one* instance of this class. You can obtain that sole instance by calling the `Database.current` static method.

NOTE The connection object for an ODMG application uses default values for all its connection policies. If these default policies are not appropriate for your application, you can retrieve the database's connection with the `getConnection` method and call connection methods to change the policies.

When you have finished interacting with an ODMG database, you should call the `close` method of the ODMG database object. An ODMG application can close its database only once; it typically does so immediately before exiting. After the `close` method has been called:

- The *only* method that can be called on the database is `isOpen`.
- All ODMG transaction objects and all persistent objects are dead objects, that is, the objects are no longer valid for persistent operations. Any attempt to perform an operation on a dead object throws an `ObjectIsDeadException`.

Managing Named Roots

An ODMG database object provides methods for managing named roots:

- The `bind` method makes an object a named root. When you commit the transaction, the named root and all other objects in its object graph are stored persistently in the database.
- The `lookup` method retrieves a named root. You can then traverse to the objects in the object graph of the named root through the persistent fields that reference those objects. See "Finding Objects in a Graph" on page 225 for further information.

  Once you have retrieved a persistent object, you can read and modify its persistent fields. See "Persistent Objects" on page 161 for further information.
- The `unbind` method deletes a named root. If the named root is not in the object graph of another named root, the named root and other objects in its object graph will be physically removed from the database when you run the `oogc` administration tool.

Transactions

Transactions guarantee consistency between the local representations of objects in an application and the objects in a federated database. An application must be *within a transaction* to perform an operation that creates, reads, modifies, or deletes an Objectivity / DB object.

Transactions also group operations on one or many Objectivity / DB objects so that they appear as a single, indivisible operation. At the end of a transaction, the

application is guaranteed that either all or none of the operations were performed. Thus, a federated database cannot be left in an inconsistent state that might result if some, but not all, of the operations had been performed.

A transaction is effectively a subsection of an application, the extent of which is designated by four operations: begin, commit, checkpoint, and abort. The application is said to be *within a transaction* after it *begins* and until it commits or aborts a transaction. While the application is within a transaction, it can obtain local representations of, and perform processing on, the objects for which it has the appropriate access rights. When the application *commits* the transaction, all modifications to the objects are saved to the federated database. If the application *aborts* the transaction instead of committing it, the changes are discarded (rolled back), leaving the federated database in the logical state it was in before the transaction started. Both operations signify the end of the transaction. The locks on any Objectivity / DB objects are released, the local representations of Objectivity / DB objects may no longer be consistent with the objects in the federated database, and invoking a system-defined operation on an Objectivity / DB object will throw a `TransactionNotInProgressException`. You can also *checkpoint* a transaction, which saves modifications to the federated database but retains the local representations of objects and their locks.

Transaction Operations

Creating a Transaction

In an ODMG application, transaction services are provided by instances of the `Transaction` class. When you create an ODMG transaction object, the transaction's session is created automatically. The session, in turn, creates a local representation of the connected federated database and a clustering strategy.

The transaction and federated database objects *belong to* their associated session object; the local representation of every Objectivity / DB object that you retrieve or create while a particular transaction object is open also belongs to that transaction's session; see "Object Ownership" on page 51. Objectivity for Java does not allow a transaction object to interact with any object that belongs to a different session; see "Object Isolation" on page 51.

NOTE The session that owns an ODMG transaction object uses default values for all its session properties. If the default behavior is not appropriate for your application, you can retrieve the transaction's session with the `getSession` method and call session methods to change the properties.

Your application can create multiple transactions, each corresponding to a particular subtask that your application performs. Transactions serve to isolate the operations performed in the different subtasks of an application. If more than one transaction uses a given Objectivity / DB object, each transaction's session has its own local representation of that object. See "Object Identity" on page 52 for further information.

Beginning a Transaction

You begin a new transaction by calling the `begin` method of a transaction object. After `begin` is called, the state of the transaction object is said to be *open* and a transaction is *in progress.*

Committing a Transaction

During a transaction, Objectivity / DB records all changes made to Objectivity / DB objects, but does not save the changes to the federated database. As a consequence, all such changes are visible only to the transaction in which they were made. To save changes and make them visible to other transactions, you can take either of the following actions:

- Commit the transaction.
- Checkpoint the transaction.

You commit a transaction by calling the transaction object's `commit` method. As well as saving all newly created or modified Objectivity / DB objects to the federated database, committing a transaction:

- Ends the transaction and changes the state of the transaction object to closed.
- Releases any locks acquired in the course of the transaction.
- Updates all applicable indexes according to the index mode of the transaction's session.

The local representations of Objectivity / DB objects may no longer be consistent with the objects in the federated database and invoking a system-defined operation on an Objectivity / DB object will throw a `TransactionNotInProgressException`. The local representation is still owned by the transaction object's session, however, and can be reused in subsequent transactions. Note that any values read into the Java objects during the transaction remain whether the transaction was committed, checkpointed, or aborted. In the cases of commit and abort, the values of these objects are still available but may no longer correspond to the objects' values in the database. Since checkpoint retains locks, the values are always in sync with those in the database and no other session can obtain a lock that would allow the objects to be modified. None of the actions performed when a transaction is committed can be undone.

If you want to save the results of a computation, but retain the open objects and locks, you can checkpoint the transaction using the `checkpoint` method. If you are finished updating the objects, the downgrading `checkpoint` method permits you to downgrade the locks you are holding to read access, thus permitting other processes to gain read access to those objects.

If many objects have been modified during a lengthy transaction, the cost of transferring all the data to the database at commit or checkpoint time can be high. During such transactions, you can call the `flush` method of the federated database, database, or container object, which transfers the data of all modified objects of that storage object to the database but does not make the changes permanent or available to other processes. Flushing a federated database also frees up space in the Java memory. To perform this operation, you must first retrieve the transaction's session with the `getSession` method and then retrieve the federated database object from the session with the `getFD` method.

Aborting a Transaction

You abort a transaction by calling the transaction object's `abort` method. When you abort a transaction, any changes are discarded (rolled back), leaving the federated database in the logical state it was in before the transaction started. The only exception to this rule is that deleting a database cannot be rolled back by aborting a transaction. In common with the commit operation, aborting a transaction also:

- Ends the transaction and changes the state of the transaction object to closed.
- Releases any locks acquired in the course of the transaction.
- Invalidates the local representations of Objectivity / DB objects.

If a transaction is ended either by stopping the process within a debugger or by killing the process, the locks acquired during the transaction are not released. To clear these locks, you can enable automatic recovery for the transaction's session, or use the `oocleanup` administration tool to perform manual recovery.

You can commit, checkpoint, or abort a transaction even if no Objectivity / DB objects were created or modified. The `TransactionNotInProgressException` is thrown, however, if you call any of these methods on a closed transaction.

EXAMPLE This code fragment illustrates a sequence of operations for beginning and ending transactions.

```
// Open a federated database connection
Database database = Database.open("bootFile", oo.openReadWrite);
Transaction tx = new Transaction();
tx.begin();  // Begin transaction
... // Perform processing on Objectivity/DB objects
```

```
tx.checkpoint(); // Checkpoint transaction
... // Perform processing on Objectivity/DB objects
tx.commit(); // End transaction
...
tx.begin(); // Begin transaction
... // Perform more processing on objects that belong
  // to this transaction's session and other Objectivity/DB
objects
tx.commit(); // End transaction
```

Transactions and Threads

An Objectivity for Java application can use multiple Java threads to execute concurrent persistent operations. Multiple threads may share a single transaction: one thread can even begin a transaction and another can commit or abort it. Persistent operations performed in a single transaction are treated serially by Objectivity for Java. Thus, if truly concurrent operations are desired, each thread must have access to its own transaction.

Before a thread can use a transaction, the ODMG specification requires that it be *joined* to that transaction. The `NotJoinedException` will be thrown if a thread that is not joined to the transaction:

- Calls the `begin`, `checkpoint`, `commit`, `abort`, or `isOpen` methods.
- Performs any persistent operation on any object.
- Calls the `leave` method of the transaction.

This requirement holds both for threads that are created explicitly by the application and for threads that are created implicitly by a library imported by the application.

At any given time, a thread can be joined to only *one* transaction. A thread is automatically joined to a transaction it creates. In addition, you can call a transaction's `join` method to join a thread with an existing transaction. When a thread joins a transaction (by creating it or by an explicit call to the transaction's `join` method), the thread automatically leaves any transaction it was already joined to. You can also call a transaction's `leave` method to make a thread leave a transaction; you should call `leave` at some point before terminating. To obtain the transaction for the current thread, you call the `Transaction.current` static method.

Your application can create multiple transactions, each corresponding to a particular subtask that your application performs. The transaction's sessions serve to isolate the operations performed in the different subtasks of an application. If more than one transaction uses a given Objectivity / DB object, each transaction's session has its own local representation of that object.

EXAMPLE This code fragment illustrates what happens when a thread creates and uses more than one transaction.

```
// Thread joined with tx1
Transaction tx1 = new Transaction();
tx1.begin();
... // Perform processing on Objectivity/DB objects
tx1.commit();
// Thread joined with tx2
Transaction tx2 = new Transaction();
tx2.begin();
... // Perform processing on other Objectivity/DB objects
tx2.commit();
```

If both transactions are created at the beginning of the sequence of operations, you must explicitly call the `join` method:

```
// Thread joined with tx1
Transaction tx1 = new Transaction();
// Thread joined with tx2 and leaves tx1
Transaction tx2 = new Transaction();
tx1.join(); // Rejoin tx1 and leave tx2
tx1.begin();
... Perform processing on Objectivity/DB objects
tx1.commit();
tx2.join(); // Rejoin tx2 and leave tx1
tx2.begin();
... // Perform processing on other Objectivity/DB objects
tx2.commit();
```

Example

Consider a server application where each thread spawned to handle incoming requests typically does not complete before a new thread is spawned. Instead of having each thread create a new transaction, the example shares a pool of transactions among the threads. Access to the pool of transactions is synchronized.

The `MultipleThreadsTP` programming example simulates a server application handling requests to look up root objects in a database. The server creates a new thread to handle each request and each thread gets a transaction from a shared pool of transactions. The transaction pool is implemented in the `TransactionPool` class.

This example can be executed after you compile the files and create a federated database named "Objects" in the ODMGApplication subdirectory of the samples directory.

4

Locking and Concurrency

The objects in an Objectivity / DB federated database are shared by sessions that may at times try to perform incompatible operations on those objects. For example, two sessions may read, and then subsequently update, an object. If both sessions perform these actions simultaneously, one of the updates would be overwritten by the other update. To prevent incompatible operations, Objectivity / DB uses a mechanism, called locking, that allows an application to inform Objectivity / DB how it plans to use an object. When an application requests a read lock, the application indicates to Objectivity / DB that it needs read-only access to an object. When an application requests a write lock, the application indicates that it intends to modify the object.

Containers are the fundamental unit of locking within Objectivity / DB; when any basic object in a container is locked, the entire container is locked, effectively locking all basic objects in the container. Whenever a lock is requested for a container, Objectivity / DB applies the session's *concurrent access policy* to determine whether the requested lock is compatible with any existing locks.

In This Chapter

Getting a Lock

Objectivity / DB automatically requests the locks needed by your application at the point at which they are needed. Your application can also explicitly request locks. Locks obtained either way are maintained by Objectivity / DB until the application commits or aborts the transaction, at which time all locks obtained during the transaction are released.

When Objectivity / DB *cannot* obtain a lock that it requires, the method that generated the lock request will throw a `LockNotGrantedException`. To minimize such exceptions, Objectivity / DB supports locking strategies and mechanisms that reduce the probability of lock conflicts.

Implicit Locking

Objectivity / DB will *implicitly* obtain the appropriate locks for your application at the point at which they are needed. An operation that reads an object will obtain a read lock; an operation that modifies an object will obtain a write lock.

Properties of Locks Obtained Implicitly

When a federated database or database is locked implicitly, the locked object can be shared; the lock is not used to restrict access to the object, only to ensure that Objectivity / DB records modifications to the object in a *journal file*. Objectivity / DB uses the journal file to restore a federated database to its previous state if the transaction is aborted or terminated abnormally.

Implicit locks on containers are governed by a session's concurrent access policy.

Obtaining a Lock Implicitly

Retrieving an object doesn't lock it or its containing object. After an object is retrieved, however, Objectivity/DB will implicitly obtain locks for the following operations:

Operation	Locks Obtained
Creating a database	When a database is created, it and its federated database are write locked.
Making an object persistent	When a basic object or container is made persistent, the object and the object's containing object are locked for write.
Copying a persistent object	When a basic object is copied, the object's container is locked for read and the container in which the copy is being stored is locked for write.
Moving a persistent object	When a basic object is moved, the object's container and the container to which the object is being moved are locked for write.
Marking an object as being modified	When an object is marked as modified via `markModified()`, Objectivity/DB tries to obtain a write lock.
Fetching an object's persistent data	When you fetch an object's persistent data, Objectivity/DB obtains an appropriate lock based on the lock mode in which the data is being fetched. If the object is a basic object, the container in which the object resides is locked.
Locking a container	When a container is locked, all basic objects within the container are also locked.
Scanning a storage object	When a federated database or database is scanned, all the containers within the federated database or database are locked for read. When a container is scanned, the container is locked for read.
Lock propagation	Locks will be propagated to objects that are connected through relationships with lock propagation behavior enabled.
Deleting an object	When an object is deleted, the object's containing object is locked for write.

NOTE When Objectivity / DB *cannot* obtain a lock that it requires, the method that generated the lock request will throw a `LockNotGrantedException`. If you don't want the session to abort the request immediately, you can set the mode of the session to wait for locks. To minimize the possibility of a conflict while a transaction is in progress, you can also explicitly reserve all the locks you might need at the beginning of the transaction.

Concurrency and Iterators

To improve concurrency, Objectivity / DB performs a special locking procedure for sessions where iterators are used to scan for one of the following:

- Database in a federated database
- Container in a database

This special locking procedure locks and unlocks objects during the iteration, which increases concurrency significantly. If your application requires repeatable read operations during one of the iterations described above, you should explicitly lock the iteration object (database or container) using the `lock` method.

Iterating on a Federated Database

When iterating to get the next database in a federated database, Objectivity / DB will automatically release the lock on the current database provided the following two conditions are met:

- The database is implicitly read-locked during the iteration. For example, at the start of the iteration the database is not locked, and the `scan` or `contains` method is called to lock and scan the database.
- There are no locked containers or basic objects within the database.

Iterating on a Database

When iterating to get the next container in a database, Objectivity / DB automatically releases the lock on the current container provided the following two conditions are met:

- The container is implicitly read-locked during the iteration. For example, at the start of the iteration the container is not locked, and `scan` or `contains` is called to lock and scan the container.
- There are no locked basic objects within the container.

Explicit Locking

Implicit locking obtains access rights to resources as they are needed by an application. In general, the automatic locking by Objectivity / DB provides a level of federated database concurrency that is sufficient for most applications.

Some applications, however, may need to reserve access to all required resources in advance. Reasons for doing so might be to secure required access rights to the necessary objects before beginning an operation, or to prevent other sessions from modifying objects critical to the operation.

An application needing to reserve access to all required objects in advance can *explicitly* lock objects. Suppose an application needs to calculate a value based upon the state of many objects at a specific point in time. Although the application cannot check all of the necessary objects simultaneously, it can achieve the same effect by freezing the state of the objects and then checking them in sequence. Explicit locking effectively freezes the objects, because no other session can modify them as long as they are locked.

Properties of Locks Obtained Explicitly

When a federated database or database is locked explicitly, the locks are exclusive. A write lock prevents all other concurrent access, and a read lock prevents another session from modifying the locked object. For example, if a database is explicitly locked for write, any operation that implicitly requests a read or write lock (such as scanning the database or adding a container to the database) will be unable to get a lock.

In addition, explicit locks on a federated database or database affect all contained objects. When a federated database is locked explicitly, all the databases, containers, and basic objects it contains are also locked. Similarly, when a database is locked explicitly, all the containers and the basic objects it contains are also locked.

When performing explicit locking of a federated database or database, you should consider how this behavior affects concurrency. While obtaining locks at a higher level in the storage hierarchy may simplify application programming, it will also prevent multiple users from accessing objects simultaneously.

Explicit locks on containers are governed by a session's concurrent access policy.

An explicit lock on a basic object or container whose relationships have lock propagation enabled propagates to any related basic objects and containers. "Propagating Operations" on page 151 describes how lock propagation works and how to specify such relationships.

Obtaining a Lock Explicitly

You explicitly lock an Objectivity / DB object with the object's `lock` method.

You can lock a basic object or container without propagating to related objects, with the `lockNoProp` method.

Objectivity/DB Lock Server

All lock requests, both implicit and explicit, are forwarded to the Objectivity / DB *lock server*, which grants, tracks, and releases locks for a particular federated database or autonomous partition. The standard lock server is a separate process running on the host specified by the federated database. If all lock requests originate from a single, multithreaded application, an application can optionally run its own lock server internally; see Chapter 18, "In-Process Lock Server".

NOTE An application by-passes the lock server when accessing objects in a read-only database; the application automatically grants its own read locks and refuses any requested update locks. See "Making a Database Read-Only" on page 94.

Limits on Locks

The open mode of the connection to the federated database and the open mode of a session limit the locks you can obtain for the objects owned by the session.

The session's open mode directly limits the locks you can obtain. A session open mode of read / write allows objects to be locked for read or write; a session open mode of read-only allows objects to be locked for read.

The session's open mode is set with the `setOpenMode` method of the session. If the `setOpenMode` method is not called on a session, its default open mode is the connection's open mode. If you try to set the session open mode to read / write when the connection open mode is read-only, an `ObjyRuntimeException` will be thrown when the session begins a transaction.

If the session is in a transaction, you can change its open mode from read-only to read / write and Objectivity for Java upgrades the lock to a write lock. If the lock cannot be upgraded, an exception is thrown. If you attempt to change the open mode from read / write to read-only while the session is in a transaction, `setOpenMode` throws an exception. If the session is outside a transaction, you can set the open mode to either read / write or read-only.

The connection's open mode limits the open mode of any sessions created during the connection. A connection open mode of read / write allows a session's open mode to be read / write or read-only; a connection's open mode of read-only allows a session's open mode to be read-only. The connection open mode is set

when the connection is opened with the static method `Connection.open` and can also be changed with the `setOpenMode` method of the connection.

If the connection is open, you can change its open mode from read-only to read / write. If you attempt to change an open connection from read / write to read-only, `setOpenMode` throws an `ObjyRuntimeException`. If the connection is closed, you can set the open mode to either read / write or read-only.

Managing Locks

Objectivity for Java provides a number of methods that allow you to upgrade, downgrade, and release locks.

Upgrading Locks

You can upgrade (from read to write) the lock of an object with an object's `lock` method.

Downgrading Locks

Committing or aborting a transaction releases *all* the locks held by the session, thus permitting other sessions to access the objects. During a transaction, you can checkpoint the changes made thus far during the current transaction. Checkpointing makes all changes permanent (i.e., writes the objects to the database) and allows other applications to access those changes. However, the transaction remains active and all previously held locks are still held by the session. You can downgrade all the write locks to read locks by calling the `checkpoint` method with the argument `oo.DOWNGRADE_ALL`.

Releasing Read Locks

If you need read access to only some objects, you can release the read lock of an individual container and all its contained objects by calling the container's `releaseReadLock` method. This allows other sessions to gain read or write access to those containers.

When Objectivity / DB acquires and releases locks automatically, it uses two-phase locking. Two-phase locking means that once a session releases a lock it will not subsequently acquire any more locks. Two-phase locking guarantees that the actions of concurrent transactions are serializable. If you explicitly acquire and release locks, you should do so in a manner that ensures serializability.

Concurrent Access Policies

When a session requests a lock for a container that has already been locked by another session, the sessions' concurrent access policies are applied to determine which lock requests are compatible. Objectivity/DB supports two concurrent access policies: *exclusive* and *MROW*.

Exclusive Policy

The *exclusive* concurrent access policy enforces the following rules:

- If a session has a read lock on a container, any other session may also obtain a read lock on the same container.
- As long as a container is locked in read mode by at least one session, no other session can obtain a write lock for it.
- If a session has a write lock on a container, no other session may obtain a lock (read or write) on the same container.

This exclusive policy prevents any application user from being misled by viewing data that may be in the process of being altered by another session.

For many applications however, preventing access to an object in the process of being updated is too restrictive. Such applications would rather access potentially out-of-date data, than not access the data at all. Consider a web application that serves web pages from a database. If the application uses a session with an exclusive access policy to read the database, a web page that was being updated by the webmaster would be unavailable.

Multiple Readers, One Writer (MROW) Policy

The *multiple readers, one writer* (MROW) concurrent access policy relaxes the restriction that a container may not be simultaneously updated and read. MROW is used to allow sessions to read the last committed or checkpointed version of a container being updated by another session. In addition, a session can update a container if all sessions that are reading the object obtained their read locks with MROW enabled. For example, if the web server application described in the previous section uses MROW-enabled sessions, a version of the web page would be available at all times.

The concurrent MROW access rules are:

- A session with MROW enabled can get a read lock for a container that is locked for write by another session.
- A session can get a write lock for a container that is locked for read by another session with MROW enabled.
- Write locks are still mutually exclusive. If a session has a write lock on a container, no other session may obtain a write lock on the same container.

NOTE MROW is for use by sessions reading objects that are being updated by non-MROW sessions. In some circumstances an MROW session may need to update these objects. To ensure serializability, the session must upgrade any locks on these objects, and any objects on which the update is dependent, to write. For example, suppose an MROW session is reading objects A and B from different containers. If the session wants to update the value of a field in A with the value of a field from B, then to ensure that A gets a valid value, both A and B must be locked for write.

Setting MROW for a Session

You enable or disable MROW using the `setMrowMode` method of a session. You can also check the MROW setting for a session by calling its `getMrowMode` method.

MROW can only be enabled or disabled on a *closed* session, that is before `begin` is called and after `commit` or `abort` is called on the session. During application execution, you can check whether a session is open using the `isOpen` method.

Managing Containers Under MROW

When you read a container in a session that has MROW enabled, you can use the `isUpdated` method to check whether the container has already been updated and committed by another session. This method returns true only if the container has been updated by another session since being locked for read by the current transaction. If the container has been modified, you can refresh the container with the `refresh` method.

EXAMPLE This example implements the web server application discussed earlier.

```
public class WebServer {
   private static Session session;
   private static ooDBObj pageDB;
   private static ooContObj pageContainer;

   private static void init() {
   ...
   session = new Session();
   // Activate MROW for this session
   session.setMrowMode(oo.MROW);
   ... // Initialize pageDB
```

```
   session.begin();
   pageContainer = pageDB.lookupContainer("Page Container");
   session.commit();
}

private void handleGet(String pageName, OutputStream out) {
   Page page;// Persistent page
   ...// Get session
   session.begin();
   // Check whether container has been updated.
   if (pageContainer.isUpdated())
      // Refresh container if necessary
      pageContainer.refresh(oo.READ);
   // Look up the page object by its scope name "pageName"
   page = (Page)pageContainer.lookupObj(pageName);
   if (page == null)
   ...// Write error message to output
   else
   ...// Write page to output
   session.commit();
   }
}
```

Concurrent Access Rules

The concurrency characteristics for exclusive and MROW concurrent access policies are:

- A container cannot be updated by two sessions simultaneously, regardless of their MROW status.
- A session can update a container that is being read by an MROW session.
- An MROW session can read a container that is being updated by another session.

To achieve the maximum concurrency in your application, you need to allocate your objects to containers based on the expected usage profiles of the objects. “Planning for Concurrent Access” on page 100 discusses a variety of strategies for allocating objects to containers.

Lock Conflicts

Within a session, an application can request either read or write locks on various objects, either implicitly or explicitly. When a competing session requests a lock on one of the same objects, it presents the potential for a *lock conflict*—that is, an incompatible operation on the same object.

There are two approaches to dealing with lock conflicts: employ mechanisms to reduce conflicts or, when they are unavoidable, handle them in an appropriate manner.

Reducing Lock Conflicts

You should analyze your application requirements and, when possible, employ locking strategies that reduce the probability of lock conflicts. Since locking a container for write prevents all other sessions from accessing both that container and all basic objects in that container, you can use the MROW concurrent access policy so that sessions can still access objects that are write locked. Alternatively, you can minimize the time that objects are locked for write.

One approach to minimizing the length of time that write locks are held is to minimize the length of the transaction. You should remember, however, that shorter transactions imply more frequent lock requests. If the lock server is remote, and network access is slow, this can have a negative impact on performance.

Another approach to limiting the length of time that other sessions are locked out is to lock for write at the point in the execution of your application when you need to update the object. If your application first needs to read an object and then, later on in the execution of the application, needs to update the object, you can upgrade the lock from read to write access. Of course, if another object also has the object locked for read, you may not be able to upgrade the lock.

Finally, when possible, you should downgrade or release your locks as soon as possible.

Handling Lock Conflicts

Some lock conflicts are bound to arise, even if your application uses the strategies for reducing lock conflicts described in the previous section. You can configure Objectivity for Java to respond to this situation in one of two ways:

- Immediately give up on the operation and throw an exception. This is the default behavior.
- Wait until the desired object is available.

Lock Waiting

If you want to wait for objects to become available, you need to activate lock waiting for a session. While a session is waiting for a lock, the thread using the session will be blocked on I/O. Once the object is unlocked, waiting sessions are granted locks in the order that they were queued.

You activate lock waiting for a session by using the `setWaitOption` method. This method allows you to specify a specific timeout period, that the session should wait indefinitely, or that it should not wait at all. This method can only be called on a *closed* session, that is before `begin` is called and after `commit` or `abort` is called on the session. During application execution, you can check whether a session is open using the `isOpen` method.

NOTE An MROW session determines immediately whether it can get a requested lock, and ignores any timeout period specified by the `setWaitOption` method.

Deadlock Detection

A deadlock is a circular condition where two or more sessions are queued, and each is waiting for a lock that will never become available. For example, a deadlock is created when the following conditions are simultaneously true:

- `Session1` is waiting for `Session2`.
- `Session2` is waiting for `Session3`.
- `Session3` is waiting for `Session1`.

If a deadlock condition is detected, Objectivity for Java throws an exception. Whenever infinite lock waiting is requested, Objectivity/DB checks to see whether queuing the request would result in a deadlock situation. If so, an error is returned to the requesting application. When finite lock waiting is requested, no deadlock checking is done. In this case, Objectivity/DB assumes that any deadlock condition that occurs will be broken when lock waiting times out.

5

Storage Objects

Objectivity / DB federated databases, databases, and containers are storage objects. Storage objects serve to group other objects to achieve performance, space utilization, and concurrency requirements. This chapter describes how to use the Objectivity for Java programming interface to perform tasks involving storage objects. As an alternative to using the programming interface, you can accomplish some of these tasks through administration tools. In those cases, the relevant tools are identified, and a reference is given for further information.

The process of assigning a container to a database and a basic object to a container is called *clustering*; that topic is addressed in Chapter 12, “Clustering Objects”. This chapter discusses the reasons for using the different types of storage objects and how to create, delete, and retrieve each type of storage object.

You can increase the robustness of your application by making use of Objectivity for Java features that support partitioning a federated database into units that can operate independently and replicating an individual database within those units. Those features are accessible transparently through the Objectivity for Java programming interface; however, they require two additional Objectivity products, Objectivity / FTO and Objectivity / DRO, to gain access to the functionality within Objectivity / DB. Further information on how to use those features can be found in Chapter 16, “Autonomous Partitions” and Chapter 17, “Database Images”.

In This Chapter

Function of Storage Objects

A federated database consists of system-created databases and databases created by your application. A federated database:

- Maintains the object model (or schema) that describes all the objects stored in the databases.
- Is the unit of administrative control for Objectivity/DB; it maintains configuration information (where Objectivity/DB files physically reside). All recovery and backup operations are performed at this level.

A database consists of system-created containers and containers created by your application. A database is physically maintained as a file, and is used to:

- Distribute processing burdens across multiple host machines.
- Locate objects physically near their users.
- Increase the capacity of the federated database.

Containers serve two main purposes:

- To group basic objects. Basic objects within a container are physically clustered together in memory pages and on disk, so access to collocated basic objects in a single container is very efficient.
- As the unit of locking. When a basic object is locked, its container and all other objects in that container are also locked. This organization reduces the burden on the lock server in systems with a large number of objects.

Working With a Storage Object

For each storage object that an application wants to access, an application must:

- Create or retrieve a local representation of the storage object. All operations on a storage object are directed to the local representation.

  Local representations of storage objects are created and typically retrieved from their containing object. Thus a database is created and retrieved from its containing federated database.

  The local representation of a storage object *belongs to* a session. A newly created storage object belongs to the session that was in a transaction in which it was created or made persistent. A previously existing storage object belongs to the session that was in a transaction in which it was retrieved. If your application uses multiple session objects, a storage object that belongs to one session *may not* interact with any objects that belong to different sessions. See "Object Isolation" on page 51.

- Obtain a lock on the storage object for the desired access.

  The session that owns the local representation of an object holds the lock on the object. For all operations on storage objects, Objectivity / DB will obtain the locks it needs automatically at the point at which the lock is required. For example, if you scan a database, Objectivity / DB will obtain a read lock on the database and all the containers in the database. If you create and make a container persistent, the container and the database in which the container is clustered will be locked for write.

  If Objectivity / DB *cannot*, due to a conflict with another lock, obtain a lock that it requires, the method that generated the lock request will throw a `LockNotGrantedException`. You can explicitly reserve locks in advance of using the object if you wish to avoid such exceptions. See Chapter 4, "Locking and Concurrency," for further information.

Any operation on a storage object (including creating or retrieving a storage object or obtaining a lock) must be performed within a transaction. Any changes to a storage object affect only the local representation of the object until you commit or checkpoint the transaction, at which point the effects of the operation are made permanent in the federated database. If the transaction is aborted, the effects of the operation are rolled back. The only exception to this rule is that deleting a database cannot be undone.

Once you have satisfied these conditions, you can:

- Read a property of the storage object, such as the system name.
- Create or cluster an object within the storage object.
- Retrieve the objects contained within the storage object.
- Name or lookup an object within a context defined by the storage object.

Committing or aborting a transaction releases all locks obtained during the transaction. The local representation of a storage object owned by a session is also invalidated: the representation may no longer be consistent with the federated database and any system-defined operation on the object will throw a `TransactionNotInProgressException`. The local representation is still owned by the session, and can be reused in subsequent transactions. Note, however, that if another client has deleted the object while your application was between transactions, an exception would be thrown the next time you try to perform an operation on the object.

The local representation of a storage object becomes dead when a session is terminated, is aborted, when the object is deleted, when a transaction in which a database is created or a container is made persistent, or a copy of an object is obtained. A *dead object* is an object that is no longer valid for Objectivity for Java operations. Any attempt to perform an operation on a dead object throws an `ObjectIsDeadException`.

Federated Databases

A *federated database* is the highest level in the Objectivity / DB storage hierarchy. Each federated database may contain:

- A default database, if any federation-wide named roots are defined.
- One or more application-defined databases.

An Objectivity / DB federated database is physically maintained in a *system database file*, which stores a catalog of all the application-defined databases, a default container, and the schema for the federated database. Each federated database has a *system name*, which is the name of its associated boot file. The *boot file* contains the following federated database configuration information:

- The host name of its *lock server*.
- An integer valued *identifier*, which is used to identify the federated database to the lock server.
- The size of its *pages*, which are the unit of storage, buffering, and data transfer in Objectivity / DB. The page size can be optimized for your application's requirements.

Creating and Deleting a Federated Database

A federated database can be created and deleted only with administration tools.

Tool: `oonewfd` and `oodeletefd` (see the Objectivity / DB administration book)

Retrieving a Federated Database

In Objectivity for Java a federated database is represented by an instance of the `ooFDObj` class. A local representation of a federated database is created automatically by a session. You obtain the local representation of the federated database from a session with the `getFD` method. You can also obtain the federated database of a database by calling its `getFD` method.

These methods do not create a local representation of the federated database, but obtain the local representation created by the session. Methods that obtain the storage object of an object (for example the container of a basic object), exist for all Objectivity / DB objects. All such methods can be called outside a transaction.

Databases

A *database* is the second highest level in the Objectivity / DB storage hierarchy. Each database is created with a default container, where basic objects are stored when they are clustered near a database or named in the scope of a database or federated database. The first time you add a root name to a federated database, Objectivity for Java creates a default database and a roots container in the default database. You can add one or more application-defined containers to a database.

A database is physically maintained in a *database file*. This file contains a catalog of all the application-defined containers, and all the containers and basic objects stored in the database. Each database is attached to exactly one federated database and is listed in that federated database's catalog by its system name.

Assigning Objects to Databases

Objectivity for Java maintains federation-wide named roots in a default database. Nothing prevents you from storing objects—even all of your persistent objects—in the default database. However, you may want to create new databases within the federation in order to:

- Distribute processing burdens across multiple host machines.

  Each database can be located on a separate storage device, typically with a separate processor to manage disk and network activity. By distributing your objects among a larger set of databases, you reduce the number of requests that must be handled by each network path, processor, and storage device.

  Distributing the processing burden in this way also enables you to support parallel-processing applications, because each application process can address a separate database without impeding other process / database pairs.

- Locate objects physically near their users.

  For wide-area intranet or internet applications, you can place geographically relevant subsets of the data on local servers, rather than forcing all users to access a central server.

- Increase the capacity of the federated database.

  Each database has a limited, though extensive, capacity. By increasing the number of databases, you increase the capacity of the federation. See the Objectivity / DB administration book for information about computing federated database capacity.

- Subdividing large datasets

  Databases and containers can be used to subdivide extremely large datasets to reduce search time. In this scheme, a fairly homogeneous set of objects is divided among databases within a federation, and one or more maps are used to associate the databases with various search keys. Within each database, the objects are subdivided among a set of containers, and one or more maps are used to associate the containers with search keys. Assigning an object to a container consists of identifying the subdivision to which the object belongs and clustering the object in the corresponding container.

  For example, consider storing satellite observations of climatic conditions. Each database within the federation might represent a geographic region, and each container within a database might represent a chronological period, such as a month, during which observations for that region were recorded. Each new observation would be clustered in the container corresponding to its month within the database corresponding to its geographical region.

  This hierarchical organization of objects simplifies retrieval. Instead of scanning the entire federated database to find a given observation, the application could first look up the database for the region, then look up the container for the month when the observation occurred, and finally scan that container for the observation of interest.

When a container is made persistent, it is assigned to the database where it will be stored when it is written to the database. You can make a container persistent by explicitly clustering it into the database of your choice or by establishing an association between the container and another persistent object. “Clustering Containers” on page 238 describes the various ways in which a container can be clustered.

Creating a Database

To create a new database, call one of the `newDB` methods on an instance of `ooFDObj`. These methods create a new database file, a database in the federated database, and an instance of `ooDBObj` in your application; it locks the new database in write mode.

Both `newDB` methods require you to specify the system name of the database. In addition, one method allows you to specify the:

- Initial number of pages to allocate for the default container.

  Select the container size based on the number of objects you plan to store in the container and the size of the objects. For a single-object container, for example, you could specify an initial size of one page. For a container that will hold a large set of objects that are rarely updated, you could set the initial size large enough to accommodate the entire set, reducing both time-consuming growth operations and wasted excess space in the final growth operation.
- Percent of its current size by which the default container should grow when needed to accommodate more basic objects.
- Name of the host where the database file is to be located.
- Path of the directory where the database file is to be located.
- Weight of the first database image if the Objectivity / DB Data Replication Option (DRO) is being used. If Objectivity / DRO is not being used, you must specify 1.
- The database identifier. By default, the database is assigned an identifier that is unique within the federated database. You can optionally specify the database's identifier when you create it, for reasons such as the following:
  - Application development is split across several teams, and each team by convention must assign database identifiers from within a certain range.
  - You are reconstructing an existing federated database, and you need to ensure that the database with a given system name has the same identifier in both the original federated database and the new one.

When you commit or checkpoint the transaction in which you create a database, the database is created in the federated database. If you instead abort the transaction, the database becomes a dead object and no physical database is created.

Tool: `oonewdb` (see the Objectivity / DB administration book).

Retrieving a Database

You can retrieve an existing database in any one of the following ways:

- Look it up by name by calling the `lookupDB` method of a federated database.
- Obtain an iterator that finds all databases in the federated database with the `containedDBs` method of the federated database.
- Obtain the default database of a federated database with the `getDefaultDB` method of the federated database.
- Obtain the database of a container with that container's `getDB` method.

EXAMPLE This code fragment creates a session and retrieves its associated federated database. It then checks whether a database named `VehiclesDB` exists and, if so, retrieves it from the federated database. You should check whether the database exists before doing the lookup, because the lookup operation will throw an exception if a database with the name `VehiclesDB` does not exist in the federated database. If the database does not exist, the example creates a new database in the federated database.

```
ooFDObj vrcFD;
ooDBObj vehiclesDB;
Session session = new Session();
session.begin();
vrcFD = session.getFD();
if (vrcFD.hasDB("VehiclesDB"))
   vehiclesDB = vrcFD.lookupDB("VehiclesDB");
else {
   vehiclesDB = vrcFD.newDB("VehiclesDB");
   System.out.println("Created database \"VehiclesDB\".");
}
session.commit();
```

Making a Database Read-Only

If you know that all of the persistent objects in a database are to be read but not updated, you can designate the database as a *read-only database*. A read-only database can be opened only for read; any attempt to open the database for update will fail as if there were a lock conflict. Making a database read-only can improve the performance of an application that performs many read operations on persistent objects, because the application can grant read locks and refuse update locks without consulting the lock server. To make a database read-only, you call the `setReadOnly` method of the database, passing true as the parameter.

When a database is read-only, an application can either read its contents or change it back to read-write. If you need to modify an object in a read-only

database, or if you want to delete the database, you must change the database back to read-write. To do this, you call its `setReadOnly` method, passing false as the parameter.

Any number of databases can be read-only in a federated database. When multiple read-only databases exist in a federated database, they are locked or unlocked as a group. Consequently, a read-only database can be changed back to read-write *only* if no other application or tool is currently reading either that database or any other read-only database in the federation.

You can also make a database read-only or read-write from the command line with the `oochangedb` administration tool (see the Objectivity / DB administration book).

(*DRO*) If a database has multiple images, making one image read-only makes *all* images read-only. While a database is read-only, you cannot add, delete, or change the properties of individual images.

Deleting a Database

You delete a database with its `delete` method. When you delete a database, all of its containers and basic objects are deleted. Until the current transaction is committed, the local representation of the database continues to exist in your application's memory, but it, and all of its contained objects, are marked dead. When you commit or checkpoint the transaction in which you delete a database, the physical database is deleted from the federated database.

WARNING This operation cannot be undone by aborting the transaction.

Tool: `oodeletedb` (see the Objectivity / DB administration book)

Containers

A *container* is the third level in the Objectivity / DB storage hierarchy. Containers serve to group basic objects. Basic objects within a container are physically clustered together in memory pages and on disk, so access to collocated basic objects in a single container is very efficient.

Containers are the fundamental units of locking; when any basic object in a container is locked, the entire container is locked, effectively locking all other basic objects in the container. The container-level granularity of locking requires some planning in your applications, but gives benefits in overall performance, because the lock server needs to manage relatively few container-level locks rather than potentially millions or billions of object-level locks.

Container Types

Objectivity / DB provides two types of containers: *garbage-collectible containers* and *non-garbage-collectible containers*. The two container types differ in how objects are deleted from them. You must consider the container types when deciding how to assign basic objects to containers. See “Selecting the Correct Container Type” on page 99.

Garbage-Collectible Containers

Garbage-collectible containers are designed to store directed graphs of objects that represent composite objects. The roots container of a database is garbage-collectible and every container of the `ooGCContObj` class or an application-defined subclass of `ooGCContObj` is garbage-collectible.

As the name implies, garbage-collectible containers adhere to a garbage-collection paradigm. Just as Java memory can contain “garbage” objects that are not referenced, a garbage-collectible container can include invalid or “garbage” objects that were left over after some object graphs were modified. Unlike Java memory, garbage collection does not occur automatically in a federated database; you must use the `oogc` administration tool when you want to delete invalid objects from the garbage-collectible containers of a federated database.

Validity of Objects

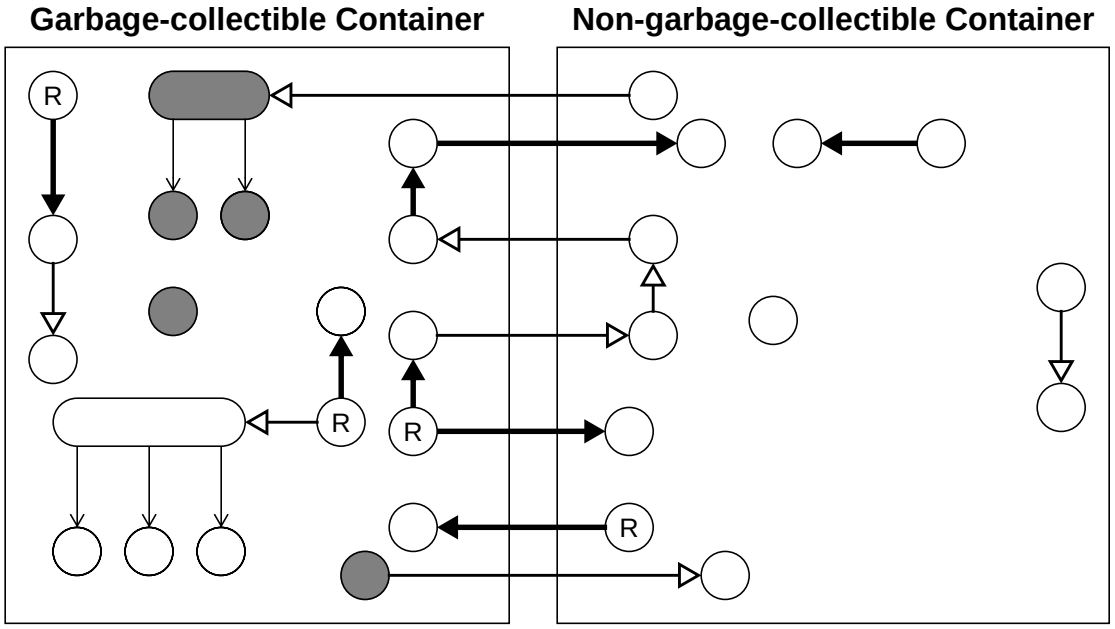
Logically, objects in a garbage-collectible container are meaningful only when they are part of a composite object, that is, when they can be reached by following links from a named root. A composite object is an object graph in which the individual objects are linked together by relationships, by references in persistent fields, and by membership in persistent collections. The root object of each composite object must be a named root. For more information about named roots, see “Named Roots” on page 212.

An object in a garbage-collectible container is assumed to be valid if:

- It is a named root.
- It can be reached from a named root. An object `Obj1` can be “reached” from another object `Obj2` if any of the following conditions is true:
  - `Obj2` references `Obj1` in a persistent field. (The referencing field can be of a scalar or array type.)
  - `Obj2` is related to `Obj1`.
  - `Obj2` is a persistent collection that contains `Obj1`.
  - `Obj1` can be reached from some object that can be reached from `Obj2`.

An object that does not satisfy one of these conditions is considered abandoned and is a candidate for garbage collection.

The following figure illustrates which objects will be deleted by the garbage collector.



Key to Symbols

- R = Named root; will not be garbage collected
- = Valid object; will not be garbage collected
- = Invalid object; candidate for garbage collection
- = Valid persistent collection; will not be garbage collected
- = Invalid persistent collection; candidate for garbage collection
- = Relationship
- = Reference in persistent field
- = Membership in persistent collection

If all the objects in a graph are stored in garbage-collectible containers, applications do not need to delete objects that cease to be part of the graph. An application simply removes links to the object as appropriate (by changing referencing fields, relationships, or membership in collections). Note that the

entire object graph doesn't need be stored in a single container as long as every object in the graph is stored in a garbage-collectible container.

As the preceding diagram illustrates, it is perfectly acceptable to store some objects in a graph in non-garbage-collectible containers. When this is done, however, the application developer assumes the responsibility for deleting any objects in non-garbage-collectible containers that are no longer needed because they cease to be part of the graph.

Scanning

Scanning the objects in a garbage-collectible container is unreliable because you may retrieve abandoned objects as well as valid ones; applications have no way to test whether an object is reachable from a named root or not. You should not call the `scan` method of a garbage-collectible container unless you are sure that the container has no garbage objects. For example, you could safely call the `scan` method in a database maintenance application that is always run immediately after `oogc` has been run.

Non-Garbage-Collectible Containers

Non-garbage-collectible containers are primarily designed for use by an application that must interoperate with an application written in a non-garbage-collected language, such as C++. They could also be used to store objects that are not necessarily part of a composite object. For example, you might create a non-garbage-collectible container to store all objects of a particular class. The default container of a database is non-garbage-collectible, and every container of the `ooContObj` class or an application-defined subclass of `ooContObj` is non-garbage-collectible.

Every object in a non-garbage-collectible container is assumed to be valid. Applications that use non-garbage-collectible containers to store some or all component objects of a composite object's graph are responsible for removing an object that becomes unlinked from a graph. To remove an object, you must make an explicit call to its `delete` method. The `oogc` administrative tool does not remove objects from non-garbage-collectible containers.

Assigning Basic Objects to Containers

When a basic object is made persistent, it is assigned to the container where it will be stored when it is written to the database. You can make an object persistent by explicitly clustering it into the container of your choice. If you make the object persistent in some other way, for example, by making it a named root, the object is clustered implicitly. “Clustering Basic Objects” on page 238 describes the various ways in which an object can be clustered.

Your plan for assigning objects to containers should take into account container type, concurrency, and runtime efficiency of the applications that use the objects, and storage requirements of the federated database. In many applications, concurrency is the most important consideration.

If at some point you determine that you need to reallocate objects to a different container configuration, you can do so by moving the objects. As a database grows larger and the number of interobject links increases, however, moving an object becomes progressively more complex. See “Moving a Persistent Object” on page 176 for more information.

Selecting the Correct Container Type

In deciding how to assign an object to a container, you need to consider how the object will be used; certain uses require specific container types. Follow these guidelines to ensure that you select the correct container type for each basic object.

- If you assign a named root to a garbage-collectible container, it is simplest to assign all objects in the named root’s object graph to garbage-collectible containers. (Otherwise, you will have to keep track of which component objects need to be deleted.)
- Unless an object is a named root or part of the object graph whose root node is a named root, you should not assign it to a garbage-collectible container.
- If you plan to use the object as a scope name, you must assign it to a hashed container (which may be garbage-collectible or not).

Remember that the action that makes an object persistent ultimately assigns it to a container. If you need to perform two or more actions to an object, both of which would make it persistent, the first of these actions makes the object persistent and so determines where the object is stored. Be sure to order the actions so that the object is assigned to the appropriate container.

For example, suppose that you perform the following actions in a transaction that uses the default clustering strategy.

1. Make `Object1` a named root of the database `DB1`.
2. Assign `Object2` to `Container1`, a non-garbage-collectible container.
3. Form a relationship from `Object2` to `Object3`.
4. Form a relationship from `Object1` to `Object3`.

Step 1 assigns `Object1` to the (garbage-collectible) roots container of `DB1`. Step 2 assigns `Object2` to `Container1`. Step 3 assigns `Object3` to `Container1`. If you want to rely on the garbage collector to remove unreachable objects that become unlinked from an object graph, all objects in the graph should be in a garbage-collectible container. Unfortunately, `Object3` has been assigned to a non-garbage-collectible container. Steps 3 and 4 should be reversed so that `Object3` is instead assigned to the (garbage-collectible) roots container of `DB1`.

Planning for Concurrent Access

Before you decide how to assign objects, consider how various users need to access the objects. You can increase concurrency by creating different containers for objects with different usage profiles. The following guidelines will help you improve concurrent access:

- Assign all components of a composite object to the same container if the entire composite object will be accessed as a unit.
- If a composite object is large and complex and can be divided logically into subsystems that may be modified independently, store the objects that make up each subsystem in a separate container.
- If a large number of objects are read frequently but rarely updated, you can safely assign them all to the same container.
- Distribute objects that require frequent update among as many containers as reasonably possible.
- Keep shared resources in separate containers from objects that use those resources.
- Use multiple readers, one writer (MROW) sessions to help manage applications that require containers to be locked for long periods of time.

The following sections describe various approaches to assigning objects to containers.

Shared Resources

Many applications use various collections as shared resources for finding objects. Maps are often shared resources; for example, any database containing named roots has a roots dictionary that maps root names to objects. Objects may also represent shared resources, for example, an assembly line or loading dock in a manufacturing company.

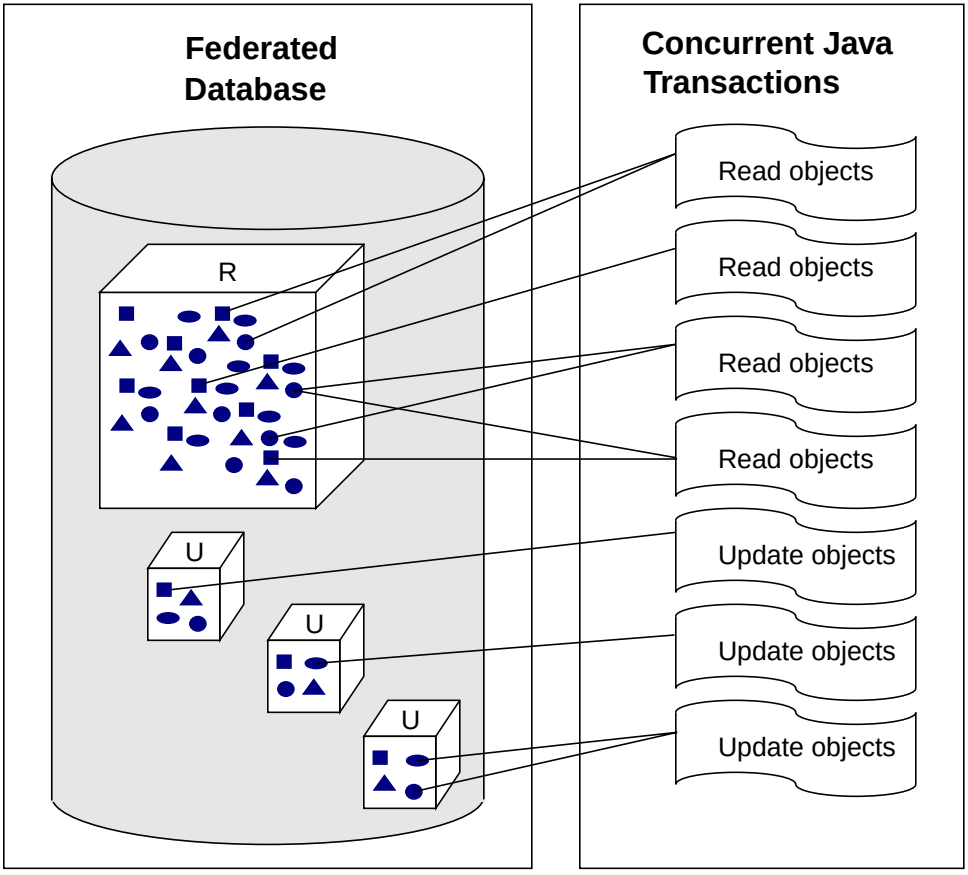
If your application uses such shared resources, you should make sure that each resource object's container is locked for write as seldom as possible. When one application locks the container for write, no other application can modify the resource (for example, to add items to a collection, or schedule use of a machine on the assembly line). Only MROW transactions can check the state of the resource (for example, look up objects in the collection or examine progress of a product through the assembly line).

A good way to ensure availability of a shared resource is to avoid assigning other objects that may change to the same container as the resource itself. For example, if a named root or the objects in its object graph are likely to be updated, you should cluster the object explicitly before making it a named root. If you don't cluster it, the default clustering strategy assigns the new named root (and any unclustered object in its object graph) to the roots container of the database. Because that container also stores the root dictionary, any application that updates the object graph prevents other transactions from adding new named roots to the database.

Read-Intensive and Update-Intensive Containers

Ideally, a container has only the objects that a transaction requires, so locks will be applied sparingly. However, the typical application accesses a given object from several different transactions, each of which requires a different mix of objects. You can go a long way toward accommodating varied transactions by first segregating read-intensive objects from update-intensive objects. Because read locks do not interfere with other read locks, a container full of read-intensive objects will rarely be locked in a way that impedes concurrent access.

The following figure illustrates a database in which objects are separated into read-intensive and update-intensive containers.



Key to Symbols

= Database
= Read-intensive container
= Write-intensive container
= Transaction
= Persistent object
= Persistent object
= Persistent object
= Persistent object

Frequently, you can facilitate this segregation of read-intensive from update-intensive objects during the object modeling phase of a project by splitting classes that contain both read-intensive and update-intensive fields. For example, in a vehicle rental application, a vehicle logically contains both static information, such as the vehicle's class, size, and transmission type, and frequently changing information, such as the collection of rental transactions, and perhaps an

availability flag. The object modeler could separate the update-intensive information about a vehicle into a `VehicleHistory` class, making it possible to cluster instances of `Vehicle` in read-intensive containers while clustering instances of `VehicleHistory` in update-intensive containers.

When an object tends to be updated by a single user at a time, you can cluster it with read-intensive objects if your application uses MROW transactions.

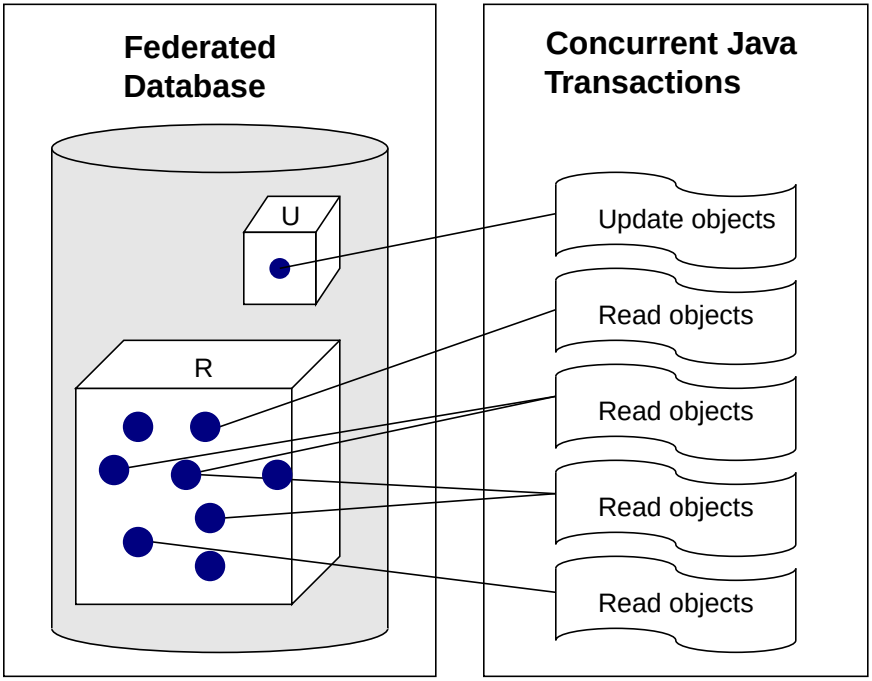
By their nature, update-intensive containers are best kept relatively small in number of objects, though not necessarily in absolute size. The fewer objects an update-intensive container has, the fewer objects each user write-locks in each transaction. In the case of highly volatile objects, many developers isolate each object in its own container. That approach is only slightly inhibited by the ceiling (32,767) on the number of containers in a database, because you can always create additional databases. For example, it would require portions of two databases to isolate each of 50,000 customer orders in its own container. Remember, however, that each container adds to the size of the database; you may need to limit the number of containers, trading off concurrency against physical storage requirements.

For read-intensive containers, deciding when to use a single large container and when to use multiple smaller containers typically hinges on performance and scalability considerations.

Young-Object and Mature-Object Containers

In some applications, objects are highly volatile during their infancy, but eventually mature to a stable state. For example, consider a contract being assembled by a team of consultants. During the bidding and negotiation phase, the contract object is likely to undergo rapid change by multiple users; this situation requires a high degree of update concurrency. Until the contract is signed, the object graph representing a contract might be isolated in a container by itself. After the contract is signed, however, its access becomes read intensive rather than update intensive, so it can be clustered with other signed contracts.

Such applications can store young objects in update-intensive containers. The following figure illustrates a database in which objects under development are separated from stable, mature objects. Concurrency considerations typically dictate how many objects you cluster in each young-object container. Performance considerations typically determine when to use a single large mature-object container and when to use multiple smaller containers.



Key to Symbols

= Database
= Update-intensive container
= Read-intensive container
= Young object under developm
= Stable, mature object
= Transaction

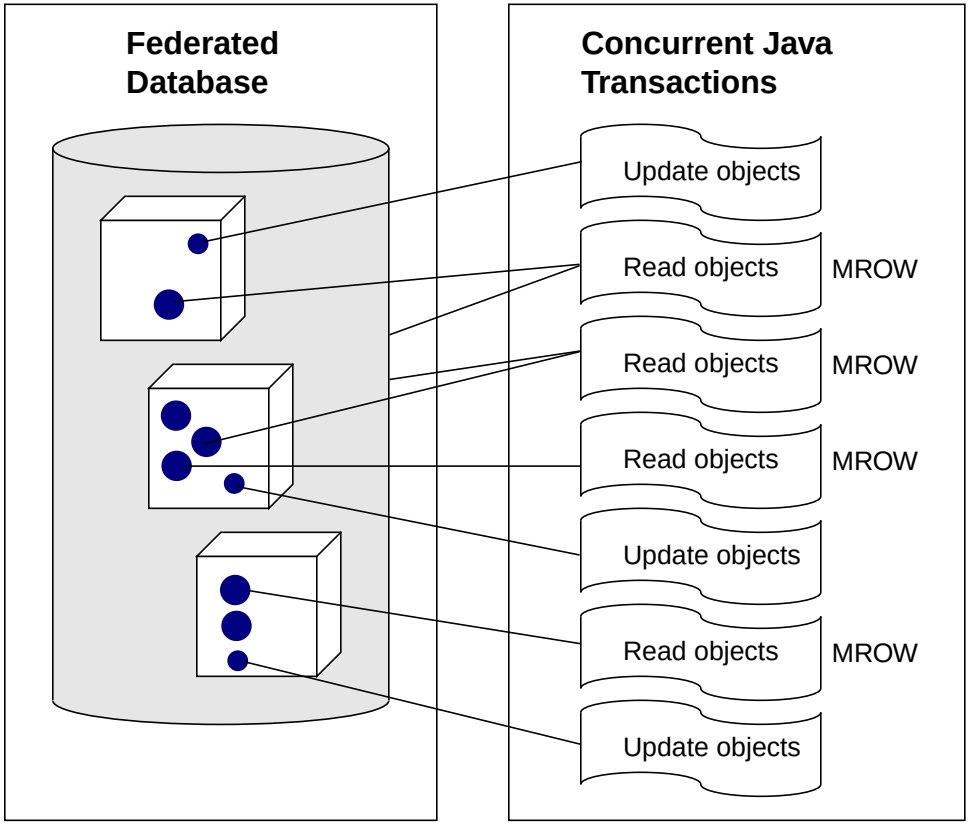
When development is complete, the object can be moved from the young-object container to a read-intensive container for stable, mature objects. When you move the contract and its network of subobjects to the read-intensive container, the objects will change their object identifiers, so you must also redirect any references from the original to the moved objects (bidirectional relationships are updated automatically). See “Moving a Persistent Object” on page 176 for more information.

Round-Robin Assignment

In some applications it is not feasible to use young-object containers and mature-object containers. If the mature object graph is too complex, the deep-move operation needed to move the object and all its subobjects to a new container may require unacceptable runtime delays or storage overhead or both. An alternative is to use a round-robin approach to assign objects to containers. Following this approach with the contract example, each new contract would be assigned to the next container in a pool of containers. The pool would be large enough to guarantee that each container would have a limited number of contracts under development at any given time; the other contracts in the container would be mature enough to be stable. The size of each container in the pool would depend on performance considerations.

NOTE If young objects are assigned to containers using a round-robin approach, an application must be able to modify the developing object while other transactions read mature objects in the same container. To maximize concurrency, the applications that read the mature objects should use MROW sessions, which ensures them read access to the mature contracts while another application has write access to the developing young contract.

The following figure illustrates a database in which objects under development are assigned to containers using a round-robin approach.



Key to Symbols

= Database
= Container
= Young object under development
= Non-MROW transaction
MROW = MROW transaction
= Stable, mature object

Estimating Availability

When trying to decide the minimum number of containers needed to ensure acceptable concurrency, the main variables to consider are:

- The number of simultaneous processes in which the database is updated (`P`).
- The number of containers updated per process (`C`).

You should decide what percentage availability will be acceptable to your users and select the total number of containers (T) to provide the desired availability.

For example, suppose you estimate that during peak times, 20 users will be updating 2 containers each in the database simultaneously. Thus:

```
P = 20
C = 2
```

If each user is constrained to update 2 particular containers that no one else updates, you could ensure 100 percent availability with the total number of containers (T) of 40 (calculated as P*C).

If users update random containers, however, you cannot predict which container a given user will open. The same 40 containers would provide less than 1 percent availability. A given user competes with 19 other users, who have locked 38 containers. That leaves just 2 containers available. The odds of either one of the two desired containers being available are 2 in 40 or 0.05; the odds that a particular desired container is available is 1 in 40 or 0.025. To summarize:

```
competingUsers = P - 1

lockedContainers = competingUsers * C
                 = (P - 1) * C

availableContainers = T - lockedContainers
                    = T - ((P - 1) * C)
```

Because the user comes up against these odds twice (once for each desired container) the odds per container must be squared, resulting in 0.0025 as the odds that both containers are available. To summarize:

```
oddsPerContainer = availableContainers / T
                 = (T - ((P - 1) * C)) / T

oddsForAllContainers = oddsPerContainer ** C
                     = ((T - ((P - 1) * C)) / T) ** C
```

Multiply by 100 to convert the decimal value to a percentage, giving 0.25 percent availability. Thus, the formula for calculating availability is:

```
availability = (((T - ((P - 1) * C)) / T) ** C) * 100
```

Performance Considerations

In general, you should cluster objects so that each transaction has to lock the minimum number of containers. Each container carries a certain amount of overhead in terms of CPU activity, network traffic, and memory usage. Specific considerations are:

- Lock request

  Each time a container is locked, the application makes a call to the Objectivity / DB lock server, incurring both processing and network delays. The more containers you lock in a transaction, the more lock server calls you make. (This issue is not a concern for an application that runs an in-process lock server; see Chapter 18.)

- Page map

  Each container maintains a table that maps logical object identifiers (OIDs) to physical device locations. This page map enables the container to quickly locate the object with a particular OID. The more objects a container has, the larger its page map.

  When a container is locked, its page map is read from the disk, transmitted across the network, and stored in the client cache managed by Objectivity for Java. The page map remains in the cache across transactions, so the CPU and network burdens are only borne by the first transaction that locks the container. However, when an object in the container is updated, the page map has to be refreshed, which is another reason to segregate update-intensive objects from read-intensive objects.

- Catalog of containers

  Each database maintains a catalog of containers. The more containers in the database, the larger its catalog. When you open a database, its catalog is read from disk and transmitted across the network to the client, where it takes up space in the cache. If a new container is created during the session, the catalog has to be refreshed in the cache at the beginning of the next transaction.

  When you access a container by name, the catalog is searched sequentially, so a large catalog is relatively slow to search. You can accelerate name searches in a large catalog by using a map to create your own table of container names.

Storage Requirements

The more containers in your federated database, the more disk space it requires; every container is allocated a certain minimum number of pages. (You can control the number of pages when you create a container and the page size when you create the federated database.) If you have many containers with only a few objects in each, you may be using more storage than necessary. In very large databases, you may decide to reduce the number of containers, thus sacrificing some concurrency and runtime efficiency in favor of reducing storage overhead.

The manner in which you delete basic objects may affect the disk space required for your federated database. The packing density of a container may be low if most of the objects in it are deleted near the end of a transaction.

If your application creates temporary objects in the database that you know will be deleted, you should consider storing those objects in their own container to reduce the fragmentation of secondary storage. For example, a design application may create temporary objects that are used during the development of a design but are deleted when the design is approved. If the temporary objects are kept in the same container as designs, that container may become fragmented. In deciding to devote a container to temporary objects, however, you trade off secondary storage requirements against runtime performance. You may prefer to reduce the performance overhead by clustering the temporary objects with their design.

If you choose to use a separate container for temporary objects, you should consider whether the entire container should be made temporary. In the design example, if one or more designs is likely to be under development at any given time, the database will always contain temporary objects, so their container should be permanent. On the other hand, if long periods of time may pass between the completion of one design and the start of another, you could create a temporary container to hold the temporary objects associated with a particular design. When the design is approved, the entire container could be deleted (which deletes all its objects).

Creating a Container

Containers are created using the normal Java object instantiation methods. To create a garbage-collectible container, instantiate the `ooGCContObj` class; to create a non-garbage-collectible container, instantiate the `ooContObj` class.

If your application needs persistent container-specific data, you can define your own container classes. Every container class should be derived from `ooGCContObj` or `ooContObj`; the persistent fields of a class represent the persistent data for containers of that class. See Chapter 6, “Defining Persistence-Capable Classes”.

Making a Container Persistent

When you call `new` to create a container object, the newly created container is transient. You must make a container persistent before you call any methods defined by the `ooContObj` class; for similar restrictions on inherited methods, see the `ooObj` method descriptions.

You make a container persistent by adding it to a database using the `addContainer` method. Parameters to this method specify:

- The transient container to be made persistent.
- A system name for the container.

  If you give the container a system name, applications will be able to look up the container by its name; use an empty string or null if you don't want to look up the container by name.
- The hash values if the container is to be hashed.

  A hashed container provides an efficient lookup mechanism for scope-named objects, but occupies more storage than a non-hashed container. If you intend to use the container or any object it contains as a scope object, the container must be hashed; if not, the container should not be hashed. For more information about scope objects and scope-named objects, see "Name Scopes" on page 214.
- The initial number of pages allocated for the container.

  Select the container size based on the number of objects you plan to store in the container and the size of the objects. For a single-object container, for example, you could specify an initial size of one page. For a container that will hold a large set of objects that are rarely updated, you could set the initial size large enough to accommodate the entire set, reducing both time-consuming growth operations and wasted excess space in the final growth operation.
- The percent of its current size by which the container should grow when it needs to accommodate more basic objects.

Because a container is a persistent object as well as a storage object, you can also make it persistent in any of the ways that you make a basic object persistent (see "Making an Object Persistent" on page 162). You can explicitly cluster the container (by calling the `cluster` method of a database, a persistent container, or a persistent basic object) or you can implicitly cluster it, for example, by making it a named root. When you call `cluster` or cluster the container implicitly, you create a hashed container in the database. The container has no system name, 5 initial pages, a hash value of 10, and a growth factor of 10%.

NOTE If you want a container to be nonhashed, or if you want to give it a system name, or a non-default hash value, number of initial pages, or growth factor, you must add it to the database explicitly before performing any other operation that would make it persistent.

A container is created immediately when you add or cluster a container in a database. When you commit or checkpoint the transaction in which you make a container persistent, the container is made visible to other sessions. If the transaction is aborted, the local representation of a container that was made

persistent during the transaction goes back to the transient state and the container in the database is removed.

Retrieving a Container

You can retrieve a container in any one of the following ways:

- If the container has a system name, look it up by name by calling the `lookupContainer` method of a database.
- Obtain an iterator that finds all containers in a database with the `contains` method of the database.
- Retrieve the default container of a database with the `getDefaultContainer` method of the database.
- Get the container of a persistent basic object with that object's `getContainer` method.

Because a container is a persistent object as well as a storage object, you can also retrieve a container in any of the ways that you would retrieve a persistent object. See Chapter 11, "Retrieving Persistent Objects".

Deleting a Container

Just as the objects in a garbage-collectible container are deleted automatically by oogc when they are no longer reachable, a garbage-collectible container itself is deleted automatically by `oogc` when it is no longer needed. Just as objects in a non-garbage-collectible container must be deleted explicitly, a garbage-collectible container itself must be deleted explicitly when it is no longer needed.

Garbage Collection of Containers

The `oogc` administration tool removes unnecessary containers as well as objects from the federated database. Only garbage-collectible containers can be removed by `oogc`.

A garbage-collectible container is a candidate for garbage collection if all of the following conditions hold:

- The container does not have a system name.
- The container does not contain any valid objects; that is, all objects in the container will be removed by garbage collection.
- The container itself is not a named root or reachable from a named root.

Explicit Deletion

You delete a container with its `delete` method. You can delete either a garbage-collectible container or a non-garbage-collectible container. The `delete` method propagates the delete operation to any objects related to this container through relationships for which delete propagation is enabled. Deleting a container deletes each object that it contains and propagates the deletion operation to that object's related objects. You can delete a container without propagating through relationships with its `deleteNoProp` method.

Until the current transaction is committed, the deleted container and its basic objects continue to exist in the database.

If your application has objects in memory that represent objects of the deleted container, those objects (as well as the container object itself) are marked dead. Each deleted object (including this container) is removed from any bidirectional relationships in which it is involved. However, if another persistent object references the deleted object in a unidirectional relationship or directly in one of its persistent fields, you are responsible for removing that reference. An exception is thrown if you attempt to write a persistent object that references a dead object.

Until the current transaction is committed, the local representations of the deleted container and basic objects continue to exist in your application's memory, but they are marked dead. If the transaction is aborted, the local representations are still dead objects, but the container and its objects continue to exist in the federated database. You may retrieve the container again if you need to work with it.

Example

This example illustrates how to use multiple containers to improve scalability and concurrency. The example revisits the vehicle rental company example introduced in Chapter 1, "Getting Started". The implementation is changed as follows:

- The `Fleet` class (see page 385) uses a map instead of a fixed-size array to maintain its link with all its contained vehicles (see page 343). Chapter 6, "Defining Persistence-Capable Classes," discusses trade-offs between various implementations of links to multiple objects. The fleet remains a named root in the vehicles database.
- The vehicles themselves are randomly distributed among a fixed-size container pool (see page 387) of garbage-collectible containers, instead of being stored in a single container. This allows one container to be updated without preventing other containers from being accessed. The container pool is also a named root in the vehicles database.

  The vehicles are stored according to the container pool clustering strategy (see page 389). This strategy is installed in the sessions used by the database

initialization class `VrcInit` (see page 361) and the interactive application class `Vrc` (see page 348).

A vehicle is deleted simply by removing its entry in the fleet map. Because the vehicles are stored in garbage collectible containers, when a vehicle is removed from its fleet, it will be garbage collected when `oogc` is executed.

All the containers are referenced from a named root (the container pool). As a consequence, they will not be garbage collected even if all their objects are deleted or garbage collected.

The files for this example are in the `Storage` subdirectory of the programming samples directory.

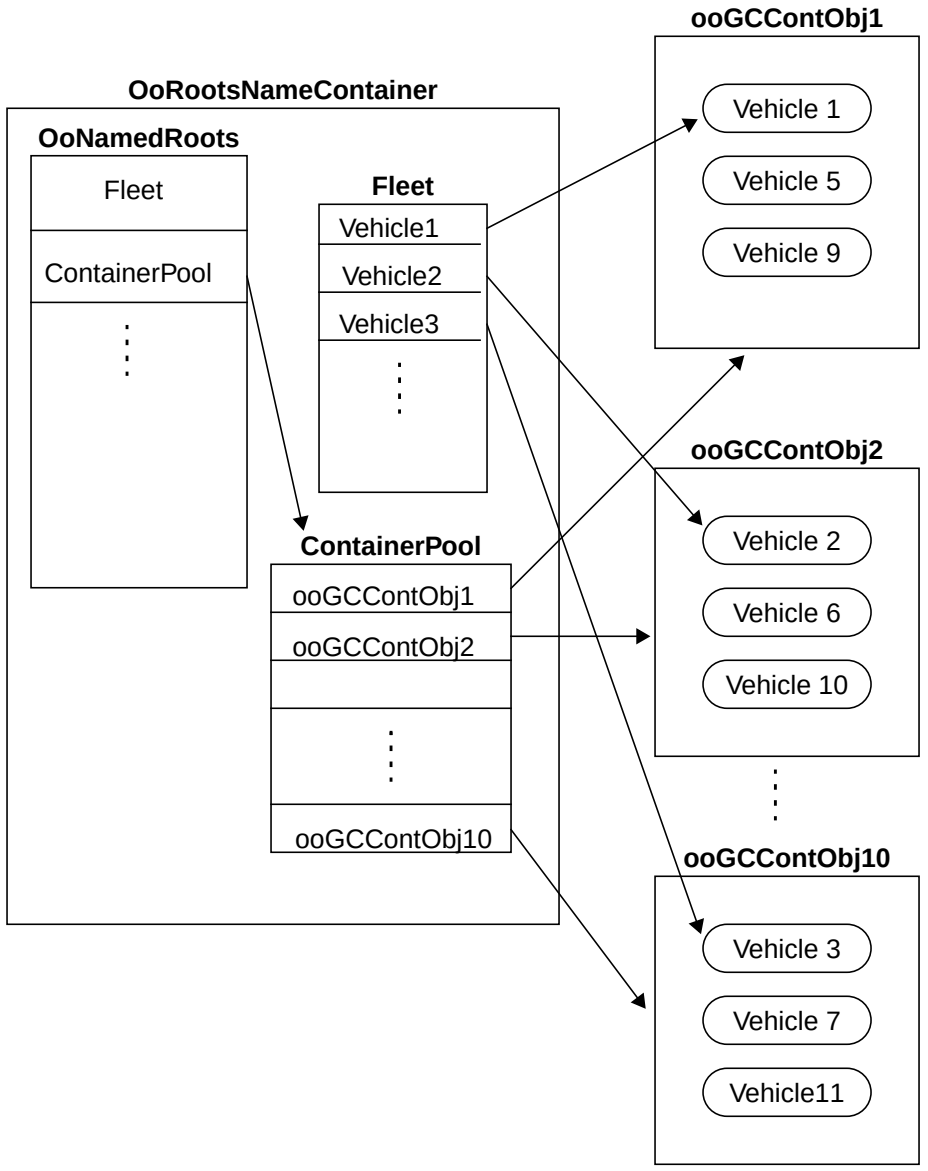
To execute this example, you need to:

1. Compile the files `Fleet.java`, `VrcInit.java`, `Vrc.java`, and `ContainerPool*.java` in the `Storage` subdirectory of the programming samples directory.
2. Start an Objectivity / DB lock server.
3. Create a federated database called `Vrc` in the `Storage` directory.
4. Execute `VrcInit` to initialize the federated database.

Once you complete these steps, you can view an actual configuration of objects with the Objectivity / DB objects and types browser. The following figure illustrates a possible configuration of objects within a database for this example.

The fleet and container names are entries in the root dictionary, which is stored in the roots container. Since the fleet and container pool objects are made persistent when the default clustering strategy is in effect, they are stored near the root dictionary (named `OoNamedRoots`), that is, in the roots container (named `OoRootsNameContainer`). See Chapter 12, “Clustering Objects,” for further information on clustering strategies.

The vehicles are stored according to the container pool clustering strategy. The figure illustrates a possible allocation of vehicles to the containers within the container pool.


ooGCContObj1
Vehicle 1
Vehicle 5
Vehicle 9
OoRootsNameContainer
OoNamedRoots
Fleet
ContainerPool
Fleet
Vehicle1
Vehicle2
Vehicle3
ooGCContObj2
Vehicle 2
Vehicle 6
Vehicle 10
ContainerPool
ooGCContObj1
ooGCContObj2
ooGCContObj10
ooGCContObj10
Vehicle 3
Vehicle 7
Vehicle11

6

Defining Persistence-Capable Classes

A *persistence-capable class* is one whose instances can be made persistent and saved in a database. When you define a persistence-capable class, you must consider its position in the inheritance hierarchies of the application, the range of persistent behavior that the class should support publicly and privately, and which of its fields contain persistent data. This chapter discusses the decisions you must make when defining a persistence-capable class and describes how to implement those decisions.

In This Chapter

Persistence-Capable Classes

A persistence-capable class supports persistent operations, allowing instances of the class to act both as normal Java runtime objects and as objects stored persistently in a federated database. An application that needs to save objects in a database must define a persistence-capable class for each kind of object to be saved.

NOTE You should not use the underscore character (_) in the name of a persistence-capable class. If you are adding persistence to an existing class whose name contains underscore characters, you must give the class a schema class name that does not contain underscores. See "Schema Class Names" on page 264.

Applications may also work with Objectivity for Java persistence-capable classes, namely classes for containers and for collections of persistent objects. An application that needs to associate persistent data with a container can define its own application-defined container classes.

Descriptions of all persistence-capable classes are stored in the schema of a federated database. Several chapters in this guide discuss various aspects of working with a schema:

- For information on how to manage a schema, see Chapter 14, "Schema Management".
- If you are defining a Java class corresponding to an existing class description in the schema of a federated database that is shared with C++ and/or Smalltalk applications, see Chapter 19, "Schema Matching for Interoperability".
- For information on what happens to a schema and any existing objects when you modify the definition of one or more persistence-capable classes, see Chapter 15, "Schema Evolution and Object Conversion".

Persistors

Every persistent object has an associated internal object called a *persistor*, an instance of a class that implements the `PooObj` interface. An object's persistor contains all the internal database states for the object and implements persistent behavior for the object.

- When a transient object is made persistent and when an existing persistent object is retrieved from the database, Objectivity for Java creates a persistor for the persistent object.
- To perform a persistent operation on a persistent object, Objectivity for Java calls the appropriate method of the object's persistor.

- If the persistent object is deleted from the federated database, Objectivity for Java replaces its persistor with a *dead persistor*, which indicates that the object no longer exists in the database and can no longer participate in persistent operations.

Persistence Behavior

Persistence-capable classes can support three general kinds of persistence behavior:

- All persistence-capable classes must have methods to get and set an object's persistor. These methods provide implicit persistence behavior in that they allow Objectivity for Java to perform persistent operations on an object of the class. They do not, however, let the object itself perform persistent operations explicitly.
- A persistence-capable class can support explicit persistence behavior by adding methods that call the appropriate methods of a persistent object's persistor.
- A persistence-capable class can support the ability to handle persistent events, which occur when certain persistent operations are performed on an object of the class.

Explicit Persistence Behavior

A persistor has methods that:

- Maintain consistency between data in the memory of a running Java application and data in persistent storage in a federated database.
- Control where persistent objects are stored physically in the database.
- Let a persistent object act as a name scope to organize and facilitate access to other persistent objects.
- Test the state of an object.

A persistence-capable class can implement corresponding methods to provide some or all of this persistence behavior explicitly.

Persistent Events

A *persistent event* is a pre- or post-processing event; when a persistent object is involved in certain persistent operations, the object receives a persistent-event notification immediately before or after the persistent operation occurs. In response to the notification, the object can perform whatever application-specific processing is required.

Persistent events occur entirely within the Objectivity for Java process space (they are not generated asynchronously by other processes and dispatched to

Objectivity for Java). These events are also session specific; that is, a persistent operation in one session affects only the persistent objects that belong to that session.

Objectivity for Java supports three kinds of persistent events:

- An *activate event* is triggered when a persistent object's data is fetched from the database.
- A *deactivate event* is triggered when the session to which this persistent object belongs is committed or aborted. This event is useful in work-flow applications because it indicates whether an object's changes were accepted or not.
- A *pre-write event* is triggered when a persistent object's data is being written to the database. An application might respond to this event by encrypting field values or nulling out fields.

Making a Class Persistence-Capable

You can make an application-defined class persistence-capable in any of four ways:

- Define the class to be a descendant class of `ooObj`.

  The class inherits default implementations for public methods that get and set an object's persistor, that perform persistent operations explicitly, and that handle persistent events. You do not need to implement any persistent behavior unless you want to modify the default implementation.

- Define the class to implement the `IooObj` interface.

  This interface provides public methods to get and set an object's persistor, to perform persistent operations explicitly, and to handle persistent events; you must implement all these methods.

  The `IooObj` interface defines the persistent operations that are available to persistence-capable classes. As such, it may change from release to release. If you define classes that implement `IooObj`, future releases of Objectivity for Java might require you to make code changes. For example, if a new method is added to the interface, you would need to implement that method for your classes.

- Define the class to implement the `PersistentEvents` interface.

  This interface has public methods to get and set the persistor and to handle persistent events; you need to implement those methods. If you desire, you can also implement public or private methods to perform persistent operations explicitly.

- Define the class to implement the `Persistent` interface.

  This interface has public methods to get and set the persistor; you need to implement those methods. If you desire, you can also implement public or private methods to perform persistent operations explicitly.

Once you have defined a persistence-capable class, any subclass you define from it inherits its persistence behavior.

You can also modify third-party classes to make them persistence-capable, but doing so requires care. See “Adding Persistence Capability to Third-Party Classes” on page 130.

The simplest way to provide the capability for persistence is to define a class that inherits from `ooObj`. One drawback is that all persistence-capable classes implemented this way form a single inheritance hierarchy with `ooObj` at the root. If your application already contains disjoint inheritance hierarchies for the classes that you want to make persistence-capable, you can preserve the hierarchies and define the classes to implement one of the persistence-capable interfaces. The choice of which interface determines whether the class can handle persistent events and whether all persistent operations are publicly accessible.

The following table lists the four ways to make a class persistence capable and shows which capabilities are available with each.

Persistence Behavior of Class	Class Definition			
	Inherits from ooObj	**Implements IooObj**	**Implements PersistentEvents**	**Implements Persistent**
Instances can be made persistent	Yes	Yes	Yes	Yes
Can get and set object’s persistor	Yes	Yes	Yes	Yes
Can handle persistent events	Yes	Yes	Yes	No
All persistent operations are public	Yes	Yes	No	No
Enforces single inheritance hierarchy	Yes	No	No	No
Includes default implementation	Yes	No	No	No

Inheriting From ooObj

You can make a class persistence-capable by subclassing `ooObj` directly, or subclassing some other application-defined class that is derived from `ooObj`. If you want to associate application-specific data with a container, you can subclass either `ooContObj` or `ooGCContObj`.

EXAMPLE In this example, `Vehicle` is a persistence-capable class whose superclass is `ooObj`; `Truck` is a persistence-capable class whose superclass is `Vehicle`.

```
import com.objy.db.app.ooObj;

// Make class persistence-capable by inheritance
public class Vehicle extends ooObj {
   …
}

// Make class persistence-capable by inheritance
public class Truck extends Vehicle {
   …
}
```

Default Handling for Persistent Events

The `ooObj` class provides the following default handling for persistent events.

- The default response to an activate event is to throw a `FetchCompletedWithErrors` exception if any errors occurred during the fetch operation. These exceptions are informational and identify any fields that were not fetched successfully.
- The default response to a deactivate event is to do nothing.
- The default response to a pre-write event is to do nothing.

If you want your class to respond differently to any of these persistent events, you must implement the appropriate behavior as described in “Handling Persistent Events” on page 122.

Getting and Setting an Object's Persistor

Unless your persistence-capable class is a descendant of `ooObj`, you must implement methods to get and set an object's persistor.

Caching the Persistor

First, decide where to cache each object's persistor. The simplest and most efficient approach is to store the persistor in a field of type `PooObj`. (The `ooObj` class uses this approach.) The field holding the persistor must be transient so that it is not stored as part of the data of a persistent object; see "Transient Fields" on page 133. If the persistor field is not transient, a schema exception will be thrown the first time the class is registered, an object of the class is stored, or an index is defined for objects of the class.

Alternatively, you could cache the persistor in some global (non-persistent) object, such as an array, vector, or hash table.

Initializing the Persistor

When a persistence-capable class is instantiated, its persistor must be initialized to null. A null persistor indicates that the newly created object is transient. When the object becomes persistent, Objectivity for Java gives it a newly created persistor. If the object becomes dead, Objectivity for Java sets its persistor to a dead persistor; see "Dead Persistent Objects" on page 190.

WARNING Only Objectivity for Java should set an object's persistor. If your application sets an object's persistor, unpredictable behavior and database corruption may result.

Implementing Methods to Get and Set the Persistor

You need to define the two methods declared in the `Persistent` interface:

- `getPersistor` retrieves the object's persistor.
- `setPersistor` sets the object's persistor.

The implementations of both methods must be synchronized for thread safety. You can refer to the Objectivity for Java class `ooObj` for an example implementation of these methods.

EXAMPLE This `Vehicle` class is made persistent by implementing `Persistent`; its methods to get and set an object's persistor follow the model set forth by `ooObj`.

```
import com.objy.db.iapp.PooObj;
import com.objy.db.iapp.Persistent;

public class Vehicle implements Persistent {
   // Get and set the persistor
   private transient PooObj persistor;

   public synchronized PooObj getPersistor() {
      return persistor;
   }

   public synchronized void setPersistor(PooObj persistor) {
      this.persistor = persistor;
   }
   …
}
```

Handling Persistent Events

You need to implement handling for persistent events in any of the following circumstances:

- Your class is a descendant of `ooObj` and you don't want to use the default event-handling mechanism.
- Your class implements `IooObj` or `PersistentEvents`.

Handler Methods for Persistent Events

A persistence-capable class that can respond to persistent events has a *handler method* corresponding to each kind of persistent event. A persistent object is notified that a persistent event has occurred by a call to the appropriate handler method. The handler method performs whatever application-specific processing is required to respond to the event.

Activate Events

A persistent object's `activate` method handles activate events. This method is called **after** the object is fetched. An activate event is triggered after execution of the `fetch` or `markModified` method of the object's persistor if the object's data had not already been fetched.

You might use the `activate` method to set appropriate values for transient fields or to handle deleted references intelligently.

Deactivate Events

A persistent object's `deactivate` method handles deactivate events. This method is called for all objects belonging to a session **after** the session's current transaction is successfully committed or aborted. If an object is made persistent during a transaction that is subsequently aborted, it is still sent a deactivate event.

You might use the `deactivate` method to allow the application to take different actions depending on whether a transaction is committed or aborted. Doing so can be useful for a user interface or in a work-flow application where an aborted transaction affects the actions of the application.

NOTE If you implement the `deactivate` method in a persistence-capable class, your implementation must not perform any Objectivity / DB operations (because it is called after the transaction has been terminated).

Pre-Write Events

A persistent object's `preWrite` method handles pre-write events. This method is called **before** the object is written to the database. A pre-write event can be triggered by any method that causes the object to be written, namely:

- The `write` method of the object's persistor.
- The `copy` method of the object's persistor.
- The `commit` or `checkpoint` method of the session to which the object belongs (or the `commit` or `checkpoint` method of the transaction object in an ODMG application).
- The `flush` method of the federated database or of the object's database or container.
- A predicate scan method if objects in the current connection are automatically written to the cache before a predicate scan operation. (See "Predicate-Scan AutoFlush Policy" on page 49.)

You might use the `preWrite` method to transform or encrypt the values of some persistent fields. Alternatively, your method could check that values in the various persistent fields are mutually consistent; if it finds a problem, it could

throw a runtime exception to prevent Objectivity for Java from writing out this particular object. In the latter case, the exception would abruptly terminate the encompassing operation.

Implementing Persistent-Event Handler Methods

Your handler methods may perform whatever application-specific processing is required in response to a persistent event.

The parameter to a handler method is a read-only information object of a class that implements the `PersistentEventInfo` interface. The information object contains information specific to the persistent event that occurred, for example, why the event was triggered.

NOTE If the same persistent operation occurs more than once, whether on the same persistent object or different objects, the information object passed in one call to the handler method may not be identical (equal) to the information object passed in the next call to the same method.

You should follow these guidelines when you implement your handler methods:

- Your handler methods must not throw any checked exceptions; if they do, you will get compilation errors when you compile your class.
- Your handler methods should not call any Objectivity for Java methods that could change an object's persistent state. For example, your methods should not call `commit`, `abort`, `flush`, `write`, `copy`, or `move`. Changing an object's persistent state while processing a persistent event can cause your application to enter an indeterminate and non-recoverable state.
- As a general rule, your handler methods should access only the persistent object being notified. Methods such as `preWrite` and `deactivate` may be called frequently. Thus, the longer it takes these methods to execute, the longer it takes the corresponding persistent operation to complete. Objectivity for Java does not create a separate notification thread to perform these operations.

Exceptions in Handler Methods

A number of rules govern how Objectivity for Java handles uncaught exceptions thrown by persistent-event handler methods. When exceptions are thrown, it is important to define the state of the objects, whether they are marked as modified or as requiring to have their data fetched.

Exceptions While Handling an Activate Event

Any exception thrown by an object's `activate` method can affect the state of that object. The following table lists methods that can trigger an activate event and shows the resulting object states and the exception propagation when the `activate` method throws an exception.

Method Triggering the Activate Event	State of Notified Object After the Exception	
	Default activate Method	Application-Defined activate Method
`fetch`	Object is marked as not needing its data fetched.	Object is marked as needing its data fetched. The exception is propagated.
`markModified`	Object is marked as needing its data fetched. Object is marked as modified. The exception is propagated.	Object is marked as not needing its data fetched. Object is marked as not modified. The exception is propagated.

Exceptions While Handling a Deactivate Event

Since more than one object may be notified of this event at any one time, it is not reasonable for Objectivity for Java to stop whenever some object's `deactivate` method throws an exception. Instead, Objectivity for Java silently consumes such exceptions and continues to notify the remaining objects.

A deactivate event is a courtesy notification in that any exceptions thrown by notified objects' `deactivate` methods do not affect the commit or abort operation that triggered the event. The database operation completes and the objects are updated according to their fetched and modified state.

If an aborted transaction triggered the event:

- An object made persistent during this transaction reverts to being transient but its `deactivate` method is still called.
- An object that was the result of a copy is made a dead object but its `deactivate` method is still called.

An object deleted during a transaction is removed from the session, so its `deactivate` method is not called when the transaction is committed or aborted. If the transaction is aborted, the deleted objects still exist in the database. However, its corresponding Java object has been marked dead; the object needs to be retrieved again before the application can access it.

Exceptions While Handling a Pre-write Event

The `preWrite` method is called for each object that is written by the persistent operation that triggered the pre-write event. If some object's `preWrite` method throws an exception, the operation terminates abruptly; the state of the object throwing the exception is not changed (that is, it is still marked as modified) and the exception is propagated. If the operation involves other objects, the state of each object depends on whether it is written to the cache prior to the exception or after the exception.

The following table illustrates the object states if an exception is thrown.

Method Triggering the Pre-Write Event	State of Notified Objects After the Exception
`write`	The object is still marked as modified.
`copy`	The object is not copied; no new instance is returned.
`flush`	Because the processing order during a flush operation is not specified, the remaining objects are not written to the cache and the exception causes abrupt termination of the flush operation. The object that throws the exception is still marked as modified.
`commit` `checkpoint`	No object is written to the database. Objects written to the cache prior to the exception are marked as not modified. The operation terminates. The object that throws the exception is still marked as modified.
Predicate scan	Same as above. No iterator instance is created or returned.

Providing Explicit Persistence Behavior

The methods in the `IooObj` interface explicitly provide the full range of persistence behavior that is available implicitly through an object's persistor. If your class implements `IooObj`, you must implement all these methods. If your class instead implements `Persistent` or `PersistentEvents`, you may implement any of these methods that you choose.

Your implementation of an explicit persistence method should take the appropriate action if the object is transient or if it is a dead object.

- To see what behavior is appropriate for a transient object, refer to the implementation notes for the method in its description within the `IooObj` reference documentation. For example, implementation notes for the `fetch` method indicate that the method should do nothing if the object is transient.

- If an object is dead, you can test its state, but you may not perform any persistent operations on it; see “Dead Persistent Objects” on page 190. Any method that performs a persistent operation should verify that the object is not dead before calling the appropriate method of its persistor. Calling methods of a dead persistor may have unexpected results.

EXAMPLE The `Vehicle` class implements the `IooObj` interface. Its `markModified` and `fetch` methods forward the call, when appropriate, to the object’s persistor.

```
import com.objy.db.iapp.IooObj;
import com.objy.db.iapp.PooObj;

public class Vehicle implements IooObj {
   private transient PooObj persistor = null;
   ...
   // Explicit persistence behavior
   void fetch() {
      if (persistor.isDead())
         throw new ObjectIsDeadException(
      "Attempted persistent operation on dead object");
      // Do nothing if object is transient
      if (persistor != null)
         persistor().fetch();
   }

   void markModified() {
      if (persistor.isDead())
         throw new ObjectIsDeadException(
   "Attempted persistent operation on dead object");
      // Do nothing if object is transient
      if (persistor != null)
         persistor().markModified();
   }
   ...
}
```

To ensure that your class accesses the persistor appropriately, you can copy the implementation of the various persistence methods from `ooObj` to your class. Note, however, that `ooObj` uses internal methods that throw exceptions if the object is transient or dead. If you copy implementations from `ooObj`, be sure to copy definitions of these internal methods as well.

EXAMPLE This `Vehicle` class implements the `IooObj` interface. Its persistence methods, copied from `ooObj`, use the internal methods `persistor` and `notDeadPersistor`.

```
import com.objy.db.iapp.IooObj;
import com.objy.db.iapp.PooObj;

public class Vehicle implements IooObj {
   // Get and set the persistor
   …
   // Internal methods
   private synchronized PooObj persistor() {
      if (persistor == null)
         throw new ObjectNotPersistentException(
   "Attempted persistent operation on transient object");
      if (persistor.isDead())
         throw new ObjectIsDeadException(
   "Attempted persistent operation on dead object");
      return persistor;
   }

   private synchronized PooObj notDeadPersistor() {
      if (persistor.isDead())
         throw new ObjectIsDeadException(
   "Attempted persistent operation on dead object");
      return persistor;
   }

   // Explicit persistence behavior
   void fetch() {
      if (persistor != null)
         notDeadPersistor().fetch();
   }

   public void write() {
      persistor().write();
   }
   …
}
```

Delegating Persistent Operations

Classes that implement either `Persistent` or `PersistentEvents` need not define persistence methods. However, each time an object of such a class needs to perform a persistent operation, it must test that its persistor is valid for the desired operation. You avoid repeating the necessary tests with each call and avoid implementing persistent methods for several classes by defining a "delegator" class whose sole purpose is to provide persistence behavior.

- Static methods of the delegator class take a persistor as a parameter. These methods perform any necessary tests on the persistor to see whether the requested operation is allowed. If so, the static methods delegate the operation to the persistor by calling the corresponding method of the persistor.
- Any persistence-capable class can perform a persistent operation by passing its persistor as a parameter to the appropriate static method of the delegator class.

EXAMPLE This example shows a few methods of a class `Delegator` whose role is to implement persistence behavior. Any number of persistence-capable classes that implement `Persistent` or `PersistentEvents` could use the `Delegator` class. The complete class definition appears in the `Delegator` programming example. (see page 416)

```
public class Delegator {
   // Internal methods
   private static synchronized PooObj notDeadPersistor(
          PooObj persistor) {
      if (persistor.isDead())
         throw new ObjectIsDeadException(
   "Attempted persistent operation on dead object");
      return persistor;
   }
   ...
   // Explicit persistence behavior
   public static void markModified(PooObj persistor) {
      if (persistor != null)
         notDeadPersistor(persistor).markModified();
   }
   ...
}
```

Adding Persistence Capability to Third-Party Classes

It is possible to add persistence capability to any class by implementing one of the persistence-capable interfaces. Before doing so however, you should keep in mind the following caveats:

- Many classes are not designed to be persistence-capable. Some classes may be designed to be serializable, but serializability is different from persistence capability.
- Certain classes, such as `Hashtable`, that would store an object's hash code (returned by the `hashCode` method) as a way of identifying an object, *cannot* be made persistent because a hash code is valid only during the execution of the Java Virtual Machine in which that hash code was generated.
- Any change to the internals of a third party class—inheritance, field names, field types, or field modifiers—will cause schema evolution and object conversion. See Chapter 15, "Schema Evolution and Object Conversion," for information on these topics.
- The accessor methods of a class made persistence-capable will need to be modified as described in "Field Access Methods" on page 138. Any methods that access the persistent fields of the class (for example, methods of the class itself or other classes in the same package) should call its accessor methods instead of accessing the fields directly.

Defining Fields

Regardless of how you make your class persistence-capable, you will follow the same approach when defining fields of the class. Fields can serve two roles for an object. They can capture the state associated with an object or they can link an object to other objects. Persistent objects can have persistent fields, whose values are saved in the database, and transient fields, whose values are not saved.

Persistent Fields

All non-static and non-final fields you define for a persistence-capable class are *persistent* by default. The values in the persistent fields of a persistent object constitute that object's *persistent data.* When the object is written to the federated database, the values in those fields are saved persistently.

NOTE Only *application-defined* fields are considered persistent fields. Any fields defined by Objectivity for Java (for example, inherited fields of `ooObj`, container classes, and persistent-collection classes) are considered part of the internal representation

of the object. We refer to the contents of those fields as *properties* of the object, not the object's persistent data.

Every persistent field must be of one of the following data types:

Category	Types

Java primitive type	`char`
	`byte`
	`short`
	`int`
	`long`
	`float`
	`double`
	`boolean`

Java string class	`String`
	`StringBuffer`
	
	Note: If you set the value of a `String` persistent field to be an empty string (""), the field will be stored in the database as *null*. When the object containing the `String` field is later read back, the field in the Java object will likewise be set to null.

Java date or time class	`java.util.Date`
	`java.sql.Date`
	`java.sql.Time`
	`java.sql.Timestamp`
	
	Note: An object of a date/time class in a persistent field is stored in the federated database as an internal persistent object.

Persistence-capable class	`ooObj` An application-defined persistence-capable class A persistent-collection class A container class Any interface **Note:** Although the declared type of a persistent field may be any interface, the actual object referenced by the field must be a persistent object.

Category	Types
Java array of any of the preceding types	For example, `long[]` or `String[]` **Note:** An array in a persistent field is stored in the federated database as an internal persistent object. Each element of a `String` array is also stored as an internal persistent object.
An Objectivity for Java relationship	See Chapter 7, “Relationships” for information on relationships.

EXAMPLE This example illustrates the persistent fields of a class called `Vehicle`.

```
public class Vehicle extends ooObj {
   // Persistent fields
   protected String license;
   protected String type;
   protected int doors;
   protected int transmission;
   protected boolean available;
   ...
}
```

As the preceding table indicates, you cannot create persistent fields that reference the database and federated database where an object is stored, the session to which the object belongs, or the object’s identifier. You can, however, obtain this information through methods defined on `ooObj` (or on the object’s persistor).

EXAMPLE This code fragment illustrates the methods for retrieving the Objectivity for Java properties of a persistent object that implements explicit persistence behavior. The complete method definition appears in the `RentalFields.Vehicle` programming example (see page 392).

```
public static void printInfo(Vehicle vehicle) {
   // This method must be called during a transaction

   // Get vehicle’s container
   ooContObj cont = vehicle.getContainer();
   // Get vehicle’s database
   ooDBObj db = cont.getDB();
   // Get vehicle’s federated database
   ooFDObj fd = db.getFD();
   // Get vehicle’s session
   Session session = vehicle.getSession();
```

```
    // Get vehicle's object identifier
    ooId oid = vehicle.getOid();
    …
}
```

If your Java application will interoperate with applications written in C++ and / or Smalltalk, you must select field types that will map to Objectivity / DB data types that are supported by the other languages. For more information on this topic, see Chapter 19, "Schema Matching for Interoperability".

NOTE If a persistent object has an array field, when you fetch the object's persistent data, you fetch the entire array and all its elements from the federated database. When you write the object, you write the entire array and all its elements to the federated database.

Transient Fields

Your class can also have *transient fields*, whose values are not saved when an object is written to the database. To specify that a field is transient, simply give it the `transient` modifier when you define your class. Transient fields are not modified when the object is read from the database or copied when a persistent object is copied. You can, however, set the value of a transient field after the object is read from the database.

You *must* define a field as transient in either of the following circumstances:

- The field's declared type is not one of the supported types for persistent fields.
- The field's declared type is an interface and you intend to reference non-persistent objects in that field.

EXAMPLE This example illustrates a transient field `dailyRate` of the class `Vehicle`.

```
public class Vehicle extends ooObj {
    …
    // Transient field
    protected transient int dailyRate;
    …
}
```

Linking Objects Together

Many applications work with *object graphs*, directed graph data structures that consist of objects linked to other objects. Objectivity / DB provides three mechanisms for linking objects together:

- Persistent fields that reference the associated objects.
- Membership of objects in persistent collections.
- Relationships between the objects.

Fields With Object References

As in any Java application, you can use fields containing object references to link objects together. A field whose type is a persistence-capable class can represent a link to the object referenced by that field. A field whose type is a Java array of objects of a persistence-capable class can represent links to the objects in the array.

EXAMPLE In this example, the `Vehicle` class has a persistent field `fleet` to link a vehicle to its rental fleet. The `Fleet` class has a persistent field `vehicles` containing a fixed-sized array of one thousand vehicles; this field serves to link a rental fleet to all the vehicles in the fleet.

```
public class Vehicle extends ooObj {
   // Persistent fields
   ...
   protected Fleet fleet;
   ...
}
```

```
public class Fleet extends ooObj {
   static final int FLEET_SIZE = 1000;
   // Persistent fields
   protected Vehicle[] vehicles =
      new Vehicle[FLEET_SIZE];
   ...
}
```

Membership in Persistent Collections

Objects can also be linked together by their membership in persistent collections. A persistent collection can be saved directly (for example, as a named root), or it can be referenced in a persistent field of a persistent object.

You can use a persistent collection to link one object to a group of objects. Instead of defining a field containing a Java array of persistent objects, you can define a field containing a persistent collection. One advantage of collections is that they are of variable size, whereas a Java array's size is fixed. A collection can grow or shrink as needed.

EXAMPLE In this example, the `vehicles` field of the `Fleet` class has been replaced by a field of type `ooMap`. Instead of an array of one thousand elements, the rental fleet's `vehicles` field now contains a map that associates each vehicle in the fleet with an identifying string, such as its license ID. At any time, the map contains only as many vehicles as are in the fleet, which may be more or less than one thousand.

```
public class Fleet extends ooObj {
   // Persistent fields
   protected ooMap vehicles = new ooMap();
   ...
}
```

See Chapter 9, "Persistent Collections" for additional details about persistent collections.

Relationships

Objectivity / DB provides a mechanism called *relationships* as an alternative way to link objects together. Objectivity / DB relationships provide a higher level of functionality than referencing objects directly from persistent fields. You can specify the directionality and cardinality of relationships, whether operations on objects propagate along relationships, and how relationships are handled when you create a new copy or a new version of an object. See Chapter 7, "Relationships," for a complete discussion of Objectivity / DB relationships and how to define and use relationships in Objectivity for Java.

EXAMPLE This example substitutes bidirectional relationships for the `fleet` and `vehicles` fields in the preceding example. The `Vehicle` class has a one-to-one relationship `fleet` that relates a vehicle to its fleet. The `Fleet` class has a one-to-many relationship `vehicles` that relates a rental fleet to the vehicles it contains. The two relationships are inverses; that is, if a given vehicle's `fleet` relationship links it to a given fleet, that fleet's `vehicles` relationship links the fleet to the vehicle.

```
package RentalRelations;
import com.objy.db.app.*;

...
public class Vehicle extends ooObj {
   ...
   // Relationships
   private ToOneRelationship fleet;
   protected static ManyToOne fleet_Relationship() {
   return new ManyToOne(
      "fleet",                  // This relationship
      "RentalRelations.Fleet",  // Related class
      "vehicles",               // Inverse
      Relationship.INLINE_NONE); // Store non-inline
   }
}
```

```
package RentalRelations;
import com.objy.db.app.*;

public class Fleet extends ooObj {
   // Relationships
   private ToManyRelationship vehicles;
   protected static OneToMany vehicle_Relationship() {
   return new OneToMany(
      "vehicles",               // This relationship
      "RentalRelations.Vehicle", // Related class
      "fleet",                  // Inverse
      Relationship.INLINE_NONE); // Store non-inline
   }
}
```

Performance Considerations

An array field containing persistent objects has performance overhead relative to a persistent collection field or a relationship. Because the array is part of the persistent data of the containing object, the entire array and all its elements are

read from the federated database when you fetch the containing object's data. Similarly, the entire array and all its elements are written to the federated database when you write the containing object's data—even if you did not modify the array or any of the persistent objects it contains.

In contrast, if you use a persistent collection or a relationship to link objects together, the destination objects are read only if they are accessed and they are written only if they are modified.

Defining Access Methods

When you retrieve a persistent object, you obtain an empty, unlocked object. You need to fetch the object's data before you can safely access the object's persistent fields or relationships; the methods that fetch data also lock the object. You can ensure that objects of your class are used safely by accessing fields and relationships only through access methods that fetch data and obtain locks as necessary.

An additional advantage of using access methods is that they hide the implementation you have chosen, which simplifies the update process if you change your implementation during the prototyping or development phases of your project. For example, suppose you decide to replace a field with a relationship or vice versa. You would need to reimplement only your access methods; the code that calls the access methods would remain unchanged. This kind of implementation change modifies the class description in the schema. As a consequence, if you change the implementation of your class after you have deployed your application, you will need to provide a conversion application to convert objects in the federated database from the old implementation to the new implementation. See Chapter 15, "Schema Evolution and Object Conversion".

If your persistence-capable class is derived from `ooObj` or implements `IooObj`, your access methods can call the `fetch` and `markModified` methods of the persistent object. If your persistence-capable class instead implements the `Persistent` or `PersistentEvents` interface, your access methods must call the `fetch` and `markModified` methods of the object's persistor. The easiest approach is to implement `fetch` and `markModified` methods for your class as described in "Providing Explicit Persistence Behavior" on page 126. The following descriptions assume that all your persistence-capable classes have `fetch` and `markModified` methods.

Field Access Methods

You should define field access methods for every persistent field of a class and call those methods whenever you get or set the value of a persistent field.

- It is not safe to access the persistent fields of a persistent object before the object has been locked for read and its persistent data has been fetched. To ensure that objects of your class are used safely, the access method that gets the value of a persistent field should call the object's `fetch` method before returning the field's value.
- Persistent fields of an object should be modified only if the object's container is locked for write and the object's data has been fetched. If changes are made, the object must be marked as modified so that the changes will be written to the database. To ensure that your objects are modified safely and that changes are written to the database, the access method that sets the value of a persistent field should call the object's `markModified` method before changing the field.

NOTE Any access method that calls the `fetch` or `markModified` method of a *persistent* object must be called when that object's session is in a transaction. While the object is transient, `fetch` or `markModified` have no effect, so they can be called outside a transaction.

Scalar Fields

If a persistent field has a scalar type, you can define one access method to get the scalar value in the field. If the value in the field can be changed, you can define another access method to set the value in the field.

EXAMPLE This example illustrates some field access methods for the `Vehicle` class. Each persistent field has an access method that gets the value of that field; the `getLicense` method illustrates the form of these methods. The `setFleet` access method sets the `fleet` field to the specified fleet. All other persistent fields are initialized when a vehicle object is created; only the `available` field can be modified after initialization. The field access methods `rentVehicle` and `returnVehicle` set the Boolean field `available` to false and true, respectively. The complete class definition appears in the `RentalFields.Vehicle` programming example (see page 392).

```
// Field access methods to get persistent field values
public String getLicense() {
   fetch();
   return this.license;
}
...
```

```
// Field access methods to set fields
public void setFleet(Fleet fleet) {   // Set fleet field
   markModified();
   this.fleet = fleet;
}

public void rentVehicle() {           // Set available field
   markModified();
   this.available = false;
}

public void returnVehicle() {        // Set available field
   markModified();
   this.available = true;
}
```

If your class that implements the `Persistent` or `PersistentEvents` interface uses a delegator class instead of implementing explicit persistence behavior (see "Delegating Persistent Operations" on page 129), then your field access methods should call the static `fetch` and `markModified` methods of the delegator class.

EXAMPLE The following field access method uses a delegator. The complete class definition appears in the `PersistentInterface.Vehicle` programming example (see page 410).

```
// Field access methods to get persistent field values
public void getLicense() {
   Delegator.fetch(this.getPersistor());
   return this.license;
}

public void setFleet(Fleet fleet) {   // Set fleet field
   Delegator.markModified(this.getPersistor());
   this.fleet = fleet;
}
```

Array Fields

If a persistent field contains a Java array, you can define access methods to get and set the value at a particular array index. If necessary, you can also define an access method to return the array itself.

EXAMPLE This example illustrates access methods for a persistent field containing an array. The `getVehicle` method gets the value at a particular array index and the `setVehicle` method sets the value at a particular index. The complete class definition appears in the `RentalFields.SimpleFleet` programming example (see page 396).

```
public Vehicle getVehicle(int n) {
   fetch();
   return this.vehicles[n];
}

public void setVehicle(
      int n,
      Vehicle newMember) {
   markModified();
   this.vehicles[n] = newMember;
}
```

An alternative approach to accessing array fields directly is to hide the implementation with field access methods that manage the array.

EXAMPLE In this example, a new persistent field, `numberOfVehicles`, keeps track of the number of vehicles in the fleet. The `addVehicle` method adds a vehicle to the fleet; the `deleteVehicle` method deletes a vehicle; the `findVehicle` method gets the vehicle with the specified license ID. The `getAllVehicles` method gets an enumeration that finds all vehicles in the fleet; this method uses an inner Enumeration class called `VehicleItr`. The complete class definition appears in the `RentalFields.Fleet` programming example (see page 397).

```
public void addVehicle(Vehicle newMember) {
   markModified();
   if (findVehicle(newMember.getLicense())
         == null) {
      this.vehicles[this.numberOfVehicles] =
         newMember;
      this.numberOfVehicles++;
   }
}
```

```
public void deleteVehicle(Vehicle vehicle) {
   markModified();
   int i = 0;
   while ((i < this.numberOfVehicles) &&
          (this.vehicles[i] != vehicle)) {
      i++;
   }
   if (i != this.numberOfVehicles) {
      // Vehicle was found; remove it
      for (int j = i + 1;
            j < this.numberOfVehicles;
            j++, i++)
         this.vehicles[i] = this.vehicles[j];
      this.numberOfVehicles--;
   }
}

public Vehicle findVehicle(String license) {
   fetch();
   if (this.numberOfVehicles == 0)
      return null;
   for (int i = 0;
          i < this.numberOfVehicles; i++) {
      if (this.vehicles[i].getLicense().equals(license))
         return this.vehicles[i];
   }
   return null;
}

public Enumeration getAllVehicles() {
   fetch();
   return new VehicleItr(this);
}
```

Persistent Collection Fields

If a persistent field contains a persistent collection, your field access methods can get and set the persistent collection. An alternative approach is to hide the implementation you have chosen for the field. Instead of methods that get and set the entire collection, you can define access methods that get and set elements of the collection.

EXAMPLE This example illustrates access methods for a persistent field containing a map. The `addVehicle` method adds a vehicle to the map; the `deleteVehicle` method deletes a vehicle; the `findVehicle` method gets the vehicle with the specified license ID; the `getAllVehicles` method returns an iterator that finds all vehicles in the map. Note that the access methods that modify the map call `fetch` rather than `markModified`. The call to `fetch` retrieved the fleet's persistent data, including the map; the map itself obtains the necessary locks and ensures that the map is written if it is modified. The complete class definition appears in the `RentalMap.Fleet` programming example. (see page 402)

```
public void addVehicle(Vehicle newMember) {
   fetch();
   String key = newMember.getLicense();
   if (this.vehicles.isMember(key))
      return;
   this.vehicles.add(newMember, key);
   this.numberOfVehicles++;
}

public void deleteVehicle(Vehicle vehicle) {
   fetch();
   String key = vehicle.getLicense();
   if (this.vehicles.isMember(key)) {
      this.vehicles.remove(key);
      this.numberOfVehicles--;
   }
}

public Vehicle findVehicle(String license) {
   fetch();
   if (this.vehicles.isMember(license))
      // Cast retrieved object to class Vehicle
      return (Vehicle)this.vehicles.lookup(license);
   else
      return null;
}

public Iterator getAllVehicles() {
   fetch();
   return this.vehicles.elements();
}
```

Relationship Access Methods

Chapter 7, "Relationships," describes how you specify and work with relationships. Briefly, a class has a special field for each relationship; when an object of the class is made persistent, the field is automatically initialized to contain a relationship (of one of the classes `ToOneRelationship` or `ToManyRelationship`). You call methods of that relationship to link the object to related objects and to get its related objects.

To ensure that relationships are used correctly, you should define relationship access methods for every relationship of a class and call those methods whenever you need to get related objects or establish relationships with other objects.

- The relationship in an object's relationship field is not read from the database until you fetch the object's data. You should call the object's `fetch` method before accessing its relationship.
- The methods of a relationship object obtain the locks they need; this means that the methods of a relationship object will throw a `LockNotGrantedException` if the session is unable to obtain whatever lock is needed for the particular operation.
- The methods of a relationship object ensure that modifications to the relationship are written to the database; your relationship access methods do not need to call `markModified`.
- Retrieved objects must be cast to the appropriate class. Your relationship access methods can perform the necessary cast and return related objects of the correct class.

To-One Relationships

EXAMPLE This example illustrates access methods for a to-one relationship. The `setFleet` method sets the related fleet; the `getFleet` method gets the related fleet. The complete class definition appears in the `RentalRelations.Vehicle` programming example (see page 406).

```
// Relationship access methods
public void setFleet(Fleet fleet) {
   fetch();
   this.fleet.clear();  // Remove any existing relationship
   this.fleet.form(fleet);
}
```

```
public Fleet getFleet() {
   fetch();
   return (Fleet)this.fleet.get();  // Cast to Fleet
}
```

To-Many Relationships

EXAMPLE This example illustrates access methods for a to-many relationship. The `addVehicle` method adds a related vehicle; the `deleteVehicle` method removes a vehicle from the relationship; the `findVehicle` method gets the related vehicle with a particular license ID; the `getAllVehicles` method returns an iterator that finds all related vehicles. The complete class definition appears in the `RentalRelations.Fleet` programming example (see page 408).

```
// Relationship access methods
public void addVehicle(Vehicle newMember) {
   fetch();
   if (this.vehicles.includes(newMember))
      return;
   this.vehicles.add(newMember);
   this.numberOfVehicles++;
}

public void deleteVehicle(Vehicle vehicle) {
   fetch();
   if (this.vehicles.includes(vehicle)) {
      this.vehicles.remove(vehicle);
      this.numberOfVehicles--;
   }
}

public Vehicle findVehicle(String license) {
   fetch();
   String predicate = new String("license == \"" +
                                 license + "\"");
   Iterator itr = this.vehicles.scan(predicate);
   if (itr.hasNext())
      return (Vehicle)itr.next(); // cast to Vehicle
   else
      return null;
}
```

```
public Iterator getAllVehicles() {
   fetch();
   return this.vehicles.scan();
}
```

Defining Application-Required Methods

You can define any methods for your persistence-capable classes that your applications require. Note, however, that Objectivity / DB saves only data persistently, not methods. Thus, if more than one application needs to use persistent objects of a given class, each application must have a definition of that class that includes both declarations for its persistent fields and implementations for its application-defined methods.

NOTE If you define a `finalize` method for a persistence-capable class, you must not perform any persistent operation in that method. Different implementations of the Java virtual machine may execute `finalize` methods in a separate thread. If a `finalize` method running in its own thread accesses a persistent object whose session enforces the restricted thread policy, a `NotJoinedException` will be thrown.

7

Relationships

A standard practice in object modeling is to capture the associations or links between the objects in a system. You can implement such links by first defining fields of a class to reference other objects and then filling in those fields when objects are created. You are responsible for managing the link each time the linked objects are modified.

Objectivity / DB provides a capability for implementing links, called *relationships*, that provide a higher level of functionality than simply using references to objects. Objectivity / DB maintains relationships in the database. Operations on a group of related objects, known as a *composite object*, are handled by the database, thus reducing the amount of work you have to do to accomplish such tasks.

In This Chapter

Objectivity/DB Relationships

Objectivity / DB allows you to specify the directionality and cardinality of relationships, how relationships should be handled when objects are copied or versioned, whether operations on objects propagate along relationships, and how relationships should be stored. This section describes the properties and behavior of Objectivity / DB relationships.

Relationship Directionality

Relationship *directionality* is defined by the declaration of traversal paths that enable applications to locate related objects. When a traversal path from class A to class B is declared, the relationship is *unidirectional*. When two traversal paths, one from class A to class B and an inverse path from class B to class A, are declared, the relationship is *bidirectional*.

An object that maintains a unidirectional relationship can locate its related object, but the related object cannot locate the relating object. Unidirectional relationships correspond most closely to fields that contain object references in a standard Java object model.

Bidirectional relationships allow two related objects to locate each other. These relationships can be connected and disconnected with a *single* method invocation; adding or removing a relationship in one direction simultaneously adds or removes the inverse relationship. In addition, bidirectional relationships provide Objectivity / DB with enough information to maintain referential integrity; when an object is deleted, all relationships referencing that object are also deleted, reducing the likelihood of dangling object identifiers.

In contrast, it is not possible to ensure that a unidirectional relationship references a valid object. Unidirectional relationships do however, require somewhat less overhead and offer better performance than bidirectional relationships.

If you are modeling a salesperson and the purchasing contacts they maintain, then you must choose whether to model this as a unidirectional relationship giving the salesperson access to the contacts, or as a bidirectional relationship giving salesperson and contact objects access to each other. If it is necessary to be able to find the salesperson responsible for a given contact, then a bidirectional relationship should be used.

Relationship Cardinality

A relationship's *cardinality* indicates the number of objects on one side of a relationship that may be related with objects on the other side. Objectivity / DB relationships support four categories of cardinality:

- one to one
- one to many
- many to one
- many to many

NOTE Objectivity / DB many-to-many relationships must be *bidirectional*.

In the example of the salesperson with many purchasing contacts, the salesperson has a one-to-many relationship with the contacts. For a bidirectional relationship, the contacts have a many-to-one relationship with the salespersons. If the salespersons share contact information, then a many-to-many relationship would be appropriate.

Object Copying and Versioning

Objectivity / DB allows you to specify how relationships are handled when a copy or new version of an object is created. The possible options are:

- Delete the relationship in the copy or new version of the object and leave it in the original object. This is the default behavior.
- Move the relationship from the original object to the copy or new version of the object.
- Copy the relationship from the original object to the copy or new version of the object.

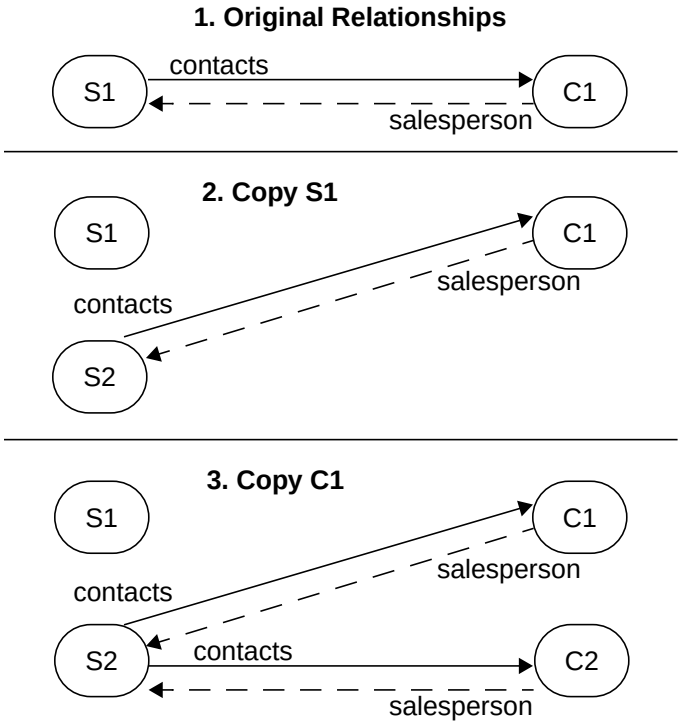
NOTE Versioning objects is *not supported* in Objectivity for Java. You should, however, specify versioning behavior when a relationship is defined if you want the class to be interoperable with other language interfaces.

The copy behavior for one path of a bidirectional relationship has the same affect on the inverse path of that bidirectional relationship. However, you can specify move for one direction of the relationship paths and copy for the other. When an object of one of the related classes is copied, the relationship path *from* that object *to* the related object is handled as requested by the copy behavior specifier for the class.

When a salesperson leaves a company, all their contacts are normally transferred to a different salesperson. One way to implement this would be to make a copy of the original salesperson, letting Objectivity/DB automatically move all the relationships from the old to new salesperson, update the new salesperson's individual data, and then delete the old salesperson.

A given salesperson should, however, be able to make a copy or new version of a contact and maintain a relationship with both the original and new objects. This behavior can be obtained by specifying that when a contact is copied, a relationship with a salesperson should be copied from the original to the new object.

The relationships that result when objects of both classes are copied are illustrated in the following figure. Salesperson specifies copy behavior as move; contacts specify copy behavior as copy. When the salesperson `S1` is copied, the salesperson relationship from the contact `C1` to `S1` is moved, relating `C1` with the salesperson `S2` and `S2` to `C1`. The bidirectional relationship between `C1` and `S1` is removed. On the other hand, when contact `C1` is copied to contact `C2`, the contacts relationship from `S2` to `C1` is copied. The original bidirectional relationship between `C1` and `S2` is kept and a new bidirectional relationship between `C2` and `S2` is created.

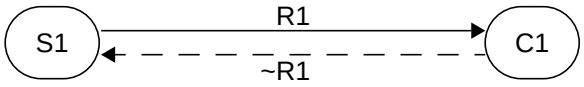


Propagating Operations

You can define relationships so that a delete or lock operation will *propagate* from one object to the next along the relationship. Propagation is a very useful property when you wish to treat related objects as a group. You specify which operations should propagate, and the direction of propagation, when you define the relationships in your classes. Propagation along a relationship is optional, and the default behavior for both delete and lock is non-propagation.

When a propagating operation is applied to an object, Objectivity / DB first identifies all objects that are affected (by identifying relationships that are declared to have propagation). It then applies the operation to all affected objects in a single atomic operation. This guarantees that a propagating operation will eventually terminate, even though the propagation graph may contain cycles. In the example we have been discussing, salesperson and contact objects are fairly loosely coupled. Thus propagation of locking and deleting operations between related objects would be disabled in both directions of the relationship.

For example, suppose your application has a salesperson `S1` that maintains a bidirectional association with a contact `C1`. A bidirectional association means that `S1` has a reference `R1` to `C1` and `C1` has a reference `~R1` to `S1`.



If you enable delete propagation, deleting `S1` would cause `C1`, `R1`, and `~R1` to be deleted automatically. Because the relationship is bidirectional, referential integrity is automatically enabled. Thus if you deleted `C1`, `R1` would automatically be deleted.

Relationship Storage

Objectivity / DB relationships can be stored *non-inline* or *inline*. The default storage mode is non-inline.

Non-Inline Relationships

A *non-inline* relationship is stored in a *system default relationship array*. Each object with relationships has a system default relationship array in which all non-inline relationships are stored. In the array, each relationship is identified by the relationship name (an identifier, not a string) and the object identifier (OID) of the related object. To trace a particular relationship on an object, Objectivity / DB traverses the relationships in the relationship array until it locates the desired relationship.

Inline Relationships

You can also define *inline* relationships. To-one inline relationships are embedded as fields of an object, while to-many inline relationships are placed in their own array instead of the system default relationship array.

There are two types of inline relationships. A *long* inline relationship uses a long OID to refer to the related object; a *short* inline relationship uses a short OID to refer to the related object. A short inline relationship uses less storage space to maintain the relationship, resulting in better runtime performance. However, you can use a short inline relationship only for objects in the same container.

NOTE For bidirectional relationships, both traversal paths must have the same storage properties. If one path is inline, the other path must also be inline. If one path is short inline, the other path must also be short inline.

Storage Requirements for Relationships

The standard storage overhead for a basic object is 14 bytes. This overhead is constant and is independent of an application's use of relationships. The following storage requirements are for unidirectional relationships. Each bidirectional relationship requires storage equivalent to two unidirectional relationships.

A *non-inline* relationship requires the following additional space:

- 4 bytes for the reference to the system default relationship array, whether there are related objects or not.
- 14 bytes for the system default relationship array, if there are any related objects.
- 12 bytes per related object.

An *inline to-one* relationship requires the following additional space:

- 8 bytes for a long reference, whether there is a related object or not.
- 4 bytes for a short reference, whether there is a related object or not.

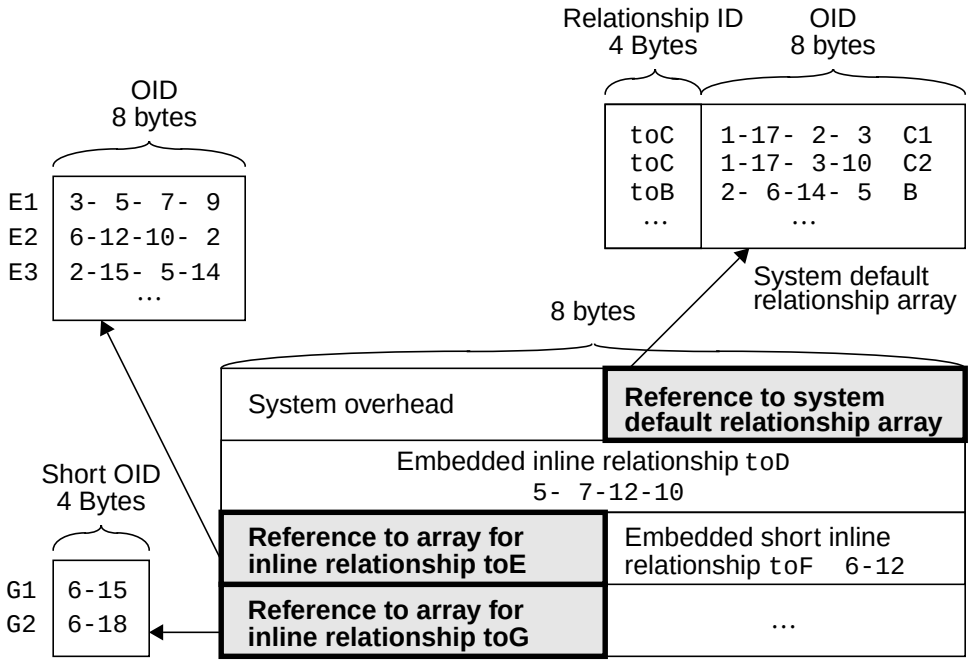
An *inline to-many* relationship requires the following additional space:

- 4 bytes per relationship, for the reference to the relationship array, whether there are related objects or not.
- 14 bytes per relationship, for the relationship array, if there are any related objects.
- 8 bytes per related object for a long reference.
- 4 bytes per related object for a short reference.

Since objects are stored on eight-byte boundaries, you should round up your size calculations to the nearest eight bytes.

The following figure illustrates an object with:

- A non-inline one-to-one relationship `toB`.
- A non-inline one-to-many relationship `toC` with objects `C1` and `C2`.
- An inline one-to-one relationship `toD`.
- An inline one-to-many relationship `toE` with objects `E1`, `E2` and `E3`.
- A short inline one-to-one relationship `toF`.
- A short inline one-to-many relationship `toG` with objects `G1` and `G2`.



Choosing Between Non-Inline and Inline Storage

Choosing between non-inline and inline relationships depends on how many related objects your application requires. Non-inline relationships use very little space for small numbers of related objects, because the overhead of only one extra array is required. However, there is an implied limit on the number of related objects because the entire array must fit into available swap space when the object is fetched. Also, traversing a non-inline relationship, particularly as the number of related objects gets large, is not very efficient.

Inline relationships have a higher space overhead. To-one inline relationships are embedded within objects, so they take up space even when they are not used. However, traversing inline relationships is very efficient. An object in a one-to-one inline relationship can be retrieved quickly because it is embedded in the object. An object in a one-to-many inline relationship can also be retrieved quickly because the application needs to traverse only that relationship instead of all of the relationships on the object.

Changing How a Relationship is Stored

You can change how a relationship is stored simply by modifying the class definition containing the relationship. If schema evolution is allowed, the class description in the schema will be evolved to match the Java class definition, and any objects of the class within the database will be converted to the new definition. See Chapter 15, “Schema Evolution and Object Conversion,” for more information about what happens when you change a class definition.

Objectivity for Java supports all permutations of conversions between the different ways of storing relationships. You should note, however, the following behaviors that accompany certain types of conversions:

- In any schema evolution involving a bidirectional relationship, Objectivity for Java evolves both classes at the same time to the same type of storage mode.
- Whenever relationships are converted to *short inline* from any other format, references contained by the converted objects are set to null if the referenced objects are *not* in the same container.

Using Relationships in Objectivity for Java

In order to use relationships in Objectivity for Java, you must first define the relationship in the classes whose objects will be related. Relationships may be defined only on application-defined persistence-capable classes; the related class may be any persistence-capable class.

Defining a relationship on a class merely makes it *possible* to relate instances of that class to objects of the related class. As new instances are created dynamically, actual relationships between those instances must also be dynamically created and deleted. Relationships are created and deleted explicitly by the application or implicitly by Objectivity / DB when objects are copied or versioned.

Defining Relationships

You must do two things to define a relationship:

- Declare a non-static *relationship field*. Objectivity for Java initializes this field with an object that will support the relationship services at runtime. This relationship field is typically declared `private` and it manages the runtime operations of the relationships.
- Define a public static *relationship definition method* that returns an instance of a subclass of `Relationship` that corresponds to the cardinality of the relationship. This method is used by Objectivity for Java to define the relationship; you never need to call it explicitly in your application.

Relationship Field

The relationship field must be of one of the following types:

- `ToOneRelationship`—if objects of the class can be related to only one other object.
- `ToManyRelationship`—if objects of the class can be related to many objects.

EXAMPLE This definition illustrates how the salesperson-contacts example discussed earlier would be defined in Objectivity for Java. It shows a class named `Salesperson` whose instances can be related to many objects. The relationship is implemented by an instance of `ToManyRelationship` identified by the name `contacts`. Note that as with other fields of objects, the recommended practice is to make the relationship field private, and provide public methods for exposing the relationship behavior.

```
class Salesperson {
   ...
   private ToManyRelationship contacts;
   ...
}
```

Relationship Definition Method

The relationship definition method must return a properly initialized instance of a subclass of `Relationship` (`OneToOne`, `OneToMany`, `ManyToOne`, or `ManyToMany`) that corresponds to the cardinality of the relationship. The name of the method must be formed by appending `_Relationship` to the name of the relationship field.

Each of the `Relationship` subclasses provides several constructors for creating the relationship instance. The parameters specify whether the relationship is

unidirectional or bidirectional, copy and version behavior, operation propagation options, and storage mode.

The first two parameters of all constructors are the name of the local relationship field and the name of the related class. If the relationship is bidirectional, the third parameter must be the name of the *inverse relationship field* in the related class. The next four parameters specify how objects are affected when the object to which they are related is locked, deleted, copied, or versioned. You must specify either all of the behavior specifiers or none of them. For example, if you want to specify lock propagation, you must also specify delete propagation, copy mode, and version mode. The constants used to specify the various behaviors are defined in the `Relationship` class. Finally, the last parameter specifies the storage mode; storage mode constants are also defined in the `Relationship` class. This parameter can be omitted, in which case the relationship is stored non-inline.

EXAMPLE The relationship between a `Contact` and a `Salesperson` is defined to be bidirectional, thus requiring relationship definitions in both the `Salesperson` and `Contact` classes. The following definition also implements our earlier decision to disable the propagation of locking and deletion operations. If any `Salesperson` or `Contact` objects were copied or versioned, relationships with any respective `Contact` or `Salesperson` objects would be moved to a new `Salesperson` object and copied to a new `Contact` object. Both relationships would be stored non-inline.

```
package Sales;
import com.objy.db.app*;

class Salesperson extends ooObj {
   private ToManyRelationship contacts;
   public static OneToMany contacts_Relationship() {
   return new OneToMany(
      "contacts",                // Relationship field
      "Sales.Contact",           // Related class
      "salesperson",             // Inverse
      Relationship.COPY_MOVE,    // Copy behavior
      Relationship.VERSION_MOVE, // Version behavior
      false,                     // Delete propagation
      false,                     // Lock propagation
      Relationship.INLINE_NONE); // Store non-inline
   }
}
```

```
package Sales;
import com.objy.db.app*;

class Contact extends ooObj {
   private ToOneRelationship salesperson;
   public static ManyToOne salesperson_Relationship() {
   return new ManyToOne(
      "salesperson"              // Relationship field
      "Sales.Salesperson",       // Related class
      "contacts",                // Inverse
      Relationship.COPY_COPY,    // Copy behavior
      Relationship.VERSION_COPY, // Version behavior
      false,                     // Delete propagation
      false,                     // Lock propagation
      Relationship.INLINE_NONE); // Store non-inline
   }
}
```

Accessing Relationships

To create and delete relationships between objects and navigate between related objects, you use the methods defined by the class, `ToOneRelationship` or `ToManyRelationship`, of the object referenced by the relationship field.

Before you can call any of these methods, however, the object containing the relationship field must be persistent and the session that owns it must be in a transaction. While the object is transient, the relationship field is uninitialized and any attempt at accessing it will throw a `NullPointerException`. On the other hand, a transient object can be passed as a parameter to one of the methods that form a relationship, and it will be made persistent as part of the operation. In that case, the default clustering strategy clusters the related object into the container where the relating object is stored. Once the related object has been made persistent, it belongs to the same session as the relating object.

Once you have defined a relationship on a class and created a persistent instance of that class, you can:

- Create a relationship between the instance and a given object of the related class.
  - `ToOneRelationship.form`
  - `ToManyRelationship.add`

- Delete a relationship between the instance and a given related object.
  - `ToOneRelationship.drop`
  - `ToManyRelationship.remove`
  - `ToOneRelationship.clear`
  - `ToManyRelationship.clear`
- Test whether the instance has any related objects.
  - `ToOneRelationship.exists`
  - `ToManyRelationship.exists`
- Retrieve related objects.
  - `ToOneRelationship.get`
  - `ToManyRelationship.scan`
- Test whether a given object is related to the instance.
  - `ToOneRelationship.includes`
  - `ToManyRelationship.includes`

To ensure that relationships are used correctly, you should define relationship access methods for every relationship of a class and call those methods whenever you get related objects or establish relationships with other objects.

- The relationship in an object's relationship field is not read from the database until you fetch the object's data. You should call the object's `fetch` method before accessing its relationship.
- The methods of a relationship object obtain the locks they need; this means that the methods of a relationship object will throw a `LockNotGrantedException` if the session is unable to obtain whatever lock is needed for the particular operation.
- The methods of a relationship object ensure that modifications to the relationship are written to the database; your relationship access methods do not need to call `markModified`.
- Retrieved objects must be cast to the appropriate class. Your relationship access methods can perform the necessary cast and return related objects of the correct class.

EXAMPLE The `Contact` class provides the typed `setSalesperson` method to create a relationship between a `Contact` object and a `Salesperson` object. A `Contact` object can get its related `Salesperson` object with the public `getSalesperson` method, which casts the object returned from the `ToOneRelationship.get` method to the correct type. The complete class definition appears in the `Sales.Contact` programming example (see page 438).

```
package Sales;
class Contact extends ooObj {
   …

   public void setSalesperson(
          Salesperson newSalesperson) {
      fetch();
      // Remove any existing relationship
      this.salesperson.clear();
      this.salesperson.form(newSalesperson);
   }

   public Salesperson getSalesperson() {
      fetch();
      // Cast retrieved object to class Salesperson
      return (Salesperson)this.salesperson.get();
   }
}
```

The `findContactsByCompany` method of the `Salesperson` class illustrates how to scan a to-many relationship for the objects that satisfy a predicate condition. The complete class definition appears in the `Sales.Salesperson` programming example (see page 435).

```
package Sales;
class Salesperson extends ooObj {
   …
   public Iterator findContactsByCompany (
          String company) {
      fetch();
      String predicate = new
            String("company == \"\"" + company + "\"");
      return this.contacts.scan(predicate);
   }
}
```

When an object is first created or when an object is first read from the database, all its relationship fields have `null` values. To fill in the relationship fields, `fetch()` must be called. Once fetched, the fields, which hold objects typed either as `ToOneRelationship` or `ToManyRelationship`, exist for the lifetime of the containing object. Developers must never set the relationship fields or assign their values to other objects. This is why the fields are shown as private in all the examples.

Throughout the lifetime of a persistent object, the relationship fields get set to different values at runtime by Objectivity for Java. This can occur at these events:

- The object defining the relations was made persistent.
- The object defining the relations was read back from the database for the first time. In other words, the object did not have a local representation in the session.
- The object is made dead. Its relationship fields are nulled out.
- The object was recently made persistent within a transaction and that transaction is aborted. The relationship fields are nulled out.

Objectivity for Java obtains read and write locks as required when using the relationship objects. The following table lists the operations and the lock types for each type of relationship.

Relationship Type	Automatically Attempts Read Lock Before Operations	Automatically Attempts Write Lock Before Operations
To-one relationship	`clear(),drop(),form()`	`exists(), includes(), get()`
To-many relationship	`clear(), remove(), add()`	`exists(), includes(), scan()`

Relationships are maintained in the database and not in Java memory. This alleviates garbage collection issues.

8

Persistent Objects

A *persistent* object is an object that has been assigned a storage location in a federated database. When you commit the transaction in which you create a persistent object, that object's data is saved in the database; the object can then be accessed by other processes. A persistent object continues to exist beyond the duration of the process that creates it. In contrast, a *transient* object exists only within the memory of the process that creates it; when that process terminates, the transient object ceases to exist.

In This Chapter

Making an Object Persistent

Only instances of persistence-capable classes can be persistent objects. Each application defines its own persistence-capable classes. In addition, Objectivity for Java includes persistence-capable classes for collections of persistent objects.

A persistent object can be:

- Given a name to facilitate retrieving the object from the database.
- Related to another persistent object.
- Added to a persistent collection.

If you perform any of these operations on a transient object, that object is made persistent. If you attempt to perform one of these operations on an object whose class is not persistence-capable, a `NonPersistentClassException` is thrown.

Container classes are themselves persistence-capable. A container is both a storage object and a persistent object; you can name a container, reference a container in a persistent field of another persistent object, add a container to a persistent collection, and establish a relationship from a persistent object to a container. You can define your own container classes if you need to create containers that have persistent data or relationships; however, most applications have no need to define their own container classes.

When you use the `new` operator to create an object of a persistence-capable class, the newly created object is transient. On the other hand, when you create an object by copying (using the persistent operation `copy`, *not* the Java `clone` method), an existing persistent object, the new copy is made persistent automatically. This section explains how to make a transient object persistent; “Copying a Persistent Object” on page 173 explains how to copy an object, creating a new object that is automatically persistent.

You can make a transient object persistent only while a session is in a transaction; the newly persistent object belongs to that session. When you make an object persistent, the corresponding persistent object is created in the federated database. When you commit or checkpoint the transaction, the object is made visible to other clients. If you abort the transaction, the object in memory reverts to being transient and the new persistent object is removed from the federated database.

Immediate and Delayed Persistence

There are many ways to cause a transient object to be made persistent. Some actions make an object persistent immediately; others cause it to be made persistent when you commit or checkpoint the transaction.

Immediate Persistence

Any of the following actions will make a transient object become persistent immediately:

- Explicitly clustering a transient object with a given basic object, container, or database.
- Explicitly adding a transient container to a database.
- Making a transient object a named root in a particular database or federated database.
- Naming a transient object in the scope of a persistent object.
- Forming a relationship from a persistent object to a transient object.
- Adding a transient object to a persistent collection.

Delayed Persistence

You can also make a transient object persistent by referencing it in a persistent field of a persistent object. The referenced object is not made persistent immediately: it only becomes persistent when the referencing object is written to a database. The referencing object is written to a database when you commit or checkpoint the transaction in which you made the object persistent or modified the object's persistent fields. Thus a transient object referenced by a persistent object is made persistent when you commit or checkpoint the transaction in which you created the reference between the persistent and transient object. This process is repeated recursively; if the referenced object references another transient object, that object is also made persistent, and so on. This process is called persistence by reachability. The operations `fd.flush`, `db.flush`, `cont.flush`, and `obj.write` will also write out any modified objects graph, hence making transient objects persistent.

Assignment of Storage Location

Making an object persistent assigns it to a storage location in the federated database. Making a basic object persistent assigns it a location in a particular container; making a container persistent assigns it a location in a particular database.

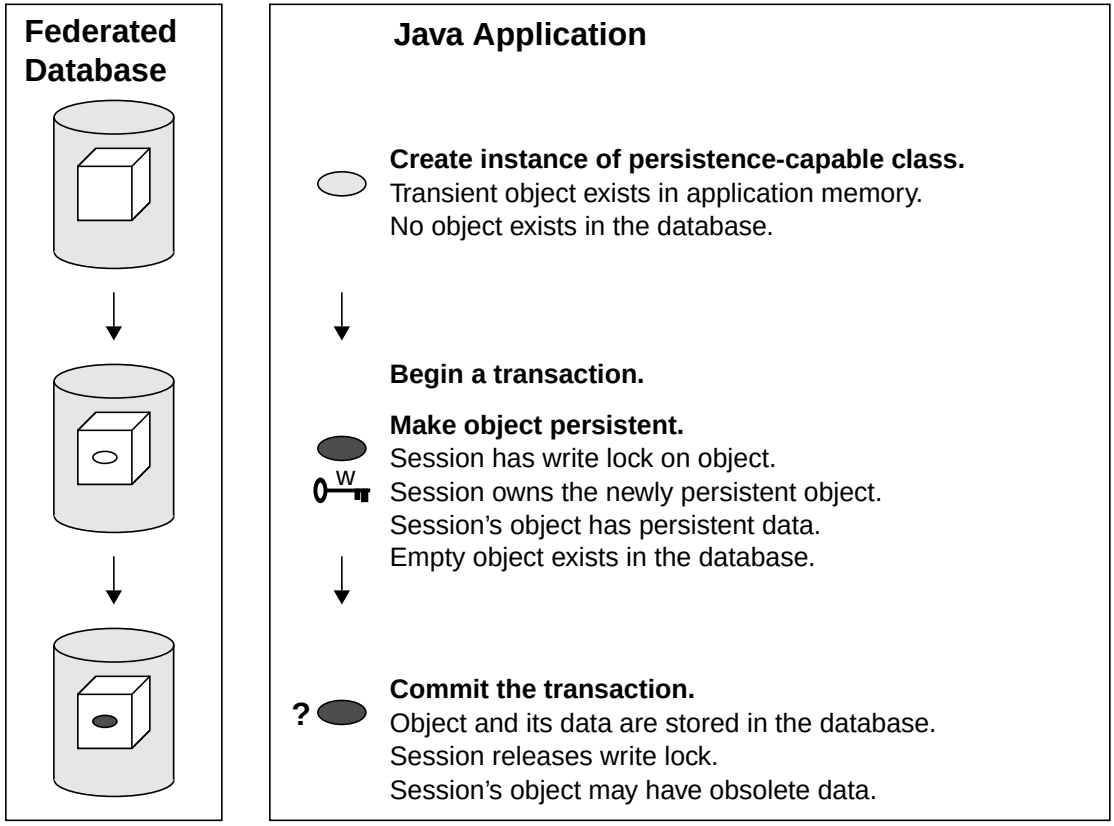
The assignment of a basic object to a container, or a container to a database, can be either explicit or implicit.

- When you make a transient object persistent by calling the `cluster` method of a database, persistent container, or persistent basic object, you explicitly assign the object being clustered a location near (or within) the clustering object.
- When you make a transient container persistent by calling the `addContainer` method of a database, you explicitly assign the transient container to that database.
- If the object becomes persistent by any other means, it is assigned to a storage location by its session's clustering strategy. See "Clustering Objects" on page 237.

For guidelines on deciding where to store your persistent objects, see "Assigning Objects to Databases" on page 91 and "Assigning Basic Objects to Containers" on page 99.

Storing an Object Persistently

The following figure illustrates what happens to a basic object's persistent data during the transaction that makes it persistent. A transient object can be created either during a transaction or outside a transaction; in this example the object is created before the start of the transaction that makes it persistent. When an object becomes persistent, the database is informed which container the new object has been assigned to. The database locks that container for read / write access by the session that owns the newly persistent object. Note, however, that the object's persistent data (that is, the values in its persistent fields) is not written to the database. You can think of the database as having an empty object in the specified container.



Key to Symbols

= Database
= Container
W = Write lock

= Transient object
= Persistent object with persistent data
= Persistent object without persistent data
? = Persistent object whose data may be obsolete

A new persistent object is first written to the database when you commit or checkpoint the transaction. If the transaction is aborted before the object is written to the database, the corresponding empty object is removed from the database and the object in memory becomes transient again.

Committing a transaction writes the object's persistent data to the database and releases all the session's locks. The object in memory continues to be the session's local representation of the persistent object. The persistent object itself resides in the database and can be shared by other clients, including other sessions of the

same application and other applications written in Java, C++, or Smalltalk. Each client has its own local representation of the persistent object.

After the transaction is committed, its session's local representation of a persistent object is not guaranteed to have the same persistent data as the corresponding object stored in the database. A different client may modify the object while the original session is outside a transaction, making the original session's local representation of the object obsolete.

NOTE To ensure that you have the most current persistent data for an object, you must start a transaction, then fetch the object's data. See "Fetching an Object's Data" on page 170.

Working With a Persistent Object

A session can work with a persistent object once all the following conditions have been satisfied:

- The session is in a transaction.
- The session has a local representation of the persistent object. This condition is satisfied if you make the object persistent or retrieve it from the database while the session is in a transaction.
- The session has a lock on the object for the desired access. This condition is usually satisfied automatically. For example, the method that fetches the object's data also locks it.
- The session's local representation of the object has the object's most current persistent data. You must fetch the object's data during the transaction to obtain the most current data from the database.

  If a particular operation does not require access to the object's data (the actual values of its field members), you need not fetch the data. In Objectivity for Java, no persistent operations require doing a fetch. For example, deleting, binding, looking up scope names, linking, and moving do not require a fetch prior to execution.

  You need to fetch data of objects of *application-defined* classes only. Remember that an object's persistent data is the values in its persistent fields, and persistent fields are always defined by the application. A class that is part of the Objectivity for Java interface (such as `ooMap` or `ooContObj`) has no persistent fields, so objects of those classes do not have persistent data and hence never require `fetch()` or `markModified()`.

Once these four conditions are met, you can perform any operation that is consistent with the session's lock on the object. For example, if the object is locked for read, you can look at its persistent data or copy it. If the object is locked for write, you can modify, move, or delete it. Any deletions or changes to an object's persistent data exist only in the application's memory until you commit or checkpoint the transaction; then the changes are written to the database.

When you commit or abort a transaction, the session releases all locks obtained during the transaction. The local representation is still owned by the session, and can be reused in subsequent transactions. Note, however, that if another client has deleted the object while your application was between transactions, an exception is thrown the next time you try to perform an operation on the object, pass the object as an argument to other persistent operations, or try to write it out either directly or from a retrieve from another persistent object.

When a transaction ends, all persistent objects that were made persistent or whose data was fetched during the transaction are marked as needing to have their data fetched. As long as the session is outside a transaction, those objects are not guaranteed to have persistent data consistent with the database, because a different client may have modified them. To ensure that you have access to the most recent version of an object's persistent data, you must start a transaction and fetch the object's data again.

If your application uses multiple session objects, remember that each persistent object, storage object, and ODMG transaction object in the application's memory *belongs to* a particular session. A newly created persistent object belongs to the session that was in a transaction when it was made persistent. A previously existing persistent object belongs to the session that was in a transaction when the object was retrieved from the database.

Objectivity for Java does not allow a persistent object that belongs to one session to interact with objects that belong to another session. For example, if `Session1` owns a particular database and `Session2` owns a particular basic object, you may not make that basic object a named root in that database. Instead, you must retrieve the same basic object while `Session2` is in a transaction. You can then make the newly retrieved basic object a named root in the database because both the newly retrieved object and the database belong to `Session2`. For additional information, see "Object Isolation" on page 51.

Retrieving an Object From the Database

When you retrieve a persistent object from a database, Objectivity for Java creates an object in the application's memory that represents the object stored in the database. The newly created object has no persistent data; you must explicitly fetch the object's data from the federated database to the local representation. Once you have done so, you can access the data in the object's persistent fields just as you would access the fields of any Java object.

Although the object in memory is called a "persistent object," it is important to remember that the persistent object actually resides in the database. The object in memory is just the session's local representation of the persistent object.

Several methods allow you to retrieve objects from the database. Some of these methods retrieve an individual object, for example, by looking up its root name or its name in the scope of a particular scope object. Other methods return an iterator that finds a number of similar objects, for example, all basic objects of a given class that are stored in a given container. For more information about these methods, see Chapter 11, "Retrieving Persistent Objects".

Every session maintains a *cache* of the persistent objects that belong to it (including internal persistent objects). The cache ensures that the session has a single object in Java memory representing any particular persistent object that is accessed while the session is in a transaction. When the session is in a transaction, if two different method calls look up the same persistent object, the first method call retrieves the object from the database and returns a local representation for it; the second call returns the local representation created by the first call. For additional information, see "Object Identity" on page 52.

A particular object remains in the cache until any of the following actions occur:

- The application drops its last reference to the object and the Java garbage collector removes the object from the cache.
- The application calls the session's `dropCachedObject` method to drop that object from the cache.
- The application calls the connection object's `dropClass` method to drop all cached information about the object's class. In the process of dropping information about the specified class, this method drops all objects of that class from the cache of every session.
- The application calls the session's `terminate` method, which deletes its entire cache.

When multiple processes run concurrently and access the same objects, there is a possibility that the cached information will become stale if you hold a reference for an object after committing or aborting the transaction in which it was retrieved. See "Avoiding Stale Cache Information" on page 181.

Locking a Persistent Object

Locking an object informs Objectivity/DB how you plan to use that object. The lock you obtain while one session is in a transaction prevents other sessions and other applications from taking actions that would interfere with your intended use of the object.

A session can lock a persistent object either for read-only access or for read/write access.

- A *read lock* gives a session read-only access to the object. In a non-MROW transaction, the session can obtain a read lock on an object if no other session has a write lock on the object. In an MROW transaction, the session can obtain a read lock even if another session has a write lock on the object.
- A *write lock* gives a session read/write access to the object. The session can obtain a write lock if no other session has a write lock on the object and no session in an non-MROW transaction has a read lock on the object.

Containers are the fundamental unit of locking; when a session locks a basic object, the session implicitly obtains a lock on the object's container. When a session locks a container, the session implicitly obtains locks on all basic objects in the container.

See Chapter 4, "Locking and Concurrency," for a description of MROW and non-MROW transactions and for further details about locking.

Obtaining a Lock

You usually don't need to make an explicit call to lock a persistent object. Built-in methods that require a persistent object to be locked will try to obtain the necessary lock. For example, you cannot fetch an object's persistent data unless the session has a read lock on the object; the `fetch` method obtains the necessary lock before it fetches the data.

If a method is unable to obtain a lock, it will throw a `LockNotGrantedException`. You can reserve locks in advance of when you need them with the `lock` method.

If an object's class has field access methods to ensure that its persistent fields are used safely, the methods to get and set the value of a persistent field will automatically lock the object for you. See "Field Access Methods" on page 138.

Once your session obtains a lock, it typically retains the lock until the transaction is committed or aborted.

Modifying a Lock

After you obtain a lock, you can modify it in the following ways during the transaction:

- You can *upgrade* a read lock to a write lock by calling the `lock` method of the locked object; the lock is propagated along relationships for which lock propagation is enabled, thus locking the related objects for read / write access. If you call an object's `lockNoProp` method instead of `lock`, the lock is not propagated to related objects.
- You cannot *downgrade* an object's write lock to a read lock independent of other locks. However, if you checkpoint a transaction, you can specify that all write locks obtained during the transaction should be downgraded to read locks.
- You can *release* the read lock on a container (and all its objects) by calling the container's `releaseReadLock` method. If the container is not locked, the method does nothing. If the container is write locked either by the session or some other object, an exception is thrown.

WARNING Objectivity / DB does not prevent you from changing the persistent fields of an object you have locked for read-only access. It is your responsibility to access an object consistently with the lock you have obtained for it.

Fetching an Object's Data

Fetching an object's data locks the object, if necessary, and transfers the object's persistent data from the federated database to the persistent fields of the local representation. Once you have fetched an object's data, you can access the data in the object's persistent fields just as you would access the fields of any Java object. If a persistent field references a persistent object that has not already been retrieved, fetching the field's data retrieves the referenced object but does not fetch that object's data; thus, the field is set to reference an empty object. A referenced object's data must be fetched independently.

Once you fetch an object's data, it typically remains locked with its data up to date until the transaction is either committed or aborted.

You must call an object's `fetch` method to fetch its data; Objectivity / DB never automatically fetches persistent data. However, if the object's class has field access methods to ensure that its persistent fields are used safely, the method to get the value of a persistent field will fetch the object's data; see "Field Access Methods" on page 138. There is no performance overhead for making multiple calls to an object's `fetch` method; if the data in memory is up to date, the method does nothing.

NOTE Objectivity / DB does not prevent you from accessing the persistent fields of an object that is marked as needing to have its data fetched; however, the data in those fields is not guaranteed to be consistent with the database.

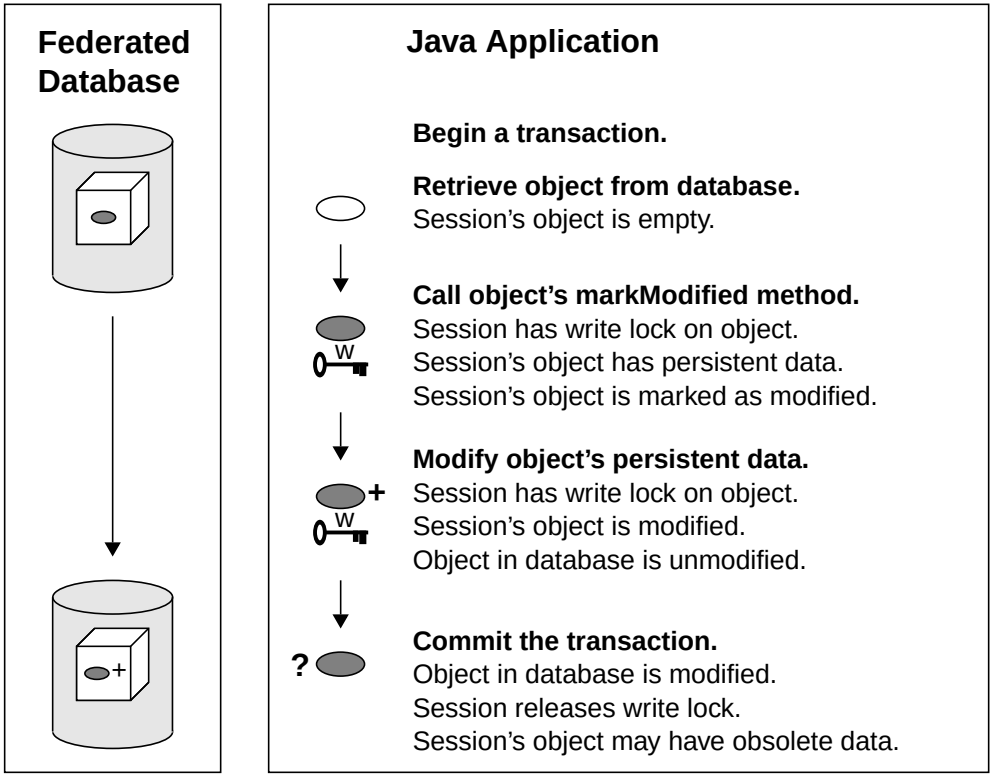
Modifying a Persistent Object

Before you change the persistent data of a persistent object, you should obtain a write lock on the object and fetch its data. To do so, you should call the object's `markModified` method; in addition to locking the object's container and fetching its data, this method marks the object as modified. When you commit or checkpoint the transaction, any persistent objects that have been marked as modified will be written to the database; your changes to the object are then available to other processes.

If the object's class has field access methods to ensure that its persistent fields are used safely, the method to set the value of a persistent field will call `markModified` for you. See "Field Access Methods" on page 138.

WARNING If an object is an element in a sorted set or the key of an element in a sorted object map, you must remove the element from the collection before making any change to the object that would affect how the element is sorted. Similarly, if an object is an element in an unordered set or the key of an element in an unordered object map, you must remove the element from the collection before making any change to the object that would affect how the element's value is computed. See Chapter 9, "Persistent Collections".

The following figure illustrates what happens to a basic object's data during a transaction that modifies it. Any changes to an object's persistent data exist only in the application's memory until you commit or checkpoint the transaction; then the changes are written to the database.



Key to Symbols

= Database
= Container
W = Write lock

= Persistent object with persistent data
= Persistent object without persistent data
+ = Persistent object with modified persistent data
? = Persistent object whose data may be obsolete

Copying a Persistent Object

To create a new persistent object with the same persistent data as an existing persistent object, call the copy method of the existing object. You can copy basic objects only, not containers.

If the object you are copying is marked modified, it is written to the federated database (but is not committed) before the copy operation begins. The operation is performed in the federated database, not in memory. That is, a new persistent object is created in the federated database; the copy method specifies the database, container, or basic object with which to cluster the new object. Persistent data is copied from the existing object in the federated database to the new object in the federated database; see "Copying the Object's Fields" on page 173. Relationships of the existing object are copied or not, as specified by the definition of each relationship; see "Copying the Object's Relationships" on page 174. The copy method returns a local representation of the new object; the returned object is persistent, locked for write, and empty. You must fetch its persistent data if you want to examine or modify that data.

When you commit or checkpoint the transaction in which you copy an object, the new copy is made visible to other clients. If you abort the transaction, the new copy is removed from the federated database and the new copy in memory becomes a dead object.

NOTE The Objectivity for Java copy operation is not the same as the Java clone operation. The following sections describe the copy operation in more detail.

Copying the Object's Fields

The copy method copies the persistent object stored in the database rather than the session's local representation of that object. In fact, the local representation of the original object does not need to have its persistent data in memory. Because the database contains only persistent data, any transient fields are not affected by the copy operation.

The copy method creates a shallow copy of the original persistent object. Each field of a *shallow copy* contains exactly the same value as the corresponding field of the original object. If the original object contains a reference to an array or an object in a persistent field, the shallow copy contains a reference to the same array or object in its corresponding field. In contrast, each field of a *deep copy* contains a copy of the value in the corresponding field of the original object. If the original object contains a reference to an array or an object in a persistent field, the deep copy contains a reference to a deep copy of the array or object in its corresponding field.

Your application may need to implement a mechanism for copying a persistent object with its transient data or for creating a deep copy of a persistent object. If so, you should implement your customized copy mechanism as follows:

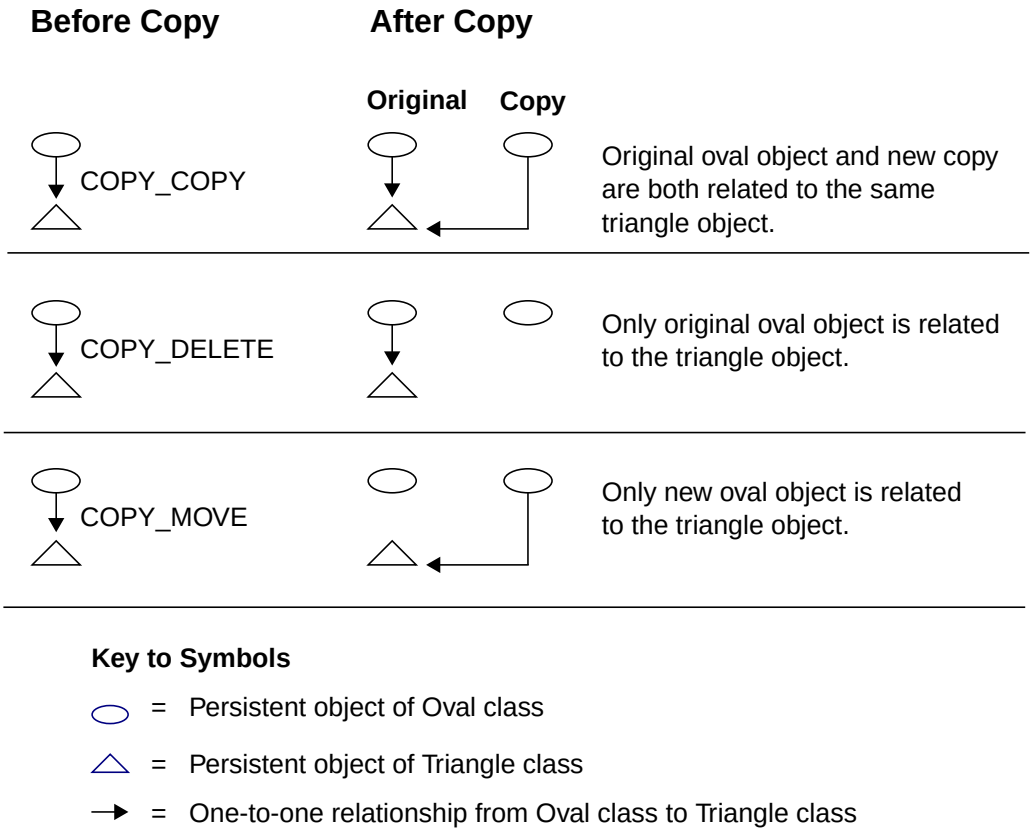
1. Call the `copy` method of the original persistent object to create a new persistent object that is a shallow copy of the original object.
2. Fetch the new object's persistent data and modify its fields as appropriate for your application. For example, copy the values of the transient fields and / or replace the object references in persistent fields with references to deep copies of the currently referenced objects.
3. If your copy mechanism modifies any persistent fields of the new copy, it must call the new copy's `markModified` method to ensure that the changes are written to the database when you commit or checkpoint the transaction. (If the object's class has field access methods to ensure that its persistent fields are used safely, the method to set the value of a persistent field will mark the object as modified for you; see "Relationship Access Methods" on page 143.)

Copying the Object's Relationships

If the object being copied has relationships to other objects, the definition of each relationship specifies how the relationship is handled. When a relationship is created, its copy behavior is specified with one of the following constants (defined in the `Relationship` class):

- `COPY_COPY`—When an object with this relationship is copied, the relationship is copied. Both the original object and the new copy are related to the same object(s).
- `COPY_DELETE`—When an object with this relationship is copied, the original object retains the relationship; the new copy does not have a relationship.
- `COPY_MOVE`—When an object with this relationship is copied, the relationship is moved from the original object to the new copy.

The following figure illustrates the result of copying an object with a relationship using each of the three copy modes.


Before Copy
After Copy
Original
Copy
COPY_COPY
Original oval object and new copy are both related to the same triangle object.
COPY_DELETE
Only original oval object is related to the triangle object.
COPY_MOVE
Only new oval object is related to the triangle object.
Key to Symbols
= Persistent object of Oval class
= Persistent object of Triangle class
= One-to-one relationship from Oval class to Triangle class

Moving a Persistent Object

At some point you may decide that you need to reallocate objects to a different container configuration. For example, you may want to increase the number of containers to achieve greater concurrency. Objectivity for Java supports moving a persistent basic object to a new container with the `move` method.

The `move` method requires that you specify the database, container, or basic object that you want the moved object to be *near*. The following table summarizes where a basic object is stored for any given *near* object.

Near Object	Where the Moved Object is Stored
Database	Default container of the database.
Container	That container.
Container currently containing the basic object	That container; the basic object *may* be moved to another location within the container.
Basic object	Container in which that basic object is stored; on the same page as the specified object, if possible, or on a page close to that object.

After a successful move, the object has a new object identifier, and the local representation of the object references the new basic object. If the move is unsuccessful, the identifier is unchanged and the local representation remains valid and still references the original object.

NOTE Moving an object moves that object alone, not any objects referenced by the object's fields. If you want to move a graph of objects, you must traverse the graph and explicitly move each object. In addition, you may explicitly move any referenced internal persistent objects.

If you abort the session after moving an object:

- The object does not get moved. The local representation of the object remains valid if you have not deleted the object, or the container or database the object is moved from, or moved to.
- If you have deleted the "from" container during the session, the local representation of the object becomes dead.
- If you have deleted the "to" container during the session, the local representation of the object becomes dead.
- If you have deleted the object during the session, the local representation of the object remains dead.

- After you move a newly persistent object (made persistent in the session aborted), the object will revert to transient.

WARNING Since moving an object changes its object identifier, all Objectivity / DB-maintained references to the object containing the old identifier become invalid and any attempt to access such a reference will cause an `ObjectNotFoundException`. Furthermore, the old identifier may eventually be reassigned to a new persistent object by Objectivity / DB. Therefore, when you move an object, you should, within the same transaction, update references to the object within all relevant relationships, persistent collections (including root named maps), scopes, and indexes. Failure to do so may result in inconsistent references and database corruption.

Object Linked by Relationship

If the moved object is referenced from a unidirectional relationship of another object, you must drop a to-one relationship or remove a to-many relationship before the move and then form a to-one relationship or add a to-many relationship after the move. For information on how to work with relationships, see "Accessing Relationships" on page 157. On the other hand, if the moved object has bidirectional relationships with other objects, Objectivity / DB updates any references to the object.

Object in a Persistent Collection

You should not move an object that is used in a persistent collection:

- As an element of a list, unordered set, or sorted set
- As a key in an element of an unordered object map or a sorted object map
- As a value in an element of a name map, an unordered object map, or a sorted object map

If you need to move such an object, you must first temporarily remove the appropriate element(s) from all affected persistent collection(s). After the move is successfully completed, you can add the removed element(s) back to the collection(s).

WARNING If an object in a persistent collection is first moved and then deleted, data corruption or data loss to the collection's objects could result. To avoid data corruption, you must remove an object from a collection before moving the object.

Object Used as a Named Root

Objectivity for Java uses name maps to store root names. As a consequence, the recommendations and warnings concerning moving objects referenced in a persistent collection apply also to named roots. To prevent possible corruption of a root dictionary and any objects it references, you must unbind a root name before you move the object, and bind the object after the move. See “Named Roots” on page 212 for information on how to work with named roots.

Scope Object or Scope-Named Object

An object that is a scope object or is scoped by other objects will lose that scope when the object is moved. This will not compromise data integrity in any way. However, any moved object will need to reestablish scope-to or scope-by any objects after the move.

To preserve the moved object as a name scope, you must:

1. Before the object is moved, call `scopedObjects` to determine all the scope-named objects.
2. Before the object is moved, call `lookupObjName` to determine the scope name of each scope-named object.
3. After the object is moved, reestablish the scope name of each scope-named object in the scope of the moved object with the `nameObj` method.

To preserve all the scope names of a moved object you must:

1. Before the object is moved, determine the scope objects of the object by calling the `scopedBy` method.
2. Before the object is moved, determine the scope name of the object in each scope, by calling the scope object’s `lookupObjName` method.
3. After the object is moved, reestablish the scope name of the moved object in each scope by calling the scope object’s `nameObj` method.

See “Name Scopes” on page 214 for further information on how to work with scope named objects.

Indexed Object

A predicate scan using an index that references a moved object will yield undefined results. In contrast to relationships, name maps, and scopes, it is not possible to delete a reference to an individual object from an index. Therefore, you must delete the entire index before moving an object that is referenced by the index and recreate the index after the object is moved. See “Working With an Index” on page 257 for information on how to delete and recreate an index.

Deleting a Persistent Object

Deleting a persistent object removes the object from the database when the transaction commits. Deleting a persistent object deletes any association links from the deleted object to destination objects. Furthermore, if any of the associations is bidirectional, the inverse link to the deleted object is removed from each destination object to maintain referential integrity. However, if another persistent object references the deleted object through a unidirectional association or directly in one of its attribute data members, you are responsible for removing that reference.

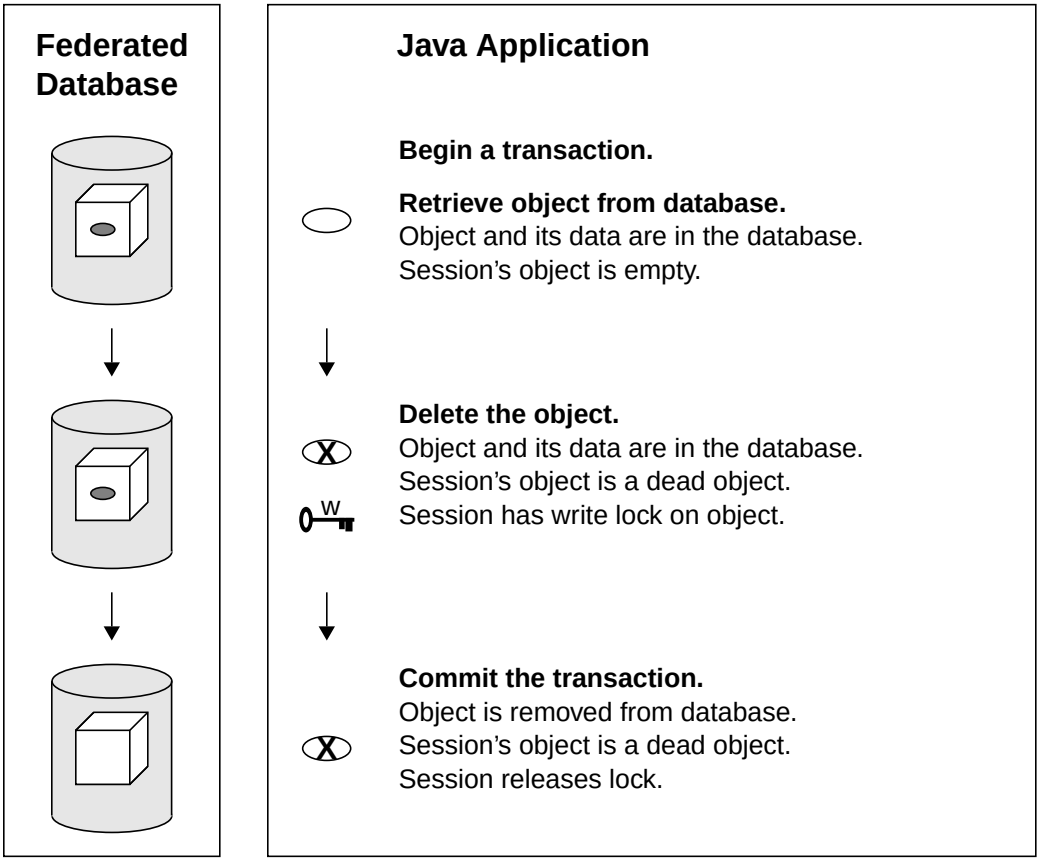
To delete a basic object, retrieve the object from the database and call its `delete` or `deleteNoProp` method. To delete a container and all its objects, get a local representation of the container, and call its `delete` or `deleteNoProp` method.

The `delete` methods propagate the deletion operation along relationships for which deletion propagation is enabled.

When you use the `delete` method to delete a persistent object that has relationships for which delete propagation is enabled, you also delete the destination objects linked by those relationships. If you want to delete a source object without deleting its destination objects, you should call its `deleteNoProp` method instead.

The session's local representations of deleted objects are converted to dead objects. When you commit or checkpoint the transaction, the objects are physically removed from the federated database. If the transaction is aborted, the objects are not removed from the federated database.

The following figure illustrates a transaction in which a basic object is deleted.



Key to Symbols

= Database
= Container
W = Write lock
= Persistent object with persistent data
= Persistent object without persistent data
= Dead object

Deleting an Object with References

Deleting an object deletes that object alone, not any objects referenced by the object's fields. If the object is stored in a garbage-collectible container, its referenced objects will become available for garbage collection if no other object references them. If you store a graph of related objects in a non-garbage-collectible container, however, you must take explicit action if you want to delete the entire graph of objects. You must traverse the graph and explicitly delete each object. In

addition, you should explicitly delete any referenced internal persistent objects. "Container Types" on page 96 explains the difference between garbage-collectible and non-garbage-collectible containers.

Deleting an Object with Relationships

A deleted object is removed from any bidirectional relationships in which it is involved. However, if another persistent object references the deleted object in a unidirectional relationship or directly in one of its persistent fields, you are responsible for removing that reference. An exception is thrown if you attempt to write a persistent object that references a dead object.

Deleting an Object in a Persistent Collection

You should not delete an object used in a persistent collection:

- As an element of a list, unordered set, or sorted set
- As a key in an element of an unordered object map or sorted object map
- As a value in an element of an unordered object map or sorted object map

If you need to delete such an object, you must first remove the appropriate element(s) from all affected persistent collections.

Avoiding Stale Cache Information

Each session maintains a cache of the objects that belong to it. You typically do not need to be concerned about the state of the information in the cache. In a concurrent environment, however, multiple processes may run simultaneously, each creating, moving, and deleting objects of the same class. Situations can arise in which the actions of one process render the cache in another process "stale."

Within the cache, objects are organized by their object identifiers (OIDs). Any method that looks up a persistent object gets the object's OID from the federated database and checks the session's cache for that OID. If the OID is not in the cache, the persistent object is retrieved from the database; otherwise the persistent object is obtained directly from the cache.

NOTE If your application runs in a multiprocessing environment in which other processes may delete or move objects that your application accesses, you should release all references to a persistent object at the end of the transaction in which the object was made persistent or retrieved. Doing so allows the Java garbage collector to remove the object from the cache so that your application will retrieve it anew if it is needed in a different transaction on the same session.

At the end of the transaction in which a persistent object is retrieved from the database, your application gives up its lock on the object, permitting other processes to access the object. If your application holds a reference to the object, the Java garbage collector cannot remove the object's data from the session's cache. If another process then moves or deletes the object, the cached information becomes stale. The cache continues to identify the object by its original OID, which is now invalid:

- When an object is deleted from the federated database, its OID becomes invalid.
- If an object is moved from one container to another, it is given a new OID; its original OID becomes invalid.

The situation is complicated by the fact that Objectivity/DB reuses OIDs that have become invalid. If an invalid OID in the cache is reassigned to a different object, your application will use the OID thinking that it identifies one object when it actually identifies a different object—possibly an object of a different class. For example, if your application performs an operation that obtains the reassigned OID, it will not retrieve the correct object from the database but will instead use the cached object.

If the OID has been reassigned to an object of a different class, the cached type number for the OID will identify the class of the deleted or moved object, whereas the type number for the OID in the database will identify the class of the object to which the OID is now assigned.

Three problems can arise when you access an object with stale cache information:

- The object's OID no longer exists in the federated database.
- The object's OID has been reassigned to an object of the same class.
- The object's OID has been reassigned to an object of a different class.

Unassigned OID

If you try to access an object whose OID is now unassigned, an `ObjyRuntimeException` is thrown, indicating that the object does not exist in the database.

If you detect this situation, you can remove the stale cached information in either of two ways:

- You can release all references to the object and let the Java garbage collector remove its data from the cache. There is a small window of time between when you drop the last reference and when the garbage collector removes the object's information from the cache. It is conceivable, but unlikely, that a problem could occur during that period.
- You can drop the object's cached information explicitly and make it a dead object by calling the `dropCachedObject` method of the object's session.

OID Reassigned to Object of Same Class

If the OID for a moved or deleted object is assigned to a different object of the same class, your application may use the cached object inappropriately instead of retrieving the new object. Although there is no way to detect that an object's OID has been assigned to a different object of the same class, your application may find that the object's data appears inconsistent.

Objectivity / DB protects against database corruption in the case that you make changes to the cached object and try to save it. If you attempt to write the object whose cached information is stale, its data is fetched first, which will bring the correct data into the Java object. If the object was moved or deleted and its OID reassigned, you will write the new object, thinking that it is writing the old (deleted or moved) object.

The same problem arises if an object X has a field that references an object Y and Y's OID is reassigned to an object of the same class; X will still have a valid reference, but to the wrong object.

Because these problems are undetectable, if your application runs in a multiprocessing environment in which applications may be moving or deleting objects, the safest programming practice is to drop all object references at the end of every transaction.

OID Reassigned to Object of Different Class

If the OID for a moved or deleted object is assigned to a different object of a different class, your application may use the cached object inappropriately instead of retrieving the new object. In this case, two exceptions may be thrown:

- A `ClassCastException` is thrown when you look up an object of a particular class, get a cached object of a different class, and cast the returned object to the expected class. For example, this situation can occur when you scan for all objects of a given class and one of those objects has an OID that used to be assigned to a cached object of a different class.

  A `ClassCastException` doesn't always indicate that your application is using a persistent object with stale cache information. This exception is also thrown in other, unrelated, cases in which you try to cast a Java object to an illegal type.

- A `JavaTypeNMismatchException` is thrown whenever you try to fetch an object's data and its cached type number does not match the type number in the database. This exception *always* indicates that you have referenced a persistent object with stale cache information.

  This exception is also thrown if you try to get a referenced or related object whose cached type number does not match the type number in the database.

If you detect that a particular object has stale cached information, you have two alternatives. You can reload the cache, obtaining a reference to the new object that uses the old object's OID, or you can drop the object's cached information.

The decision whether to get a new object reference or drop the cached information depends on the specifics of your application. For example, suppose you perform a scan operation that retrieves an object that does not match its cached information. In that case, you should reload the cache so that you can get the new object that matches the scan criteria. On the other hand, suppose that object X references object Y. If Y has been deleted by another process and it's OID reused by an object of a different class, you should drop the cached information for Y and remove X's reference to the new object.

Reloading the Cache

You can reload the object calling the `reloadCachedObject` method of the object's session. This method removes the old object's cached information, making it a dead object; it then retrieves and returns the new persistent object that uses the old object's OID and caches up-to-date information about the new object.

EXAMPLE This transaction gets all objects from a particular container and tries to cast each one to the `SimpleClassWithField` class. If the cast fails, the cache has stale information about an object of a different class that used to have the same OID. In that case, the code reloads the object and casts it to `SimpleClassWithField`.

```
session.begin();
System.out.println("getting iterator...");
Iterator itr = cont.contains();
SimpleClassWithField scwf = null;
int count = 0;
while (itr.hasNext()) {
   Object o = itr.next(); // Retrieve generic Object
   try {
      scwf = (SimpleClassWithField) o; // Try the cast
   }
   catch (ClassCastException e) {
      // Cast didn't work, reload the object
      scwf =
         (SimpleClassWithField) session.reloadCachedObject(o);
   }
   System.out.println("Got " + scwf.toString());
   count++;
}
System.out.println("count of objects: " + count);
session.commit();
```

Dropping the Object

You can drop the object's cached information and make it a dead object by calling the `dropCachedObject` method of the object's session.

EXAMPLE This transaction scans for all objects of the `SimpleClassWithRef` class. It calls the `getRef` method of each retrieved object to get a referenced object, which should be an instance of `SimpleClass`. If the referenced object is not of the correct type, it drops that object's cached information and calls the referencing object's `setRef` method to remove the reference to the deleted or moved object.

```
session.begin();
System.out.println("getting iterator...");
Iterator itr =
   cont.scan("test.data.SimpleClassWithRef");
Object o = null;
SimpleClassWithRef scwr = null;
SimpleClass sc = null;
boolean needToCheckRefType = false;
int count = 0;
while (itr.hasNext()) {
   scwr = (SimpleClassWithRef) itr.next();
   System.out.println("fetching " + scwr.toString());
   scwr.fetch();
   sc = scwr.getRef();
   try {
      sc.fetch();
   }
   catch (JavaTypeNMismatchException e) {
      // Get the referenced object as type Object
         o = session.reloadCachedObject(sc);
         needToCheckRefType = true;
   }
```

```
   if (needToCheckRefType) {
      // Got a JavaTypeNMismatchException
      needToCheckRefType = false;
      try {
         sc = (SimpleClass) o;
      }
      catch (ClassCastException e) {
         // This reference is bad
         System.out.println(
            "ClassCastException: " +
            e.getMessage() +
            ", setting reference to null");
         // Remove the reference
         scwr.setRef(null);
         // Remove sc from the cache
         session.dropCachedObject(sc);
      }
   } // If need to check
   count++;
} // While more objects
System.out.println("count of objects: " + count);
session.commit();
```

Removing Suspect Cached Data

In addition to dropping or reloading cached information for a particular object, you can drop all objects of a particular class from every session's cache or you can drop all objects that belong to a particular session.

If your application detects repeated problems with its cached data, you can try either of the following corrections:

- Call connection object's `dropClass` method to drop all cached information about a particular class. This method drops all objects of the specified class from the cache of every session; in addition, it drops the schema class description from the cache.

  If you want to continue to retrieve or create objects of the class, you must ensure that the schema class description is restored. If the dropped class uses a custom schema class name, after dropping it, you must call the connection object's `setSchemaClassName` method to reset the schema class name, then call the connection object's `registerClass` method to retrieve the schema class description from the federated database to the cache.
- Call a particular session's `terminate` method, which terminates the session and deletes its entire class. You may not use the session after you terminate it.

If problems do not appear to be localized to a small number of classes or a small number of sessions, the best approach is to terminate all sessions in your application and create one or more new sessions for future interactions with the federated database.

Internal Persistent Objects

When you store a persistent object containing a non-null field of one of the following Java types, an internal persistent object for that field is created in the same container:

- Java classes related to date or time:

  ```
  java.util.Date
  java.sql.Date
  java.sql.Time
  java.sql.Timestamp
  ```

- Java array types (for example `long[]`)

Furthermore, an internal persistent object is created for *each element* of an array of the following types:

```
String[]
java.util.Date[]
java.sql.Date[]
java.sql.Time[]
java.sql.Timestamp[]
```

NOTE A `String` field is *not* stored as an internal persistent object; however, the elements of a `String[]` array *are* stored as internal persistent objects.

You generally do not need to deal directly with these internal persistent objects, but you need to be aware of their existence in the following circumstances:

- If you move an object from one container to another, any internal persistent objects it references remain in the original container. You can move the internal objects by calling the `moveReference` method of the federated database.
- If you store an object in a non-garbage-collectible container, any internal persistent objects it references remain in that container even after the object is deleted. If you know that no other object references the same internal persistent objects, you can call the object's `deleteReference` method to delete the internal persistent objects it references.

- When you fetch data for a persistent object that has an array field, the entire array and all its elements are read from the federated database. When you write data for a persistent object that has an array field, the entire array and all its elements are written to the federated database—even if the array and its elements were not modified.

Moving Internal Persistent Objects

If a class has fields that reference internal persistent objects, you can override the `move` method to move the internal objects along with the referencing object. If move than one object can reference the same internal persistent object, however, you may not want to override `move`.

Your `move` method should move:

- The object referenced by any field of type `java.util.Date`, `java.sql.Date`, `java.sql.Time`, or `java.sql.Timestamp`.
- Each array element in a field of type `String[]`, `java.util.Date[]`, `java.sql.Date[]`, `java.sql.Time[]`, or `java.sql.Timestamp[]`.
- The array object referenced by any field of an array type.

EXAMPLE The `Appointment` class has three fields that contain, respectively, a date, an array of times, and a string. This class overrides the `move` method to move any internal objects referenced by the moved object. Note that the `String` field `announcement` does not need to be moved because it does not reference an internal persistent object.

```
public class Appointment extents ooObj {
   private java.util.Date scheduledDate;
   private java.sql.Time availableTimes[];
   private String announcement;
   ...
   public void move(Object o) {
      moveReferences(o);
      super.move(o);
   }

   private void moveReferences(Object o) {
      markModified();
      com.objy.db.app.ooFDObj fd = getSession().getFD();

      // Move date referenced by shecheduledDate field
      fd.moveReference(scheduledDate, o);
```

```
        // Move each time in the availableTimes array
        for (int i=0; i<3; i++)
           fd.moveReference(availableTimes[i], o);

        // Move the availableTimes array
        fd.moveReference(availableTimes, o);
    }
}
```

Deleting Internal Persistent Objects

If you always store objects of your classes in garbage-collectible containers, you do not need to delete internal persistent objects. When a given internal persistent object cannot be reached from a named root, it will be available for garbage collection. On the other hand, if objects that reference internal persistent objects may be stored in non-garbage-collectible containers, you should take care to delete the internal persistent objects when they are no longer needed. “Container Types” on page 96 explains the difference between garbage-collectible and non-garbage-collectible containers.

For example, if objects of your class may be stored in non-garbage-collectible containers, and you know that a given internal persistent object can be referenced by at most one other object, you can override the `delete` and `deleteNoProp` methods to delete the internal objects along with the referencing object. Your methods should delete:

- The object referenced by any field of type `java.util.Date`, `java.sql.Date`, `java.sql.Time`, or `java.sql.Timestamp`.
- Each array element in a field of type `String[]`, `java.util.Date[]`, `java.sql.Date[]`, `java.sql.Time[]`, or `java.sql.Timestamp[]`.
- The array object referenced by any field of an array type.

EXAMPLE The `Product` class has three fields that contain, respectively, a time stamp, an array of long integers, and an array of strings. This class overrides the `delete` and `deleteNoProp` methods to delete any internal objects referenced by the deleted object. Note that the elements of the `long[]` array in the `repairCodes` field do not need to be deleted, but elements in the `String[]` array in the `notices` field do.

```
public class Product extents ooObj {
   private java.sql.Timestamp lastUpdated;
   private long repairCodes[];
   private String notices[];
   …
```

```
    public void delete() {
       deleteReferences();
       super.delete();
    }

    public void deleteNoProp() {
       deleteReferences();
       super.deleteNoProp();
    }

    private void deleteReferences() {
       markModified();
       com.objy.db.app.ooFDObj fd = getSession().getFD();

       // Delete timestamp referenced by lastUpdated field
       deleteReference(lastUpdated);

       // Delete the repairCodes array
       deleteReference(notices);

       // Delete each string in the notices array
       for (int i=0; i<3; i++) {
          deleteReference(notices[i]);
          notices[i] = null;
       }

       // Delete the notices array
       deleteReference(notices);
    }
}
```

Dead Persistent Objects

A *dead object* is an object that is no longer valid for Objectivity for Java operations. A persistent object becomes dead when it is deleted or when the session that owns it is terminated.

- When an object is deleted, it is removed from the federated database. A delete operation converts the session's local representations of deleted objects into dead objects. Dead objects that result from delete operations may correspond to any of the following objects:
  - A basic object or container that was deleted during a transaction.
  - A basic object whose container was deleted during a transaction.

- A basic object or container whose database was deleted during a transaction.

If the transaction is aborted, an object that was made persistent during the transaction goes back to the transient state; an object that was retrieved remains a dead object. If the object's database was deleted, the object no longer exists in the federated database; however, if the object itself (or its container) was deleted, the object continues to exist in the federated database after the transaction is aborted. The local representation of the object is a dead object, but you may retrieve the object again if you need to work with it.

- When a persistent object is copied, the new copy is made persistent automatically. If the transaction is aborted before the new copy has been written to the federated database, the new copy is converted to a dead object.
- When a persistent object is moved and the transaction is aborted before the moved object has been written to the federated database, the object is converted to a dead object.

When a persistent object becomes a dead object, it loses any relationships it had to other objects.

Any operation that you would normally perform on a persistent object is invalid on a dead object; if you attempt such an operation, an `ObjectIsDeadException` will be thrown. You can call an object's `isDead` method to test whether it is dead.

After a session is terminated, all the objects that belong to the session will behave like dead objects. Although `isDead` will return false for these objects, an `ObjectIsDeadException` will be thrown if you attempt to operate on them.

9

Persistent Collections

A *persistent collection* is an aggregate persistent object that can contain a variable number of elements. The elements of a collection can be either persistent objects or key-value pairs. In the latter case, the values are persistent objects; the keys may be either strings or persistent objects. You can create persistent collections to organize persistent objects.

In This Chapter

Persistent-Collection Classes

Collections are classified according to whether the order of the elements is relevant:

- The elements of an *unordered collection* are kept in an unspecified order; the relative order of any particular pair of elements is subject to change.
- The elements of an *ordered collection* are maintained in a particular order. Ordered collections are further classified by how their order is determined.
  - If elements are sorted according to some criteria of the elements themselves, the collection is said to be *sorted*.
  - If the operations that add elements to the collection determine their order, the collection is simply said to be ordered (but not sorted).

Collections can also be classified according to their implementation. *Scalable* collections can increase in size with minimal performance degradation; *nonscalable* collections cannot. Objectivity for Java collections are implemented using three different mechanisms:

- Nonscalable unordered collections use a traditional hashing mechanism.
- Scalable ordered collections use B-tree data structures.
- Scalable unordered collections use an extendible hashing mechanism.

Objectivity for Java provides the persistence-capable collection classes listed in the following table.

Class	Used for	Description
`ooTreeList`	List	Scalable ordered collection of objects that can contain duplicates and null elements.
`ooHashSet`	Set	Scalable unordered collection of objects with no duplicates and no null elements.
`ooTreeSet`	Sorted set	Scalable sorted collection of objects with no duplicates and no null elements.
`ooMap`	Name map	Nonscalable unordered collection of key-value pairs in which the key is a string and the value is a persistent object or null. Maintains referential integrity.
`ooHashMap`	Object map	Scalable unordered collection of key-value pairs in which the key is a persistent object and the value is a persistent object or null.
`ooTreeMap`	Sorted object map	Scalable sorted collection of key-value pairs in which the key is a persistent object and the value is a persistent object or null.

Referential Integrity

Referential integrity is a characteristic of a persistent collection that ensures that the collection has references only to objects that actually exist. Maintaining referential integrity requires that, when an object is deleted, any reference from a persistent collection to the deleted object is removed. Name maps can maintain referential integrity automatically; scalable collections cannot.

Name Maps

By default, a name map maintains referential integrity of its elements. That is, the name map ensures that each object in the name map is a valid persistent object. When a persistent object in a name map is deleted, Objectivity / DB automatically removes the corresponding key-value pair from the name map.

After you create a name map and before you add any elements, you can call its `setIntegrityMaintained` method to disable the automatic maintenance of its referential integrity. When you do so, you reduce the overhead in adding and deleting elements; however, you become responsible for ensuring that the name map does not contain any dangling references to deleted objects.

Scalable Collections

Scalable collections do not maintain referential integrity. Before you delete an object from the database, you are responsible for removing it from any persistent collection to which it belongs. You can restore the referential integrity of any of these collections by calling its `removeAllDeleted` method. If the collection contains any persistent objects that have been deleted, that method removes the deleted objects from the collection.

Properties of a Collection

A collection has properties that affect its growth, the storage of any auxiliary objects it may use, and concurrency of access to its objects. The particular properties supported by each collection class depend on how collections of that class are implemented. Before you create a persistent collection of a given class, you should be familiar with the relevant properties of that class.

Nonscalable Unordered Collections

Objectivity for Java supports one type of nonscalable unordered collection, namely name maps. Name maps are implemented with a traditional hashing mechanism:

- The elements of a name map are stored in a hash table.

- Hash values are computed from the key of each element.

The hash table of a name map can grow dynamically; however, increasing its size requires rehashing the entire hash table.

When you create a name map, you can specify the following growth characteristics of its hash table.

- The *initial number of bins* (hash buckets). For optimal performance, the number of hash buckets should always be a prime number.
- The *maximum average density*, that is, the average number of elements per hash bucket allowed before the hash table must be resized. The hash table is resized whenever:

  ```
  totalElements >= numberBins * maximumAverageDensity
  ```

- The *growth factor*. This number gives the percentage by which the hash table grows when it is resized. Each time the hash table is resized, the number of hash buckets is increased by the growth factor, then rounded up to the nearest prime number.

Scalable Ordered Collections

Scalable ordered collections (lists, sorted sets, and sorted object maps) are implemented as B-trees. The B-tree organization supports efficient binary search and reduces the runtime overhead of inserting elements into the middle of the collection.

An element's position within the ordered collection is given by a zero-based *index*.

B-Tree Nodes and Arrays

A B-tree is composed of nodes, each with a corresponding array. The array for a non-leaf node contains references to the first leaf-node descendants along each branch from the node; the array for a leaf node contains references to elements of the collection whose indexes are within a particular range. B-Tree nodes and their arrays are internal objects that the collection creates as needed; you never create them or work with them directly.

A newly instantiated ordered collection consists of the root node and its array. As the collection grows, additional nodes are created as necessary. When each node is created, its corresponding array is also created. Most existing nodes do not need to be modified; in fact, those nodes can be accessed for read or write while a new node is being added.

Node Size

Every ordered collection has a *node size* property that determines the maximum size of a node in the collection's B-tree; that is, the maximum number of references in its array. As elements are added to a newly created ordered collection, they are assigned to the root B-tree node until the collection's node size is reached. At that point, a new node must be added to the tree.

The default node size allows each array to contain as many references as will fit on a single page in the federated database. When you create an ordered collection, a parameter to the constructor allows you to specify a different node size.

- You may choose a small node size to minimize lock conflicts when multiple applications update the collection simultaneously.
- You may choose a large node size to minimize the number of nodes in the B-tree. For example, if you don't expect the collection to get very large, you might choose a large node size to force all elements to be stored in a single node.

NOTE Once an ordered collection has been created, you cannot set or change its node size.

Containers for Nodes and Arrays

The B-tree nodes and their arrays are all persistent objects and, as such, they are stored in containers. To access an element of an ordered collection, an application must be able to obtain a lock on the B-tree node and array corresponding to the element's index. As is the case with all persistent objects, locking a B-tree node or array locks its container, effectively locking any other objects stored in the same container. As a consequence, the distribution of nodes and arrays in containers affects concurrent access to the collection.

You can increase concurrent access to the collection by making sure that the collection's nodes and arrays are distributed in different containers. Of course, the more containers used for internal objects, the larger the federated database will be. "Assigning Basic Objects to Containers" on page 99 describes how the use of containers can affect concurrency, storage requirements, and runtime performance.

Node Containers

At any given time, an ordered collection uses a particular container, called its *current node container*, to store its newly created B-tree nodes. The collection's container is its initial node container.

An ordered collection serves as the root node of its B-tree; that is, no additional B-tree node object is required if all elements can fit in the root node. If the number of elements exceeds the capacity of a single node, the collection creates additional nodes, as necessary, to accommodate the elements.

When the collection creates a new node, it clusters the B-tree node object in its current node container. When the current node container is full, the collection creates a new container, which becomes the current node container. Each new node container is created in the same database as the previous node container. When the database contains at least 30,000 containers, a new database is created automatically for the next node container.

If an ordered collection is clustered in a garbage-collectible container, all its node containers are garbage-collectible. If the collection is clustered in a non-garbage-collectible container, all its node containers are non-garbage-collectible.

An ordered collection stores only B-tree nodes in the node containers it creates. An application typically does not access those containers directly.

Array Containers

At any given time, an ordered collection uses a particular container, called its *current array container*, to store the arrays for its new B-tree nodes.

When you create an ordered collection, Objectivity for Java creates the array for the collection's root B-tree node. By default, Objectivity for Java also creates the collection's initial array container and stores the array in that container. The new container is created in the same database as the ordered collection. If an ordered collection is clustered in a garbage-collectible container, its initial array container is garbage-collectible; if the collection is clustered in a non-garbage-collectible container, its initial array container is non-garbage-collectible.

If you prefer, you can specify an ordered collection's initial array container as a parameter to the constructor that creates the collection. For example, you might want to minimize the number of containers used by a collection by specifying the collection's container as its initial array container. Alternatively, you might use a container in a different database as the collection's initial array container. In that case, the collection's arrays would be stored in a different database from its nodes.

As the collection creates a new array for each new node, the arrays are added to the initial array container until that container is full. Then, a new container is created and used as the current array container.

As more nodes are needed, the ordered collection stores each new node's array in its current array container until that container is full; it then creates a new current array container. Each new array container is created in the same database as the previous array container. When the database contains at least 30,000 containers, a new database is created automatically for the next array container.

All array containers for a given ordered collection are of the same type. If the initial array container is garbage-collectible, all subsequent array containers will be garbage-collectible; if the initial array container is non-garbage collectible, all subsequent array containers will be non-garbage collectible.

An ordered collection stores only arrays in the array containers it creates. An application typically does not access those containers directly.

Tree Administrator

Every ordered collection has an internal object, called a *tree administrator*, to manage the containers for the collection's nodes and arrays. The collection's tree administrator is created when the collection itself is created. By default, the tree administrator is stored in a new container in the same database as the ordered collection itself. If you want a collection's tree administrator to be stored in an existing container, however, you can specify that container as a parameter to the constructor that creates the ordered collection. For example, instead of having Objectivity for Java create a new container just for the tree administrator, you might choose to store the tree administrator in the same container as the ordered collection itself.

A collection's tree administrator is created when the collection itself is created. By default, the tree administrator is stored in a new container in the same database as the ordered collection itself. If you want a container's tree administrator to be stored in an existing container, however, you can specify that container as a parameter to the constructor that creates the ordered collection. For example, instead of having Objectivity for Java create a new container just for the tree administrator, you might choose to store the tree administrator in the same container as the ordered collection itself.

The tree administrator uses two properties of an ordered collection to control when the current node container and the current array container are considered "full."

- The *maximum nodes per container* property specifies how many nodes can be clustered together in the same container. Because B-tree nodes are small objects, many of them can fit on a single page in a federated database. Because nodes are not updated frequently, many can be clustered in the same container without causing locking problems. The default value for this property depends on the chosen page size; it is calculated as:

  ```
  pageSize / 47
  ```

  To use a different value for this property, call the ordered collection's `maxNodesPerContainer` method.

  Changing the maximum nodes per container affects only the collection's current node container and any node containers created in the future. If you reduce the number of nodes per container, existing node containers are left

with more nodes than the new maximum; if you increase the number, existing node containers are left with fewer nodes than the new maximum.

- The *maximum arrays per container* property specifies how many arrays can be clustered together in the same container. One array fills up an entire page in the federated database. It is typical for a node's array to be updated frequently; the default value of 1 for this property minimizes lock conflicts. If you know that a particular collection will be used by a single user, locking is not an issue. In that case, a larger value, such as 5000, may be appropriate. To use a different value for this property, call the `maxVArraysPerContainer` method.

  Changing the maximum arrays per container affects only the collection's current array container and any array containers created in the future. It does not affect existing array containers that are already full.

Comparator

Every sorted collection has an associated comparator that controls how elements are sorted. The comparator defines a total ordering to be used by the underlying B-tree.

- The default comparator for a sorted set sorts elements by increasing object identifier (OID).
- The default comparator for a sorted object map sorts elements by increasing OID of their keys.

You can implement an alternative sorting criteria with an application-defined comparator class. See "Defining a Comparator Class for Sorted Collections" on page 204.

To use your own sorting criteria, assign an instance of your comparator class to a sorted collection when you create the collection. If you do not assign a comparator explicitly, the collection uses the default comparator. For additional information, see "Using a Comparator" on page 206.

Scalable Unordered Collections

Scalable unordered collections (unordered sets and unordered object maps) are implemented with an extendible hashing mechanism that uses a two-level directory structure to locate elements. You can think of the elements in the unordered collection as being divided into disjoint groups, each with its own directory. The top-level directory identifies a *hash bucket*, which acts as the directory for one of the disjoint groups. A hash bucket locates elements whose hash values are within a certain range. Adding elements may cause individual hash buckets to be rehashed, but the entire collection never needs to be rehashed.

The two-level directory structure allows the unordered collection to increase in size with minimal performance degradation. Regardless of the size of the

collection, accessing an element requires one look-up in the top-level directory and one look-up in the appropriate hash bucket.

Hash-Buckets

Hash buckets are persistent objects that the collection creates as needed and uses internally; you never create them or work with them directly. By default, one initial hash bucket is created for the collection; if you prefer, you can specify a different number of initial hash buckets as a parameter to the constructor that creates the unordered collection object. The number hash buckets created initially is a power of two; if you do not specify a power or two, the next higher power of two is used. For example, if you specify 5 initial hash buckets, 8 initial hash buckets are actually created. If the collection has *N* hash buckets, the first *N* high-order bits of an object's hash value are used to determine which hash bucket it belongs to.

Preallocating multiple hash buckets increases the speed of adding and finding map elements. If each hash bucket is stored in a separate container (the default behavior), preallocating hash buckets also reduces the chance of lock conflicts. However, an unordered collection with a large number of initial hash buckets requires more disk space, more memory for the directory, and more time to create.

As an unordered scalable collection grows past the capacity of its existing hash buckets, new hash buckets are added.

Hash-Bucket Size

Every scalable unordered collection has a *bucket size* property that determines the size of a hash bucket in its hash table. The size of a hash bucket is the number of elements that can be hashed into each bucket. The default hash-bucket size is 30011. If you want to use a different bucket size, you can specify the desired size as a parameter to the constructor that creates the unordered collection object.

For optimal performance, the hash-bucket size should be a prime number. If you specify a number that is not prime, the next higher prime number is computed and used as the actual hash-bucket size.

Containers for Hash Buckets

The hash buckets of a scalable unordered collection are persistent objects; as such, they are stored in containers. To access an element of a scalable unordered collection, an application must be able to obtain a lock on the hash bucket corresponding to the element's hash value. As is the case with all persistent objects, locking a hash bucket locks its container, effectively locking any other

objects stored in the same container. As a consequence, the distribution of hash buckets in containers affects concurrent access to the collection.

By default, a separate hash-bucket container is created for each of the collection's initial hash buckets—that is, for each of the hash buckets that are created by the constructor. As the collection grows, and additional hash buckets are created, a new hash-bucket container is created by default for each new hash bucket; this default behavior optimizes concurrent access to the collection. However, the more containers used for hash buckets, the larger the federated database will be; for a discussion of the trade-offs between concurrency and storage requirements, see "Assigning Basic Objects to Containers" on page 99.

If you prefer to store more than one hash bucket in a container, you can specify an existing container in which to store the all the initial hash buckets. In addition, you can change the number of hash buckets that are clustered in the same container using the collection's hash administrator; see "Hash Administrator" on page 202.

By default, the first hash-bucket container is created in the same database as the unordered collection. Additional hash-bucket containers are created in the same database. If you specify an existing container for the initial hash buckets, additional hash-bucket containers will be created in the same database as that container. New hash-bucket containers are added to a given database until it contains at least 30,000 containers. Then a new database is created automatically for subsequent hash-bucket container.

All hash-bucket containers for a given unordered collection are of the same type. If an unordered collection is clustered in a garbage-collectible container, all its hash-bucket containers are garbage-collectible; if the collection is clustered in a non-garbage-collectible container, all its hash-bucket containers are non-garbage collectible.

An unordered collection stores only hash buckets in the hash-bucket containers it creates. An application typically does not access those containers directly.

Hash Administrator

Every scalable unordered collection has an internal object, called a *hash administrator*, to manage the containers for the collection's hash buckets.

A collection's hash administrator is created when the collection itself is created; the hash administrator is stored in a new container in the same database as the unordered collection itself. The collection's hash administrator is created when the collection itself is created; by default, the hash administrator is stored in a new container in the same database as the unordered collection itself. If you prefer, you can specify an existing container in which to store the hash administrator.

The hash administrator uses a property of the unordered collection to control when the current hash-bucket container is considered "full". The *maximum buckets per container* property specifies how many hash buckets can be clustered together in the same container. It is typical for a hash bucket to be updated frequently. The default value for this property is 1, which minimizes lock conflicts. If you know that a particular collection will be accessed only by a single user, locking is not an issue. In that case, a larger value may be appropriate. To use a different value for this property, call the collection's `maxBucketsPerContainer` method.

Changing the maximum buckets per container affects only the clustering of hash buckets that are created *after* the call. After you change the value from the default of 1 to a larger number, newly created hash buckets will be clustered in the collection's most recently created hash-bucket container until the maximum number is reached. New hash-bucket containers will created as needed, and the maximum number of has buckets will be clustered in each.

Comparator

Every scalable unordered collection has an associated comparator that controls how an element's hash value is computed.

- The default comparator for an unordered set computes an element's hash value from its OID.
- The default comparator for an unordered object map computes an element's hash value from the OID of its key.

You can implement an alternative hashing algorithm with an application-defined comparator class. See "Defining a Comparator Class for Unordered Collections" on page 205.

To use your own hashing algorithm, assign an instance of your comparator class to a scalable unordered collection when you create the collection. If you do not assign a comparator explicitly, the collection uses the default comparator. For additional information, see "Using a Comparator" on page 206.

Application-Defined Comparator Classes

A *comparator* is an object of a concrete descendant class of `ooCompare`. It provides a comparison function for ordering elements of sorted collections, and a hashing function for computing the hash values for elements of unordered collections.

You can implement your own sorting or hashing behavior in an application-defined comparator class; to do so, you define your own subclass of `ooCompare` and override the `compare` and / or `hash` method as appropriate. An application-defined comparator also allows you to identify elements of a collection based on persistent data in the element or its key.

NOTE Using an application-defined comparator slows down most operations on a collection.

Defining a Comparator Class for Sorted Collections

If your application uses sorted collections with elements (or keys) of some particular class, you may want to sort the elements based on the data in some persistent field(s) of each element (or key). You might additionally want to use the same data as a unique identifier for an element (or key) of a sorted collection so that you can look up the element with particular value(s) in the relevant field(s).

Comparing Elements of a Sorted Collection

Elements in a sorted collection are ordered based on the sorting criteria embodied in the `compare` method of its comparator. That method compares a persistent object in the collection with another object and indicates the second object's position in the collection relative to the persistent object.

When you define a comparator class to be used with sorted collections, you can override the `compare` method to compare two persistent objects based on whatever sorting criteria you choose. Typically the comparison uses some data that uniquely identifies an object of its class. Uniqueness is necessary because the `compare` method must impose a total ordering on the elements.

For example, suppose the elements of a sorted set are objects of the `Person` class, and you know that no two elements of the set will have the same name. You might choose to order the elements based on the string in each element's `name` field. You could define the `compare` method of your comparator class to verify that its parameters are `Person` objects and, if so, to compare the name strings of the two objects. A comparator of this class could also be used for a sorted object map whose keys are `Person` objects.

Uniquely Identifying Elements of a Sorted Collection

A comparator class for sorted collections can optionally provide the ability to identify an element (or key) based on its sorting criteria. This ability allows you to use the data that identifies a particular element to:

- Look up that element in a collection
- Test whether a collection contains that element
- Remove that element from a collection

In the preceding example, the application-defined comparator class could support the ability to use a name to uniquely identify an element in a sorted set of `Person` objects, or in a sorted object map in which the keys are `Person` objects.

If you want your comparator class to be able to identify an element (or key) of a sorted collection based on its sorting criteria, you should implement the `compare` method to be able to compare an element (or key) either with another persistent object or with data that identifies a persistent object. In our example, the `compare` method would be able to compare a `Person` object either with another `Person` object or with a string containing a person's name.

Defining a Comparator Class for Unordered Collections

If your application uses scalable unordered collections with elements (or keys) of some particular class, you may want to hash elements based on the data in some persistent field(s) of each element (or key). You might additionally want to use the same data as a unique identifier for an element (or key) of an unordered collection so that you can look up the element with particular value(s) in the relevant field(s).

Hashing Elements of an Unordered Collection

When you define a comparator class to be used with unordered collections, you can override the `hash` method to compute hash values for persistent objects using whatever criteria or algorithm you choose.

NOTE Your `hash` method should distribute hash values throughout the range of 32-bit integers. In particular, the distribution of the high-order bits should be relatively even, because those bits are used to select a hash bucket. All bits of the hash value are used to select a position within the hash bucket.

For example, suppose the elements of an unordered set are objects of the `Employee` class, representing people employed by a particular company in the United States; the `SSN` field of this class is a string representation of an employee's Social Security Number (SSN). You might define a comparator class whose `hash` method computes hash values from SSNs. The `hash` method would verify that its parameter is a an `Employee` object and, if so, convert the SSN string to a 32-bit integer to be used as the hash value. A comparator of this class could also be used for an unordered object map whose keys are `Employee` objects.

Uniquely Identifying Elements of an Unordered Collection

A comparator class for unordered collections can optionally provide the ability to identify an element (or key) based on the data from which its hash value is computed. This ability allows you to use the data that identifies a particular element to:

- Look up that element in a collection
- Test whether a collection contains that element

In the preceding example, the application-defined comparator class could support the ability to use a Social Security Number to uniquely identify an element in an unordered set of `Employee` objects or in an unordered object map in which the keys are `Employee` objects.

If you want your comparator class to be able to identify an element (or key) of an unordered collection based on class-specific data, you must override both the `hash` method and the `compare` method.

- Your `hash` method should be able to compute a hash value either from a persistent object or from data that identifies the persistent object.

  In our example, the `hash` method would be able to compute a hash value either from an `Employee` object or from a Social Security Number. The method might allow the SSN to be specified in a variety of different forms.
- Your `compare` method should be able to compare an element (or key) of the unordered collection either with another persistent object or with data that identifies a persistent object.

  In our example, the `compare` method would be able to compare an `Employee` object either with another `Employee` object or with a SSN specified in any form that your `hash` method supports.

Using a Comparator

To use a comparator of an application-defined comparator class, you create it, make it persistent, and assign it to one or more collections. Special care may be required when modifying objects in the collection (page 207).

Creating a Comparator

To create a comparator, instantiate your comparator class and make it persistent by clustering it in the desired container. A comparator is locked whenever you access its associated collection. To avoid locking conflicts, you typically cluster the comparator in a separate container. If the comparator is stored in the same container as the collection, applications may fail to get the necessary read lock on the comparator when another process is updating the collection.

The persistent data for a persistent collection references its comparator. Your application should not explicitly save any comparator. For example, you should not add a comparator to a collection or save a comparator in an instance variable of any persistent object. Typically, an application uses only comparators that it creates dynamically; it does not explicitly retrieve comparators from the database.

Assigning a Comparator to a Collection

After creating the comparator and making it persistent, you can assign it to any collections that need to use the comparator's particular comparison and hashing methods. You assign a comparator to a collection by passing the comparator as a parameter to the constructor that creates the collection.

NOTE Once a collection has been created, you cannot set or change its comparator.

Modifying Objects in the Collection

A collection's comparator may affect how an application modifies objects in the collection.

- If a sorted collection's comparator sorts elements on the basis of some field of an object, modifications to an element of a sorted set or to the key of an element in a sorted object map might cause the element's appropriate order in the collection to be changed. To make such a modification, you must first remove the affected element from the collection. After making the desired modification, you add the element back to the collection, which will insert it at its (new) correct position.
- If an unordered collection's comparator computes hash value on the basis of some field of an object, modifications to an element of an unordered set or to the key of an element in an unordered object map might cause the element's hash value to be modified. To make such a modification, you must first remove the affected element from the collection. After making the desired modification, you add the element back to the collection, which will associate it with its (new) correct hash value.

Interoperability

If a persistent collection uses a comparator of an application-defined class, the data for the collection in the federated database includes a reference to the comparator. Any application that retrieves the collection will also retrieve its comparator. As a consequence, any application that retrieves the comparator must include a comparator class with the same schema class name as the comparator's class.

Objectivity / DB provides persistent storage for data only, not for methods, so the federated database does not store `compare` and `hash` methods of the comparator. The comparator class in the retrieving application must include implementations for these methods; furthermore, those methods must use the same sorting criteria and the same hashing algorithm as the application that stored the collection.

WARNING Data corruption may occur if applications share a collection but use different `compare` and `hash` methods for the collection's comparator.

If the applications that use a given persistent collection are all implemented in the same language (for example, Java), they can all share the definition of the comparator class. If the applications are written in different languages (for example, some in Java and some in C++), their comparator classes must have equivalent comparison and hashing methods.

Working With a Persistent Collection

You must create a persistent collection during a transaction; the new collection object belongs to that transaction's session and all persistent objects that you add to the collection must belong to the same session. You create a persistent collection in the same way as you would a typical Java object with the `new` operator, and you make it persistent as you would for an object of any persistence-capable class. For example, you might name the collection, reference it in a persistent field of a named root (or other persistent object), or use it as an element of another persistent collection.

You can retrieve a persistent collection from the database just as you retrieve any persistent object. See Chapter 11, "Retrieving Persistent Objects".

After you create or retrieve a persistent collection, you can call its methods to:

- Add and remove elements.

  Any object you add to a collection must be an instance of a persistence-capable class. If the object is transient, it is made persistent when it is added to the collection; the default clustering strategy assigns it to the collection's container.
- Test whether the collection contains particular elements.
- Retrieve elements from the collection.

For a complete description of the available methods, see the descriptions of the various persistent-collection classes.

Examples in Chapter 6, “Defining Persistence-Capable Classes,” Chapter 10, “Naming Persistent Objects,” and Chapter 11, “Retrieving Persistent Objects,” illustrate the creation and use of a name map.

10

Naming Persistent Objects

Naming a persistent object allows you to identify the object with a name that is meaningful to you or the users of your application. Naming also facilitates retrieval of the object from the database by enabling you to look it up by name. This chapter describes how to give a persistent object (including a persistent collection or a persistent container) names of any of the following types:

- A root name in a federated database or database.
- A scope name in the scope of a federated database, database, container, or basic object.
- A name in a name map used as an application-defined dictionary.

Unlike other persistent objects, a container can also have a system name because it is a storage object. You can specify a container's system name as a parameter to the `addContainer` method that adds the container to a database; to look up the container by its system name, you call the `lookupContainer` method of its database. For more information, see "System Names" on page 26.

In This Chapter

Named Roots

A *named root* is a persistent object that can be located by a *root name*, which is unique within the federated database or a particular database. Typically, a named root is the root object in a directed graph of persistent objects. When you first write the named root to the database, you store the entire graph. You can use the root name to retrieve the named root, and you can then traverse links of the named root to retrieve all the objects in its graph. See “Finding Objects in a Graph” on page 225.

Root Names

A root name can be any valid Java string of arbitrary length. A federated database or database stores its root names in its *root dictionary* (a name map). An object can have more than one root name within the same database and can have a root name in more than one database. Each root name in a particular root dictionary must be unique within that root dictionary; that is, a given database cannot have more than one object with the same root name.

A database with named roots has a special container, called its *roots container*, where it stores its root dictionary. A roots container is used primarily for the root dictionary, but the default clustering strategy may store named roots (and the objects in their object graphs) in the roots container. You can increase concurrent access to the root directory by storing named roots in different containers. For more information about clustering and concurrency, see “Assigning Basic Objects to Containers” on page 99.

Making an Object a Named Root

To give an object a root name, you call the `bind` method of the federated database or database in which you want the object to be a named root. The session that owns the local representation of the federated database or database must be in a transaction.

The `bind` method takes the object to be named and the root name as parameters. The session must be able to obtain a write lock on the root dictionary of the federated database or database. The `bind` method throws a checked exception if the federated database or database already has a root with the specified root name.

If an object is transient when you make it a named root, the `bind` method makes the object persistent. In that case, the default clustering strategy clusters the object into the roots container. Once the object has been made persistent, it belongs to the same session as the federated database or database in which it was named.

NOTE If you store a named root in the roots container or any other garbage-collectible container, and your application depends on the `oogc` administrative tool to delete unreferenced objects, be sure that all objects in the named root's object graph are also stored in garbage-collectible containers. For more information, see "Garbage-Collectible Containers" on page 96.

EXAMPLE This code fragment makes an object of the `Salesperson` class a named root in the database `salesDB`. The complete method definition appears in the `Sales.Interact` programming example (see page 422). The `Interact` class has a static field called `session`, which is initialized to contain an instance of the `Session` class when the connection is opened; all examples in this chapter use that session object for transaction control.

```
// Static utility to add a salesperson
public static Salesperson addSalesperson (...) {
   ...
   session.begin();
   ...
   ooDBObj salesDB = ...;
   ...
   Salesperson salesperson = ...;
   ...
   String rootname = ...;
   // Make new salesperson a named root
   try {
      salesDB.bind(salesperson, rootname);
   } catch (ObjectNameNotUniqueException e4) {
      System.out.println("Employee ID " + rootname +
                         " already in use.");
      session.abort();
      return null;
   }
   session.commit();
   ...
}
```

Working With Root Names

Once you have defined root names in the connected federated database or a particular database, you can take any of the following actions. All operations involving root names must be performed while the session that owns the local representation of the federated database or database is in a transaction.

- Retrieve an object with a specified root name with the `lookup` method.

  The session must be able to obtain a read lock on the root dictionary.
- Find all the root names in a root dictionary with the `rootNames` method.

  The session must be able to obtain a read lock on the root dictionary.
- Delete a root name from a root dictionary with the `unbind` method.

  The session must be able to obtain a write lock on the root dictionary.
- Replace a named root.

  Although named roots are stored in name maps, there is no method to directly replace an object associated with a name in the root dictionary. If you want to replace an object in the root dictionary, you must first unbind a name and then rebind the name to a new object.

NOTE If a named root is stored in the roots container or any other garbage-collectible container, removing its root name makes the object subject to garbage collection unless the object can be reached from another named root or from an indexed object. If the named root cannot be reached, all objects in its graph are also subject to garbage collection unless they can be reached by some other path.

Name Scopes

Objectivity / DB provides a mechanism that allows you to name persistent objects within the scope of a particular *scope object*. A scope object can be the federated database, a database, a persistent container, or a persistent basic object.

NOTE If you want to use a container as a scope object, the container must be hashed. If you want to use a basic object as a scope object, its container must be hashed. When you add a container to a database, a parameter to the `addContainer` method specifies whether the container should be hashed.

A scope object defines a *name scope*, that is, a set of names for persistent objects. Each name in the set is called a *scope name* and uniquely identifies a persistent

object to the scope object (but not to other objects). You can think of a name scope as a local name space defined by the scope object.

Scope Names

A scope name can be any valid Java string of arbitrary length. Different scope objects may use different scope names to refer to the same object.

A scope object uses the hashing mechanism of a hashed container to associate each name in the name scope with the appropriate object. The following table identifies the hashed container used by each kind of scope object.

Scope Object	Uses Hashing Mechanism Of
Basic object	That basic object's container
Container	That container
Database	The default container of that database
Federated database	The default container of the default database of that federated database

The default clustering strategy may store named objects in the hashed container used by the scope object. You can increase concurrent access to the scope names by storing scope-named objects in different containers. You should store scope-named objects in non-garbage-collectible containers. For more information about clustering and concurrency, see "Assigning Basic Objects to Containers" on page 99.

Defining a Scope Name

To give an object a scope name, you call the `nameObj` method of the scope object. The session that owns the local representation of the scope object must be in a transaction.

The `nameObj` method takes the object to be named and the scope name as parameters. The session must be able to obtain a write lock on the hashed container used by the scope object. A runtime error is thrown if `nameObj` fails, for example, because some object already has the specified name in the scope object's name scope.

If an object is transient when you give it a scope name, the `nameObj` method makes the object persistent. In that case, the default clustering strategy clusters the object into the hashed container used by the scope object. Once the object has been made persistent, it belongs to the same session as the scope object.

You cannot associate a scope name with a null object reference. If you have a need to associate names with null object references, you must use one of the other two naming mechanisms.

EXAMPLE This code fragment names an object of the `Contact` class in the scope of the container `scope`. The complete method definition appears in the `Sales.Interact` programming example (see page 422).

```
// Static utility to add a contact
public static Contact addContact (...) {
   ooContObj scope;
   ...
   session.begin();
   ...
   Contact contact = ...;
   ...
   scope = ...;
   ...
   String scopename = ...;
    // Give the contact a scope name
   try {
      scope.nameObj(contact, scopename);
   } catch (ObjyRuntimeException e) {
      System.out.println("Contact " + scopename +
                         " already exists.");
      session.abort();
      return null;
   }
   session.commit();
   ...
}
```

Working With Scope Names

Once you have defined scope names for a scope object, you can take any of the following actions. These operations must be performed while the session that owns the local representation of the scope object is in a transaction.

- Retrieve the object with a scope name from that scope object with the `lookupObj` method.

  The session must be able to obtain a read lock on the hashed container used by the scope object.

- Retrieve the scope name of an object in the scope of that scope object with the `lookupObjName` method.

  The session must be able to obtain a read lock on the hashed container used by the scope object.
- Delete an object's scope name from that scope object with the `unnameObj` method.

  The session must be able to obtain a write lock on the hashed container used by the scope object.
- If the scope object is a basic object or a container, find all objects named in its scope with the `scopedObjects` method. The session must be able to obtain a read lock on the hashed container used by the scope object.
- Find all scope objects that have scope names for an object with the object's `scopedBy` method. The session must be able to obtain read locks on all the hashed containers used by the scope objects.

Application-Defined Dictionaries

You can implement an application-defined dictionary with a name map in which the keys are object names. You can create as many such dictionaries as you like; each dictionary represents a separate name space for objects. You can name a given object in as many dictionaries as you like.

One advantage of using an application-defined dictionary is that you have control over the clustering and growth performance of the name map that implements the dictionaries.

Creating a Name Map

You can create a name map and make it persistent just as you would with any persistent object. Before you create the name map, you should decide:

- Where to store the name map.

  In many applications, a name map is a resource that will be shared by many users who need to name and look up objects. If you expect names to be added to, and deleted from, the dictionary by multiple users, you need to plan for concurrent access to the name map. For example, you could put the name map in a container by itself or in a container with few other objects that require updates. For more information about clustering and concurrency, see "Assigning Basic Objects to Containers" on page 99.

- How to set the name map's growth parameters.

  You can set the initial number of bins (hash buckets), the maximum average number of elements in a bin, and the percentage by which the name map's hash table grows when it is resized.

Your decisions determine how to set the parameters to the constructor that you use to create the name map. For additional information about growth characteristics and a detailed description of the constructors that create name maps, see the `ooMap` class description in the Objectivity for Java reference.

In addition to creating the name map, you must plan how applications will retrieve it from the database.

- If a name map represents a name space associated with a particular persistent object, you can store the name map in a persistent field of that object. Applications can then retrieve the persistent object and get the name map from its field.
- If a name map represents an application-wide name space, you may name the name map (using any naming mechanism). Applications can then look up the name map by its name.

Adding a Name to the Dictionary

After creating a name map to be used as an application-defined dictionary, you can add a name to the dictionary by calling the `add` or `forceAdd` method of the name map. The session that owns the local representation of the name map must be in a transaction and must be able to obtain a write lock on the name map. Both methods take as parameters the object to be named and its proposed name; they differ in their behavior when the proposed name is already a key in the name map.

- The `add` method throws a runtime error if the name map already contains an object with the specified name.
- The `forceAdd` method adds an element even if its name is a duplicate. You should call this method only when you are certain that the name is not already in use; if you add more than one element with the same name, it is indeterminate which element would be found by `lookup`, changed by `replace`, or removed by `remove`.

  The `forceAdd` method is faster than the `add` method and can be used to initialize dictionaries when they are created. After initial creation, the `add` and `replace` methods can be used for maintenance of the dictionary.

If an object is transient when you add the name to the name map, it is made persistent. In that case, the default clustering strategy clusters the object into name map's container. Once the object has been made persistent, it belongs to the same session as the name map.

EXAMPLE This code fragment names an object of the `Client` class in the application-defined dictionary `nameTable`; the name of the client company is used as the object's name. The complete method definition appears in the `Sales.Interact` programming example (see page 422).

```
// Static utility to add a client
public static Client addClient (String companyName, ...) {
   ...
   ooMap nameTable;
   ...
   session.begin();
   ...
   Client client = ...;
   ...
   nameTable = ...;
   ...
   // Add client to name map
   try {
      nameTable.add(client, companyName);
   } catch (ObjyRuntimeException e) {
      System.out.println("Company " + companyName +
                         " already exists.");
      session.abort();
      return null;
   }
   session.commit();
   }
```

Working With Name Maps

Once you have created a name map, you can take any of the following actions. All these operations must be performed while the session that owns the local representation of the name map is in a transaction.

- Replace the object associated with a particular name in the dictionary by calling the name map's `replace` method. The session must be able to obtain a write lock on the name map.
- Remove a named object from the dictionary by calling the name map's `remove` method. The session must be able to obtain a write lock on the name map.
- Test whether a particular name appears in the dictionary by calling the name map's `isMember` method. The session must be able to obtain a read lock on the name map.

- Retrieve the object with a particular name in the dictionary by calling the name map's `lookup` method. The session must be able to obtain a read lock on the name map.
- Find all objects that have names in the dictionary by calling the name map's `elements` method. The session must be able to obtain a read lock on the name map.
- Find all the names used in the dictionary by calling the name map's `keys` method. The session must be able to obtain a read lock on the name map.

Comparison of Naming Mechanisms

All three naming mechanisms accept arbitrary Java strings as names and use a hashing mechanism to associate names with persistent objects. Scope names use a hashed container to organize the names; root names and application-defined dictionaries both use name maps.

The following table contains a summary of the limitations and advantages of the three naming mechanisms.

	Limitations	Advantages
Root Names	You must name objects in a name space defined by the federated database or a database. Each database and federated database can have only one such name space.	You can store named roots and their object graphs in garbage-collectible containers to simplify database maintenance. You can look up named objects directly through the federated database or database. If desired, you can associate a root name with a null object reference.
Scope Names	Scope objects must use hashed containers, which take more space than non-hashed containers. You cannot associate a scope name with a null object reference.	You can name objects in a name space defined by a basic object, a container, a database, or the federated database (but each scope object can have only one such name space). You can look up named objects directly through the scope object.
Names in Name Maps	You must explicitly retrieve the name map before you can look up objects.	You can name objects in any name space that is meaningful to the application; each name map represents such a name space. You can control where the name map is stored, thus improving concurrency. You can tune the name map for the performance needs of your application. If desired, you can associate a name with a null object reference.

11

Retrieving Persistent Objects

Objectivity/DB provides a variety of mechanisms for retrieving objects from a federated database. You can retrieve individual objects based on their application-defined names and links with other objects. In addition, you can search storage objects for objects of a given class, possibly restricting the search based on the values of certain persistent fields. Finally, you can traverse the entire storage hierarchy, retrieving all objects within each storage object.

In This Chapter

General Guidelines

The following general guidelines apply to retrieval methods.

The return type of every retrieval method is `Object`; you should cast the retrieved object to the appropriate class before using it.

- Any method that retrieves an object must be called while a session is in a transaction. If the session already has a local representation of the requested object, the retrieval method returns that local representation. Otherwise, it retrieves the object from the database and returns a new local representation that belongs to the session. Objectivity for Java does not allow the returned object to interact with other objects that belong to different sessions.
- Any method that searches for multiple objects will return an iterator initialized to find the desired object. An *iterator* is an instance of a class that implements the `java.util.Iterator` interface. You call the iterator's `next` method repeatedly to get a local representation of each object; for loop control, the iterator's `hasNext` method tests whether additional elements remain.

NOTE Many retrieval methods do not lock the retrieved object or fetch its persistent data. Certain methods take a parameter that specifies the lock mode for the retrieved object; those methods lock the object as indicated but do not fetch its data.

Looking Up an Object by Name

The most direct way to retrieve an object is to look it up by name. Of course, this assumes that the object has been given a name, as described in Chapter 10, "Naming Persistent Objects".

Root Name

To retrieve an object with a particular root name, call the `lookup` method of the federated database or database in which it is a named root. The session that owns the local representation of the federated database or database must be in a transaction.

The `lookup` method takes the root name as its parameter and returns the object with that root name. The session must be able to obtain a read lock on the root dictionary of the federated database or database. The `lookup` method throws a checked exception (`ObjectNameNotFoundException`) if there is no object with the specified name.

EXAMPLE This code fragment retrieves an object of the `Salesperson` class by looking up its root name in the database `salesDB`. The complete method definition appears in the `Sales.Interact` programming example (see page 422). The `Interact` class has a static field called `session`, which is initialized to contain an instance of the `Session` class when the connection is opened; all examples in this chapter use `session` for transaction control.

```
// Static utility to retrieve a salesperson
public static Salesperson lookupSalesperson (...) {
   Salesperson salesperson;
   ...
   session.begin();
   ...
   ooDBObj salesDB = ...;
   ...
   String rootname = String.valueOf(employeeID);
   // Look up the salesperson
   try {
      salesperson = (Salesperson) salesDB.lookup(rootname);
   } catch (ObjectNameNotFoundException e) {
      System.out.println("No salesperson with Employee ID " +
                         rootname);
      session.abort();
      return null;
   }
   session.commit();
   ...
}
```

Scope Name

To look up an object by its scope name, call the `lookupObj` method of the scope object. The session that owns the local representation of the scope object must be in a transaction.

The `lookupObj` method takes the scope name as its parameter and returns the object with that scope name. The session must be able to obtain a read lock on the scope object's name map. A runtime error is thrown if `lookupObj` fails to find the object; for example, because there is no object with the specified name, a runtime error is thrown.

EXAMPLE This code fragment retrieves an object of the `Contact` class by looking up its name in the scope of the container `scope`. The complete method definition appears in the `Sales.Interact` programming example (see page 422).

```
// Static utility to retrieve a contact
public static Contact lookupContact (...) {
   Contact contact;
   ...
   session.begin();
   ...
   String scopename = ...;
   ooContObj scope = ...;
   try {
      contact = (Contact) scope.lookupObj(scopename);
   } catch (ObjyRuntimeException e) {
      System.out.println("No contact named " + scopename);
      session.abort();
      return null;
   }
   ...
   session.commit();
   ...
}
```

Name in an Application-Defined Dictionary

To look up an object by its name in an application-defined dictionary:

1. First retrieve the name map representing the dictionary.
2. Next, call the name map's `isMember` method to check whether the name map contains an element with the specified name.
3. If the name exists, call the name map's `lookup` method.

The session must be able to obtain a read lock on the name map.

The `lookup` method takes the name as its parameter and returns the object in the name map whose key is the specified name. You should call `lookup` only if `isMember` returns true; `lookup` throws an `ObjyRuntimeException` if you try to look up a name that does not exist in the name map.

EXAMPLE This code fragment retrieves an object of the `Client` class by looking up its name in the application-defined dictionary represented by the name map `nameTable`. The complete method definition appears in the `Sales.Interact` programming example (see page 422).

```
// Static utility to retrieve a client
public static Client lookupClient (String companyName) {
   ooMap nameTable;
   ...
   session.begin();
   ...
   nameTable = ...;
   // Look up name in table
   ...
   if (nameTable.isMember(companyName)) {
      Client client = (Client)nameTable.lookup(companyName);
      session.commit();
      return client;
   }
   else {
      System.out.println("No client named " + companyName);
      session.abort();
      return null;
   }
}
```

Finding Objects in a Graph

An *object graph* is a group of related persistent objects that are linked together in a directed graph data structure. Each link in the graph can be a persistent field that references another object or a relationship. Once you have retrieved one object in the graph (for example, a named root), you can follow links to retrieve the other objects in the graph.

Persistent Fields

When you fetch a retrieved object's data, you retrieve any objects the retrieved object references in its persistent fields. Note however, that the referenced objects are empty; you should fetch their persistent data before you access their persistent fields.

If you define field access methods for every class whose objects are included in an object graph, the process of fetching persistent data and retrieving referenced objects is completely transparent. The access methods that get the values of persistent fields also fetch the data for you, retrieving any referenced objects after obtaining the necessary locks. Remember that you should call a persistent object's field access methods only when the session that owns the object is in a transaction.

EXAMPLE This code fragment first retrieves an object of the `Client` class; the `lookupClient` method starts its own transaction, so that method is called outside a transaction. Once the client is retrieved, a call to its `getSalesRep` field access method gets the client's sales representative, an object of the `Salesperson` class. When `getSalesRep` (shown at the end of the example) calls `fetch`, the client's data (including the empty salesperson object) is retrieved from the database. The complete `printClientRepresentative` method definition appears in the `Sales.Interact` programming example (see page 422).

```
// Static utility to print the name of a client company's
// sales representative
public static void printClientRepresentative(String companyName)
{
   ...
   Client client = Interact.lookupClient(companyName);
   ...
   session.begin();
   // Get the client's sales representative
   Salesperson salesPerson = client.getSalesRep();
   ...
   session.commit();
}
```

```
   public Salesperson getSalesRep() {
      fetch();
      return this.salesRep;
}
```

If a persistent field contains a persistent collection rather than a scalar persistent object, you can retrieve the elements of the collection as described in "Retrieving Elements of a Persistent Collection" on page 229.

Relationships

After you retrieve an object, you must fetch its persistent data to read its relationship field into memory. If you try to access the object's relationship field before you have fetched its data, a `NullPointerException` is thrown.

If you define relationship access methods for each relationship, the process of fetching persistent data and retrieving related objects is completely transparent. The access method for a relationship fetches the object's data, retrieves the related object, and casts the related object to the correct class. Remember that you should call a persistent object's relationship access methods only when the session that owns the object is in a transaction.

To-One Relationships

You can retrieve an object related by a to-one relationship by calling that relationship's `get` method. The `get` method takes no parameters and returns the related object, or null if there is no related object.

EXAMPLE This code fragment first retrieves an object of the `Contact` class; the `lookupContact` method starts its own transaction, so that method is called outside a transaction. Once the contact is retrieved, a call to its `getSalesperson` relationship access method gets the contact's salesperson, an object of the `Salesperson` class. When `getSalesperson` (shown at the end of the example) calls `fetch`, the contact's data, including its `salesperson` relationship, is retrieved from the database. The complete `printSalespersonForContact` method definition appears in the `Sales.Interact` programming example (see page 422).

```
// Static utility to print the name of a contact's salesperson
public static void printSalespersonForContact (...) {
   Contact contact = Interact.lookupContact(...);
   ...
   session.begin();
   // Get the contact's salesperson
   Salesperson salesPerson = contact.getSalesperson();
   ...
   session.commit();
}
```

```
public Salesperson getSalesperson() {
   fetch();
   // Cast retrieved object to class Salesperson
   return (Salesperson)this.salesperson.get();
}
```

To-Many Relationships

You can retrieve objects related by a to-many relationship by calling that relationship's `scan` method. The `scan` methods return an iterator initialized to find the related objects.

All Related Objects

When you want to retrieve all related objects, you should call the relationship's `scan` method with no parameters. The scan method returns an iterator that finds all related objects.

EXAMPLE This code fragment first retrieves an object of the `Salesperson` class; the `lookupSalesperson` method starts its own transaction, so that method is called outside a transaction. Once the salesperson is retrieved, a call to its `getAllContacts` relationship access method (shown at the end of the example) gets an iterator that finds all contacts for the salesperson. The iterator is used to retrieve each of the salesperson's contacts, which are objects of the `Contact` class. The complete `printContactsForSalesperson` method definition appears in the `Sales.Interact` programming example.

```
// Static utility to print the names of a salesperson's contacts
public static void printContactsForSalesperson(int employeeID) {
   ...
   Salesperson salesperson =
         Interact.lookupSalesperson(employeeID);
   ...
   session.begin();
   // Get the salesperson's contacts
   Iterator itr = salesperson.getAllContacts();
   ...
   Contact contact;
   while (itr.hasNext()) {
      contact = (Contact)itr.next(); // Cast to Contact
      contact.printName();
   }
   session.commit();
}
```

```
public Iterator getAllContacts () {
   fetch();
   return this.contacts.scan();
}
```

Related Objects that Satisfy a Condition

When you want to retrieve only those related objects that satisfy some condition, you should call the relationship's `scan` method, passing the condition as a parameter. The condition is a predicate in the Objectivity / DB predicate query language. The `scan` method returns an iterator that finds those related objects satisfying the specified condition.

EXAMPLE In this example, the `findContact` relationship access method of the `Salesperson` class retrieves a particular related contact of a salesperson; the `findContactsByCompany` access method returns an iterator that finds the related contacts in the specified company.

```
// Relationship access methods
...
public Contact findContact(String firstName,
                           String lastName,
                           String company) {
   fetch();
   String predicate = new
            String("company == \"\"" + company + "\" and " +
                   "lastName == \"\"" + lastName + "\" and " +
                   "firstName == \"\"" + firstName + "\"");
   Iterator itr = this.contacts.scan(predicate);
   if (itr.hasNext())
      return (Contact)itr.next(); // Cast to Contact
   else
      return null;
}

public Iterator findContactsByCompany(String company) {
   fetch();
   String predicate = new String("company == \"\""
                                 + company + "\"");
   return this.contacts.scan(predicate);
}
```

Retrieving Elements of a Persistent Collection

Once you have retrieved a persistent collection, you can call its methods to retrieve the elements it contains. Different methods are available for different kinds of collections.

Collection of Objects

Lists, unordered sets, and sorted sets are collections of persistent objects. You can iterate through such a collection, retrieving each object it contains. If the collection is ordered, you can additionally retrieve the element at a particular position within the collection.

Iterating Through the Elements

You can call the `iterator` method of a collection of objects to obtain an iterator for finding all elements of the collection. For loop control, you call the iterator's `hasNext` method to test whether additional elements remain. Within the loop, you make successive calls to the iterator's `next` method to get a local representation of each object in the collection.

You can call the `listIterator` methods of a list to obtain a list iterator for finding its elements. One version of `listIterator` returns a list iterator initialized to start at the first element; the other version returns a list iterator initialized to start at a specified index within the list. A *list iterator* is an instance of a class that implements the `java.util.ListIterator` interface. In addition to the standard iterator methods, a list iterator has additional methods that allow you to reverse the direction of iteration:

- The `previous` method gets the previous element of the list.
- The `hasPrevious` method tests whether a previous element exists.
- The `nextIndex` method gets the index of the next element in the list.
- The `previousIndex` method gets the index of the previous element in the list.

Retrieving Elements by Position

Ordered collections of objects (lists and sorted sets) have methods to retrieve particular elements, identified by their position in the list.

- Call the ordered collection's `first` method to retrieve the first element of the collection.
- Call the ordered collection's `last` method to retrieve the last element of the collection.
- Call the ordered collection's `get` method to retrieve the element at a particular position within the collection. The parameter to `get` is the zero-based index of the element to be retrieved.

Collection of Key-Value Pairs

Name maps, unordered object maps, and sorted object maps are collections of key-value pairs. You can retrieve the keys and values from such a collection.

Retrieving Keys

You can iterate through a collection of key-value pairs, getting each key it contains. If the collection is ordered, you can additionally retrieve the key at a particular position within the collection.

Iterating Through the Keys

To obtain an iterator for finding all the keys of a collection of key-value pairs:

- Call the `keys` method of a name map.
- Call the `keyIterator` method of an object map.

You call the iterator's `next` method repeatedly to get each key in the collection. In the case of a name map, `next` simply returns the key string; in the case of an object map, `next` gets a local representation of the key object.

Retrieving Keys by Position

Sorted object maps have methods to retrieve particular keys, identified by their position in the list.

- Call the ordered map's `first` method to retrieve the key of the first element of the collection.
- Call the ordered map's `last` method to retrieve the key of the last element of the collection.
- Call the ordered map's `get` method to retrieve the key of the element at a particular position within the collection. The parameter to `get` is the zero-based index of the element to be retrieved.

Retrieving Values

You can obtain values from a collection of key-value pairs in either of two ways. First, you can iterate through a collection of key-value pairs, retrieving each value it contains. Second, if you have already obtained a key from the collection, you can retrieve the value associated with that key.

Iterating Through the Values

To obtain an iterator for finding all the values of a collection of key-value pairs:

- Call the `elements` method of a name map.
- Call the `valueIterator` method of an object map.

You call the iterator's `next` method repeatedly to get a local representation of each value in the collection.

Retrieving Values by Key

Once you have obtained a key from a collection of key-value pairs, you can retrieve the object that is the value associated with that key. In the case of a name map, the key is a name. Retrieving the associated value is equivalent to looking up the object by name; see "Name in an Application-Defined Dictionary" on page 224.

In the case of an object map, the key is another persistent object. You call the object map's `get` method, passing the key as the parameter.

The `get` method returns null if the collection does not contain an element with the specified key. To distinguish that situation from the situation in which the collection contains an element with the specified key and a null value, call the `containsKey` method; the parameter to `containsKey` is the key.

Scanning Storage Objects

You can scan any storage object to find its contained objects of a particular persistence-capable class. In particular, you can:

- Scan a container for basic objects of a specified class stored in that container.
- Scan a database for basic objects of a specified class stored in any container within that database.
- Scan a database for containers of a specified class within that database.

  If you need to find *all* containers in a database, you should use the `contains` method, which performs this operation more quickly than the `scan` method. See "Traversing the Storage Hierarchy" on page 234.
- Scan the federated database for basic objects of a specified class stored in any container within any database in the federation.
- Scan the federated database for containers of a specified class within any database in the federation.

To scan a particular storage object, call its `scan` method. Different versions of the `scan` method allow you to scan for all objects of the specified class or to scan for those objects of the class that satisfy a condition. In either case, the `scan` method returns an iterator initialized to find the specified objects.

WARNING Scanning garbage-collectible containers is not reliable because the scan operation may find invalid objects (which are subject to garbage collection) as well as valid ones; applications have no way to test whether an object is valid. If some objects of a given class are stored in garbage-collectible containers, you should not scan for objects of that class in any of the garbage-collectible containers, in any database that contains those containers, or in the federated database. (You could perform such scan operations if you are sure that the federated database contains no garbage objects, for example, in a database maintenance application that is always run immediately after `oogc` has been run.)

All Objects of a Class

When you want to retrieve all objects of a given class in a storage object, you should call the storage object's `scan` method, passing the package-qualified name of the class as the parameter. The session that owns the local representation of the storage object must be in a transaction.

EXAMPLE This code fragment retrieves all objects of the `Salesperson` class from the database `salesDB`. The complete method definition appears in the `Sales.Interact` programming example (see page 422).

```
// Static utility to print the names of all salespeople
public static void printAllSalesPeople() {
   ...
   session.begin();
   // Get sales database
   ooDBObj salesDB = ...;
   Iterator itr = salesDB.scan("Sales.Salesperson");

   Salesperson salesperson;
   while (itr.hasNext()) {
   // Cast to Salesperson
   salesperson = (Salesperson)itr.next();
   salesperson.printName();
   }
   session.commit();
}
```

Objects of a Class that Satisfy a Condition

When you want to retrieve only those objects of a class that satisfy some condition, you should pass both the package-qualified class name and the condition as parameters to the storage object's `scan` method. The session that owns the local representation of the storage object must be in a transaction.

The condition is a predicate in the Objectivity/DB predicate query language. The `scan` method returns an iterator that finds objects of the specified class that satisfy the specified condition. You can make a predicate scan more efficient by defining an index whose key fields are the fields you use in the predicate.

EXAMPLE This code fragment scans a container for clients (of the `Client` class) located in the specified state. The complete method definition appears in the `Sales.Interact` programming example (see page 422).

```
// Static utility to print the clients in the specified state
public static void printClientsInState(String state) {
   ...
   session.begin();
   ooContObj clientCont = ...;

   String predicate = new
         String("state == \"\"" + state + "\"");
   Iterator itr = clientCont.scan("Sales.Client", predicate);

   Client client;
   while (itr.hasNext()) {
      client = (Client)itr.next(); // Cast to Client
      System.out.println(client.getCompanyName());
   }
   session.commit
}
```

Traversing the Storage Hierarchy

Each class of storage objects has a method that allows you to get all contained objects. The session that owns the storage object must be in a transaction when you call one of these methods.

- The `containedDBs` method of the federated database gets the databases in the federated database.
- The `contains` method of a database gets the containers in that database.
- The `contains` method of a container gets the basic objects in that container.

All these methods return an iterator initialized to find the objects contained in the storage object.

EXAMPLE This code fragment illustrates how to traverse the storage hierarchy of a federated database. The complete method definition appears in the `Traversal.Tester` programming example (see page 442).

```
public static void traverse(ooFDObj fd) {
   ooDBObj db;
   ooContObj cont;
   ooObj basicObject;
   Iterator dbItr, contItr, objItr;

   session.begin();
   // Get all databases
   dbItr = fd.containedDBs();
   while (dbItr.hasNext()) {
      db = (ooDBObj)dbItr.next();//Cast to ooDBObj
      // Get all containers in current database
      …
      contItr = db.contains();
      while (contItr.hasNext()) {
         // Cast to ooContObj
         cont = (ooContObj)contItr.next();
         …
         // Get all objects in current container
         objItr = cont.contains();
         while (objItr.hasNext()) {
            // Cast to ooObj
            basicObject = (ooObj)objItr.next();
            …
         }
      }
   }
   session.commit();
}
```

Looking Up an Object by OID

Objectivity / DB identifies each persistent object in a federated database by a unique object identifier (OID). Although OIDs are internal identifiers that application programs usually don't use, the interface allows you to obtain the OID of an object and to retrieve an object by its OID.

To look up an object by its OID, call the `objectFrom` method of the federated database. The session that owns the local representation of the federated database

must be in a transaction. You can specify the OID to be looked up either as an `ooId` object or as a string. The `objectFrom` method allows you to retrieve storage objects (databases and containers) as well as persistent objects.

You might specify the OID as an `ooId` if one session has a local representation of the object and you want another session to get a local representation of the same object. When the first session is in a transaction, you could call the `getOid` method of the object to obtain an `ooId` representing that object's OID. When the second session is in a transaction, you could pass the `ooId` as a parameter to the `objectFrom` method of the second session's federated database; the `objectFrom` method would create a local representation of the object belonging to the second session.

EXAMPLE This code fragment creates a new session and retrieves an object with the specified OID; the retrieved object belongs to the new session. The complete method definition appears in the `Traversal.Tester` programming example (see page 442).

```
// Static utility to retrieve an object by its OID
public static void retrieveAgain(ooId oid) {
   Session session = new Session();
   ooFDObj fd = session.getFD();
   ...
   // Get new local representation of object with specified OID
   session.begin();
   ooObj myBasicObject = (ooObj)fd.objectFrom(oid);
   ...
   session.commit();
}
```

12

Clustering Objects

When a transient object is made persistent, it is assigned a storage location in the federated database. The process of assigning an object to a storage location is called *clustering*. Clustering a basic object assigns it a location in a particular container; clustering a container assigns it a location in a particular database. An object may be clustered *explicitly* or *implicitly*.

This chapter explains how objects are clustered. For guidelines on deciding where to store your persistent objects, see “Assigning Objects to Databases” on page 91 and “Assigning Basic Objects to Containers” on page 99.

In This Chapter

Explicit Clustering

When you cluster an object explicitly, you cause it to be stored in or near a particular *clustering object*, which may be a database, a persistent container, or a persistent basic object. Explicit clustering provides the greatest control over where the object is stored.

Clustering Basic Objects

You explicitly cluster a basic object by calling the `cluster` method of a clustering object, passing the object to be clustered as the parameter.

- Call the `cluster` method of a persistent basic object to store the object as close as possible to that clustering object. The newly persistent object is stored in the same container as the clustering object. If possible, the newly persistent object is stored on the same page as the clustering object or on a page close to that object.
- Call the `cluster` method of a persistent container to store the object in the clustering container.
- Call the `cluster` method of a database to store the object in the default container of the clustering database.

The session that owns the clustering object must be in a transaction, and able to be write-locked, when you call the object's clustering method.

After the successful execution of the `cluster` method, the newly persistent object belongs to the same session as the clustering object.

Clustering Containers

You can explicitly cluster a container in one of two ways. You can add the container to the database where you want to store the container; in this case, the database is the clustering object. Alternatively, you can store the container in or near a particular clustering object. In either case, the session that owns the clustering object must be in a transaction and is able to be write locked.

- To add a container to a database, call the `addContainer` method of the database; parameters specify the container to be added and the container's system name, hash value, initial number of pages, and growth factor.
- To cluster a container with a clustering object, call the `cluster` method of the clustering object, passing the object to be clustered as the parameter.
  - Call the `cluster` method of a database to store the container in the clustering database.
  - Call the `cluster` method of a persistent container or a persistent basic object to store the container in the same database as the clustering object.

  The container will be added to the appropriate database with the default properties: no system name, 5 initial pages, a hash value of 10, and a growth factor of 10%.

After the successful execution of the `addContainer` or `cluster` method, the newly persistent container belongs to the same session as the clustering object.

Implicit Clustering

You can make an object persistent by establishing some kind of association between it and some existing storage object or persistent object. For example, making an object a named root in a database establishes an association between the object and that database; adding an object to a persistent collection establishes an association between the object and the collection.

Whenever an object is made persistent without an explicit call to `cluster` or `addContainer`, the object is clustered implicitly; a particular database, persistent container, or persistent basic object *requests* the clustering operation. The following table lists the actions that can cause implicit clustering and the requesting object for each of those actions.

Action Making the Object Persistent	Object Requesting Clustering
Make the object a named root in a database	The root dictionary of that database
Make the object a named root in the federated database	The root dictionary of the default database of the federated database
Establish a relationship from a persistent object to the object	That persistent object
Name the object in the scope of a persistent basic object	That persistent basic object
Name the object in the scope of a persistent container	That persistent container
Name the object in the scope of a database	That database
Name the object in the scope of the federated database	The default database of the federated database
Add the object to a persistent collection	That persistent collection
Commit or checkpoint a transaction in which you created or modified a persistent object that references the object	The referencing persistent object

Each session has a clustering strategy that performs implicit clustering. A *clustering strategy* is an object of a class that implements the `ClusterStrategy` interface. Objectivity for Java provides one such class, `DefaultClusterStrategy`; you may define your own classes to implement customized clustering strategies for your application.

When you create a session, it is initialized to use the default clustering strategy; that is, its clustering strategy is an instance of `DefaultClusterStrategy`. You can change a session's clustering strategy at any time by calling its `setClusterStrategy` method. You can use the same object as the clustering strategy for several different sessions.

When an object is clustered implicitly, the requesting object determines which clustering strategy is used; Objectivity for Java calls the `requestCluster` method of the session that owns the requesting object. That method, in turn, calls the `requestCluster` method of the session's clustering strategy.

Objectivity for Java calls `requestCluster` whenever an object is made persistent by one of the actions listed in the previous table. You may have additional application-specific reasons for making an object persistent. If so, you can make an object persistent with a direct call to `requestCluster`; the object will be clustered implicitly by the clustering strategy. See "Application-Specific Reasons for Clustering" on page 246.

Default Clustering Strategy

The `requestCluster` method of the default clustering strategy always clusters the object being made persistent with the requesting object.

NOTE If you make a direct call to the `requestCluster` method of a default clustering strategy, your call specifies the requesting object. The requesting object must be a database, a persistent container, or a persistent basic object, and it must belong to the session that is in a transaction when `requestCluster` is called.

Clustering Basic Objects

The following table lists how the default clustering strategy assigns basic objects to containers. When a basic object is clustered in the same container as a persistent basic object, the newly persistent basic object is stored as close as possible to the persistent basic object. If possible, the two objects are stored on the same page.

Action Causing Clustering	Where Basic Object Is Stored
Make the object a named root in a database	The roots container of that database
Make the object a named root in the federated database	The roots container of the default database of the federated database
Establish a relationship from a persistent basic object to the object	The container in which the persistent basic object is stored

Action Causing Clustering	Where Basic Object Is Stored
Establish a relationship from a persistent container[1] to the object	That container
Name the object in the scope of a persistent basic object	The container in which the persistent basic object is stored
Name the object in the scope of a persistent container	That container
Name the object in the scope of a database	The default container of that database
Name the object in the scope of the federated database	The default container of the default database of the federated database
Add the object to a persistent collection	The container in which that collection is stored
Commit or checkpoint a transaction in which you created or modified a persistent basic object that references the object	The container in which the referencing basic object is stored
Commit or checkpoint a transaction in which you created or modified a persistent container[2] that references the object	The referencing container
Call `requestCluster` directly with a database as the requesting object	The default container of the requesting database
Call `requestCluster` directly with a persistent container as the requesting object	The requesting container
Call `requestCluster` directly with a persistent basic object as the requesting object	The container in which the requesting object is stored

1. Only containers of an application-defined class can have relationships.
2. Only containers of an application-defined class can have persistent fields that reference other persistent objects.

Clustering Containers

The following table lists how the default clustering strategy assigns containers to databases. In this table, a "persistent object" means either a persistent basic object or a persistent container.

Action Causing Clustering	Where Container Is Stored
Make the container a named root in a database	That database
Make the container a named root in the federated database	The default database of the federated database
Establish a relationship from a persistent object to the container	The database in which the persistent object is stored
Name the container in the scope of a persistent object	The database in which the persistent object is stored
Name the container in the scope of a database	That database
Name the container in the scope of the federated database	The default database of the federated database
Add the container to a persistent collection	The database in which that persistent collection is stored
Commit or checkpoint a transaction in which you created or modified a persistent object that references the container	The database in which the referencing persistent object is stored
Call `requestCluster` directly with a database as the requesting object	The requesting database
Call `requestCluster` directly with a persistent object as the requesting object	The database in which the requesting object is stored

Defining a Clustering Strategy

You can define your own clustering strategy to change the default location where objects are implicitly clustered and to handle application-specific reasons for making an object persistent.

You define your own clustering strategy with a class that implements the `ClusterStrategy` interface. You must define a customized `requestCluster` method for your class. Parameters to `requestCluster` specify:

- The requesting object.

  The requesting object is a database, a persistent container, or a persistent basic object; it belongs to the session that is in a transaction when `requestCluster` is called.

- The clustering reason, which specifies why the object is being made persistent.

  A *clustering reason* is an object of a class that implements the `ClusterReason` interface. Your `requestCluster` method can call the `getReason` method of the clustering reason to find out why the object is being made persistent. That method returns one of the following constants, defined in the `ClusterReason` interface:

`BIND`	The object was made a named root.
`NAMEOBJ`	The object was given a scope name.
`RELATIONSHIP`	A relationship was established to the object.
`COLLECTION`	The object was added to a persistent collection.
`REFERENCE`	The object was referenced from an object being written to the database.
`APPLICATION`	The application made a direct call to `requestCluster`.

- The object to be clustered.

Your `requestCluster` method should use the information passed in its parameters to select a clustering object; it should then call the clustering object's `cluster` method to store the object being clustered in or near the clustering object. The clustering object must be a persistent basic object, a persistent container, or a database; it must belong to the same session as the requesting object.

Your `requestCluster` method is responsible for ensuring that the object is, in fact, made persistent and that the newly persistent object belongs to the correct session. An exception will be thrown in the following circumstances:

- The object is still transient after `requestCluster` has been executed. This situation arises when `requestCluster` does not pass the object to a `cluster` method.
- The object has been made persistent but belongs to the wrong session. This situation arises when `requestCluster` selects a clustering object that does not belong to the same session the requesting object belongs to.

In many cases, the referencing object really decides how to cluster the objects (see the `DefaultClusterStrategy`) and there are a number of different cluster

strategies that can be designed. For example, you can choose to cluster objects based on their type rather than who is referencing them: putting all instances of `SalesMan` into one container and all `Contacts` into another container. You can also use round-robin or random order cluster strategies to distribute objects to a set of containers. Another cluster strategy can be employed to fill containers up to some limit and then switch to a newly created container, and so on.

It is important to delegate clustering to these strategy objects. It enables you to radically alter and manage your clustering and concurrency issues by simply changing the strategy object and leaving all other application code "as is". The strategy object is dynamic, and can be changed at any time—even in mid-transaction.

EXAMPLE This example shows the `requestCluster` method of the `ContainerPoolStrategy` class (see page 453). This strategy clusters any container in the database of the requesting object; it uses the clustering reason to decide where to cluster a basic object. The strategy stores named roots and the objects in their object graphs in garbage-collectible containers; it stores scope-named objects and the objects in their object graphs in non-garbage-collectible containers. Every database has two container pools, one of garbage-collectible containers and one of non-garbage-collectible containers. When the strategy clusters a named root or scope-named object, it uses the appropriate container pool in the database of the requesting object. A container pool is an object of the `ContainerPool` class (see page 450). The `clusterObject` method of a container pool (shown at the end of the following example) selects a container at random from the pool and clusters the specified object in that container.

```
public void requestCluster(Object requestObject,
                           ClusterReason reason,
                           Object object) {
   ooDBObj db;
   String poolName;

   // Set db to the database of the requesting object
   if (requestObject instanceof ooDBObj)
      db = (ooDBObj)requestObject;
   else if (requestObject instanceof ooContObj)
      db = ((ooContObj)requestObject).getDB();
   else if (requestObject instanceof ooObj)
      db =
         (((ooObj)requestObject).getContainer()).getDB();
   else
      throw new
         ClusteringException("Illegal requesting object.");
```

```
        if (object instanceof ooContObj) {
            // Cluster a container in the database of the
            // requesting object
            db.cluster(object);
            return;
        }

        int reasonCode = reason.getReason();
        if ((reasonCode == ClusterReason.RELATIONSHIP) ||
            (reasonCode == ClusterReason.COLLECTION) ||
            (reasonCode == ClusterReason.REFERENCE) ||
            (reasonCode == ClusterReason.APPLICATION)) {
            // Cluster with requesting object to keep entire
            // object graph in same kind of container
            ((ooObj)requestObject).cluster(object);
            return;
        }

        // Get name of the appropriate container pool
        if (reasonCode == ClusterReason.BIND) {
            // Cluster named root in a garbage-collectible container
            poolName = "GC Container Pool";
        }
        else if (reasonCode == ClusterReason.NAMEOBJ) {
            // Cluster scope-named object in a
            // non-garbage-collectible container
            poolName = "Non-GC Container Pool";
        }
        else
            throw
                new ClusteringException("Illegal clustering reason.");

        // Retrieve the container pool
        ContainerPool containerPool =
                        (ContainerPool)(db.lookupObj(poolName));

        // Cluster object in a container chosen at random from
        // the selected pool
        containerPool.clusterObject(object);
    }
```

```
// Utility method to cluster the specified object with a
// container chosen at random from the containers in this
// container pool
public void clusterObject(Object object) {
   // Randomly select a container from this pool
   int index = Math.abs((new Random()).nextInt()) %
               this.getNumberOfContainers();
   ooContObj container = this.getContainer(index);

   // Cluster the object with the selected container
   container.cluster(object);
}
```

Application-Specific Reasons for Clustering

You can define one or more application-specific reasons for making objects persistent and implement clustering strategies that take these application-specific reasons into account. You must define a clustering reason class corresponding to a new clustering reason; to make an object persistent for that reason, you make a direct call to `requestCluster`, using an object of your new reason class as the clustering reason.

Defining a Clustering Reason

You define an application-specific clustering reason with a class that implements the `ClusterReason` interface. Define the `getReason` method of your class to return the value `ClusterReason.APPLICATION`, which indicates an application-specific reason for clustering the object. If objects of your class will be used with an application-specific clustering strategy, your class may include whatever additional fields and methods are needed by the clustering strategy.

EXAMPLE This example shows the `JustCreatedReason`. A clustering reason of this class indicates that an object is being made persistent, because all objects of its class are made persistent as soon as they are created. This class defines only the `getReason` method; more complicated clustering reasons could have additional fields and methods to be used by a clustering strategy.

```
public class JustCreatedReason implements ClusterReason {
   public int getReason() {
      return ClusterReason.APPLICATION;
   }
}
```

Using an Application-Specific Reason in a Clustering Strategy

If you define an application-specific clustering reason, you can also implement one or more clustering strategies that recognize the new clustering reason. Your clustering strategy may use the same logic for all application-specific reasons, or it may use different logic depending on the class of the clustering reason.

NOTE The default clustering strategy always clusters an object with the requesting object, independent of the clustering reason.

EXAMPLE This code fragment is taken from the `requestCluster` method of the `ClusterByClassStrategy` class (see page 455). This strategy recognizes the `JustCreatedReason` clustering reason (see page 460) and selects a clustering object based on the class of the newly created object. It clusters a newly created object of the `Account` class in a garbage-collectible container selected from the container pool of the database named `Accounting`; it clusters a newly created object of the `Employee` class in a non-garbage-collectible container selected from the container pool of the database named `Staff`.

```
public void requestCluster(Object requestObject,
                           ClusterReason reason,
                           Object object) {
   ooDBObj requestingDB = …;
   …
   int reasonCode = reason.getReason();
   if ((reasonCode == ClusterReason.APPLICATION) &&
      (reason instanceof JustCreatedReason)) {
      // Make object persistent because all objects of its
      // class are made persistent when they are created
      if (object instanceof Account) {
         // Cluster in a container from the pool of
         // garbage-collectible containers in the database
         // named "Accounting"
         …
      }
      else if (object instanceof Employee) {
         // Cluster in a container from the pool of
         // non-garbage-collectible containers in the
         // database named "Sales"
         …
      }
```

```
        else
            throw new ClusteringException("Unrecognized class.");
            // ClusteringException is an application-defined
            // exception
    }
    …
}
```

Making a Direct Call to requestCluster

If you have defined an application-specific clustering reason, you can make an object persistent with a direct call to `requestCluster` of a session or a clustering strategy. The parameters to `requestCluster` should be the requesting object, an object of your clustering reason class, and the object to be made persistent.

You typically call the `requestCluster` method of the session that is in a transaction.

EXAMPLE In this example, the constructor for the `Account` class (see page 460) makes the newly created object persistent by calling the `requestCluster` method of the current session.

```
// Constructor makes an object persistent; the branch office
// requests it to be clustered.
public Account(BranchOffice branch) {
    …
    // Cluster the new object using the clustering strategy of
    // the session that owns branch
    Session session = branch.getSession();
    session.requestCluster(branch, reason, this);
}
```

You may call the `requestCluster` method of *any* clustering strategy, not just the clustering strategy of the current session. Remember, however, that you can make an object persistent only while a session is in a transaction and the object must be clustered with a basic object, container, or database belonging to that session.

EXAMPLE In this example, the constructor for the `Employee` class (see page 462) makes the newly created object persistent by calling the `requestCluster` method of a clustering strategy of the `ClusterByClassStrategy` class (see page 455). The clustering strategy is not associated with any session.

```
// Constructor makes an object persistent; the branch office
// requests it to be clustered.
public Employee(BranchOffice branch, String name) {
   ...
   // Cluster the new object using the ClusterByClassStrategy
   // clustering strategy
   ClusterByClassStrategy strategy =
                              new ClusterByClassStrategy();
   strategy.requestCluster(branch, reason, this);
}
```

13

Optimizing Searches With Indexes

If your application scans a storage object to find the persistent objects that satisfy a particular predicate, you can define an *index* to speed the search.

In This Chapter

Indexes

An *index* is a data structure that maintains references to the objects of a particular class within a particular storage object. An index sorts objects according to the values in one or more of their fields, called the *key fields* of the index. As a consequence, a scan operation whose predicate tests the objects' key fields can use the index to find the desired objects quickly.

The decision to create an index involves a trade-off of the runtime efficiency of predicate scans against the runtime cost to create and update the index and the storage space required to store it. If the indexed objects seldom change, the maintenance cost is negligible. However, if objects of the indexed class are frequently added, deleted, or modified, the maintenance cost of updating the index may be significant.

Indexes are created dynamically by application programs and continue to exist until they are explicitly removed by application programs. As long as an index

exists, it is available to be used by scan operations on the relevant storage object with predicates that test the key fields of objects of the indexed class. Multiple clients can read a given index concurrently; however, only one client at a time can modify the index or its objects. If one client is modifying an index, other clients can read the index in MROW sessions.

By default, if an index exists that is relevant to a particular scan operation, the index is used automatically. If you do not want indexes to be used during one or more scan operations, you can disable indexes during those operations and, if desired, re-enable them afterward.

Indexes can be long-lived or short-lived. At one extreme, developers may anticipate that certain predicate scans will be used frequently to search for objects that seldom change. In that case, after the objects are created, a simple application could be run to create indexes that optimize the expected scans. Those indexes might never be removed. At the other extreme, an inventory report application that runs once a year might repeatedly scan using predicates that test the same combination of fields. An index on those fields would improve the performance of the scans, but would not be needed by other applications that run more frequently to modify the inventory. The inventory report application could create the necessary index, perform its various predicate scans, then delete the index.

Key Fields

The key fields of an index must be persistent fields of the following Java types:

- A primitive type: `byte`, `short`, `int`, `long`, `float`, `double`, `boolean`, `char`.
- A string class: `String`, `StringBuffer`.

Persistent fields of the following types *cannot* be used as key fields:

- Date and time types
- Persistence-capable classes
- Relationships

Indexes can optimize tests that compare a key field with a literal numeric or string value. If you define a key field of type `boolean`, an optimized predicate must use an *integer* literal (1 for true, 0 for false).

An index sorts objects according to the values in their key fields; the key fields are considered in the order specified when the index is created. For example, suppose objects of the `Person` class are stored in a given container and you create an index for `Person` on that container with the key fields `name` and `age`. The index sorts objects by their name first and their age second; if two or more objects have the same name, the one with the lowest age comes first in the indexed order.

Optimized Scan Operations

An index defined on a particular storage object is used to optimize predicate scans of that storage object for objects of the indexed class. The predicate used in the scan must be one of the following:

- A single optimized condition that tests the first key field of the index.
- A conjunction of conditions in which the first conjunct is an optimized condition that tests the first key fields of the index.

An *optimized condition* is a condition of one of the following forms:

Optimized Condition	Notes
keyField = constant *keyField == constant* *keyField > constant* *keyField < constant* *keyField >= constant* *keyField <= constant*	*keyField* is a key field of the index; its type is a numeric primitive type or a string type. *constant* is a constant of the same type as the *keyField*.
stringKeyField =~ stringConstant	*stringKeyField* is a key field of the index; its type is a string class. *stringConstant* is a string constant that begins with a non-wildcard character.

The following table illustrates which predicate scans are optimized by indexes. In each row, the first column lists a predicate; the second column lists the key fields of the index; the third column indicates whether the index is used to optimized the predicate scan.

Predicate	Key Fields	Index Used?
`age = 40`	`age`	Yes; predicate is optimized condition.
`weight > 100`	`age`	No; `weight` is not a key field.
`age != 40`	`age`	No; test for inequality is not an optimized condition.
`(age > 40) && (age < 60)`	`age`	Yes; starting with an optimized conditions that tests the first key field.
`(age > 40) && (age != 60)`	`age`	Yes; starting with an optimized conditions that tests the first key field.
`(age > 40) OR (age < 60)`	`age`	No; predicate is a disjunction.

Predicate	Key Fields	Index Used?
`height > 60`	`weight, height`	No; predicate doesn't test first key field (`weight`).
`(height > 60) && (weight >& 100)`	`weight, height`	No; first condition doesn't test first key field (`weight`).
`name =~ "Me.er"`	`name`	Yes; pattern begins with non-wildcard character `"M"`.
`name =~ ".*son"`	`name`	No; pattern begins with wildcard character `"."`.

If the predicate is a conjunction of optimized conditions that test the first *n* key fields of the index in the correct order, where *n* is an integer greater than one and less than or equal to the number of key fields, the index optimizes search for the objects that satisfy those conditions. For example, if the key fields of an index are `age`, `weight`, and `height`, the index optimizes search for objects that satisfy the following conditions:

```
age > 40 and weight > 100
age > 40 and weight > 100 and height > 60
```

If the first *n* conditions of a predicate test the first *n* key fields in the correct order, and the predicate contains additional conditions, those additional conditions are tested after the index has found objects satisfying conditions on its first *n* key fields. For example, suppose the key fields of an index are `age`, `weight`, and `height` and a scan uses the following predicate:

```
age > 40 and weight > 100 and salary > 40000 and height > 60
```

The first two conditions test the first two key fields, so the index optimizes the search for objects whose age and weight are in the specified ranges. Then, each of those objects is tested to see whether its salary and its height are in the specified ranges.

Similarly, suppose a scan used the following condition:

```
age > 40 and height > 60
```

The first condition tests the first key field. Because `height` is the third key field, and the predicate does not test the second key field (`weight`), only the condition on `age` is optimized. The index optimizes the search for objects whose age is greater than 40; then, each of those objects is tested to see whether its height is greater than 60.

Creating an Index

You create an index for objects of a particular class contained in the federated database, in a particular database, or in a particular container. To create an index and operate on it, you call methods of the storage object in which the indexed objects are located; the session that owns the storage object must be in a transaction. You typically create indexes in a separate transaction, not intermixed with other persistent operations.

You can create unique indexes and nonunique indexes. Every object indexed by a *unique index* must have a unique combination of values in its key fields. Nonunique indexes do not place this restriction on the indexed objects.

NOTE You are responsible for ensuring that every object indexed by a unique index has a unique combination of values in its key fields. If two or more objects have a given combination of key values, the index will contain only the first such object that is encountered when the index is created or updated.

You create a nonunique index for a storage object with its `addIndex` method.

You create a unique index with the storage object's `addUniqueIndex` method.

When you create an index, you specify:

- The name of the index.

  Each index of the federated database must have a unique name. Each index of a given database and each index of any container in that database must have a unique name. For example, if a particular database has an index named "`By Name`" on the class `Person`, it cannot have an index named "`By Name`" on the class `Company` and no container in the database can have an index named "`By Name`". However, a different database may have an index named "`By Name`" and the federated database may have an index named "`By Name`".
- The persistence-capable class whose objects are to be sorted by the index. Note that you must specify a class and not an interface.
- A list of key fields separated by commas. The order of the key fields within the list determines the sorting order of the indexed objects.

EXAMPLE This code fragment creates a non-unique index named `ByLocation` for the container `clientCont`. The index sorts objects of the `Client` class by the key fields `state`, `city`, and `zipCode`. Thus, all clients in Alabama are sorted before clients in other states. Among the Alabama clients, those in Birmingham are sorted before those in Montgomery. Among the Birmingham clients, those with zip code `35204` are sorted before those with zip code `35205`. The complete

method definition appears in the `Sales.Interact` programming example (see page 422).

```
package Sales;
// Static utility to add a client
public static Client addClient (...) {
   ooContObj clientCont;
   ...
   clientCont = ...;
   // Create an index of clients by geographical location
   clientCont.addIndex(
                  "ByLocation", // Name of new index
                  "Sales.Client", // Class of indexed objects
                  "state, city, zipCode");  // Key fields
   ...
}
```

A field of any of a class' superclasses can be a key field. If the key field of the superclass has the same name as a field of its subclass or is not visible due to access control, the key field must be qualified by the name of the superclass using the following syntax:

packageQualifiedClassname::*keyfieldname*

NOTE It is possible for a superclass field to be of a different type from the type of the field in the class on which we want to declare the index.

EXAMPLE This example shows class definitions for `Vehicle` and `Truck` classes. If you want to use a vehicle's `capacity` field as a key field for the index of `Truck` objects, the required specification is:

```
   RentalFleetVehicle::capacity.
```

```
package RentalFleet;
public class Vehicle extends ooObj {
   int capacity;
   ...
}
```

```
public class Truck extends Vehicle {
   int capacity;
   …
}
```

Working With an Index

Once you have defined indexes for a storage object, you can perform the following operations:

- Test whether the storage object has the specified index with the `hasIndex` method.
- Delete the specified index from the storage object with the `dropIndex` method.

Updating Indexes

An index sorts the indexed objects and uses its ordering to determine which objects satisfy a particular predicate scan. In order for the scan operation to be performed correctly, the index must contain all objects for which the predicate is to be tested and no other objects; furthermore, those objects must be ordered correctly in the index. Immediately after an index is created, it contains the correct objects in the correct order. However, when an object of the indexed class is created, it must be inserted into the index; when an existing object is deleted, it must be removed from the index; when the value of an object's key field is modified, the object must be moved to the appropriate spot in the sorting order of the index.

You can control when indexes are updated, relative to when indexed objects are modified. A session's index mode controls when indexes are updated while that session is in a transaction. You can set a session's index mode by calling its `setIndexMode` method; you can get a session's current index mode by calling its

getIndexMode method. Index modes are specified by the following constants, defined in the oo interface:

Index Mode	Meaning
INSENSITIVE	When the transaction is committed, indexes are updated automatically. This is the default index mode.
SENSITIVE	Indexes are updated automatically during the transaction immediately after an indexed object is deleted or an object of the indexed class is made persistent. Immediate updates ensure that you do not have to wait until commit for changes to be reflected in the index.
EXPLICIT_UPDATE	The application must update indexes manually by explicit calls to the updateIndexes method of each new object or modified indexed object.

Disabling and Enabling Indexes

The index usage policy of a session controls whether indexes are used when performing predicate scans. The use of indexes is disabled by default. You can enable and later disable the use of indexes by a particular session; to do so, you call the setUseIndex method of the session object.

- When indexes are disabled, no scan operations will be optimized even if a relevant index exists.
- When indexes are enabled, any index you define is used whenever it can optimize a predicate scan, as described in “Optimized Scan Operations” on page 253.

Disabling the use of indexes may be desirable in either of the following circumstances:

- You are scanning for objects with values in the entire range of the key fields and sorting is not necessary. In such a case, indexes do not speed up the query.
- You are scanning for objects of a particular class and you know that some objects of that class have been created or modified since the last time indexes were updated.

14

Schema Management

An Objectivity / DB federated database has a *schema* that describes every class whose objects are saved in the federated database. The schema is shared by all applications that access the federated database. The schema description for a class specifies the name of the class and the data type of each persistent field. The schema uses class names and field data types that are independent of the application source language. This language-independent representation allows applications written in Java, C++, and Smalltalk to read and write persistent objects in the same federated database.

When a Java application writes an object to the federated database, the object's persistent data in memory is mapped from Java data types to the corresponding Objectivity / DB data types specified in the schema. When a Java application reads an object, the object's persistent data in the federated database is mapped from the Objectivity / DB data types to the corresponding types declared in the Java class.

This chapter introduces schema policies, explains how Java applications add class descriptions to the schema, and describes the language-independent class name and field data types used in a class description in the schema.

In This Chapter

Schema Policies

Each language has its own mechanism for adding class descriptions to the schema. In the case of Java, descriptions are added dynamically by a running application. The schema policy of the connection determines whether descriptions can be added to the schema. If so, class descriptions can be added automatically by Objectivity for Java or explicitly by the application.

A *schema policy* is an object of a class that implements the `SchemaPolicy` interface. Every connection has an associated schema policy that controls what kinds of modifications the application can make to the schema. Properties of the schema policy specify whether:

- New class descriptions can be added to the schema.
- Existing class descriptions in the schema can be modified. (Modifications to class descriptions are discussed in Chapter 15, “Schema Evolution and Object Conversion”.)
- A field’s access control is considered in testing whether its description is the same in the schema and the Java class declaration.
- Informational messages are printed.
- The policy is locked, preventing its properties from being changed.

When you open a connection, the connection’s schema policy is unlocked and it allows your application to add class descriptions to the schema and to change existing descriptions. The schema policy requires a field’s access control to be the same in the schema and the Java class declaration. Informational messages related to the schema are printed. If you want to change the schema policy in any way, you must call methods of the schema policy.

To obtain the schema policy that governs your application, you call the `getSchemaPolicy` method of the connection object. You can call methods of the schema policy to get and set its properties.

During the prototyping or development phases of a project, you should not need to change the schema policy. However, you may want to modify the schema policy in a deployed application.

- You may want to prohibit deployed applications from modifying the class descriptions in the schema.
- If you initialize the schema of an empty federated database for delivery to installation sites, you may want to prohibit your deployed application from adding class descriptions to the schema.
- If your application needs to interoperate with applications written in C++ or Smalltalk, you may need to prohibit applications from modifying the class descriptions in the schema; modifications made by a Java application could

result in a class description with Objectivity / DB data types that are not supported by C++ or Smalltalk applications.

- If your application needs to interoperate with applications written in C++ or Smalltalk, you may need to allow the access control of fields in the schema to be different from the access control in the Java class declarations. For example, if a class description in the schema was created by a C++ program, the access control of the fields will be the access control specified by the C++ program; you may need to use different access control for corresponding fields of your Java class.
- You may prefer to print only error messages related to the schema in a deployed application.
- You may prefer for the schema policy to be locked in a deployed application. Once the policy is locked, it cannot be unlocked or modified in any way.

EXAMPLE This code fragment sets the schema policy in a deployed application immediately after the application opens a connection to the federated database.

```
...
Connection connection = Connection.open(...);
SchemaPolicy policy = connection.getSchemaPolicy();
// Prohibit addition of new class descriptions
policy.setCreateClassAllowed(false);
// Prohibit modification of existing class descriptions
policy.setChangeClassAllowed(false);
// Allow access control to be different in schema and Java class
policy.setFieldAccessControlEnforced(false);
// Use terse messages
policy.setVerbose(false);
// Lock the schema policy
policy.setPolicyLocked();
```

Adding Class Descriptions to the Schema

Class descriptions can be added to the schema in either of two ways: automatically as they are needed, or under explicit control of the application. If informational messages are enabled, the schema manager will print a message whenever a new class description is added to the schema.

The class description will contain the application's schema class name for the class and field descriptions corresponding to the persistent fields of the Java class. For details, see "Content of a Schema Class Description" on page 263.

NOTE If you want to use a custom schema class name for any of your classes, you must set the class name *before* you perform any operations that cause class descriptions to be added to the schema.

Adding Descriptions Automatically

A Java application does not need to take any explicit action to add class descriptions to the schema; they are added automatically when they are needed. Objectivity for Java needs to have a schema description for a particular persistence-capable class, `PCclass`, when:

- An instance of `PCclass` is made persistent.
- The application scans a storage object for persistent objects of the class `PCclass`.
- The description of a subclass of `PCclass` is added to the schema.
- The description of another class is added to the schema and that class references `PCclass` in a persistent field.
- The description of another class is added to the schema and that class has a relationship in which `PCclass` is the related class.
- An index is defined for objects of `PCclass`.
- A subclass of PCclass is added to the schema for any of the above reasons.

When Objectivity for Java needs a class description, it searches the schema for the application's schema class name for the class. If it finds the name, it uses the class description associated with that name. If not, it automatically adds a new class description to the schema.

Adding Descriptions Explicitly

You can add class descriptions to the schema explicitly before any objects of the class are made persistent. For example, you might initialize the schema of an empty federated database to contain class descriptions of all persistence-capable classes used by an application. You could then deliver the empty federated database to the various installation sites when you deploy the application. When the deployed application adds persistent objects to the federated database, their class descriptions will already exist in the schema; you may want to disable the deployed application from adding more class descriptions to the schema, as described in "Schema Class Names" on page 264.

To add the description of a persistence-capable class to the schema, call the `registerClass` method of the connection object. The parameter to `registerClass` is the package-qualified name of the Java class whose description is to be added. This method adds to the schema the descriptions of the specified

class, its superclass, and all its other ancestor classes. If the class has persistent fields that reference other persistence-capable classes, `registerClass` adds their descriptions recursively.

EXAMPLE This code fragment adds descriptions of four classes to the schema: `Truck` (whose Java class name is the parameter to `registerClass`), `Vehicle` (the superclass of `Truck`), `RentalFleet` (which is referenced from the `fleet` field of `Vehicle`), and `Fleet` (the superclass of `RentalFleet`).

```
...
Connection connection = Connection.open(...);
connection.registerClass("RentalFleet.Vehicles.Truck");
```

```
package RentalFleet.Vehicles;
   ...
public class Truck extends Vehicle {
   ...
}
```

```
public class Vehicle extends ooObj {
   // Persistent fields
   RentalFleet fleet;
   ...
}
```

```
public class RentalFleet extends Fleet {
   ...
}
```

```
public class Fleet extends ooObj {
   ...
}
```

Content of a Schema Class Description

When a Java application adds a class description to the schema, either automatically or with a direct call to `registerClass`, the new class description contains a class name, the name of the class's superclass, an ordered collection of field descriptions, and an ordered collection of relationship descriptions.

- The class name in the schema is the application's schema class name for the class at the time when the class description was added to the schema.
- The name of the superclass in the schema is the application's schema class name for the Java superclass.

- The fields described in the schema are the persistent fields of the Java class, in the same order as they appear in the Java class declaration. Each field description specifies the field's name, data type, and access control. The field's:
  - Name in the schema is the same as the Java field name.
  - Objectivity / DB data type in the schema is the default mapping for the Java type of the field.
  - Access control (public, protected, or private) in the schema is the same as the access control of the Java field. Java fields with a default access control are mapped to private in the schema.
- The relationships described in the schema are the relationships of the Java class, in the same order as the corresponding relationship fields appear in the Java class declaration. Each relationship description specifies the same information as the relationship definition method of the Java class. The:
  - Relationship name in the schema is the same as the Java relationship name.
  - Name of the related class in the schema is the application's schema class name for the related Java class.
  - Name of the inverse relationship (if any) is the same in the schema as in Java.
  - Relationship's directionality and cardinality, its delete and lock propagation behavior, and its copying and versioning behavior are all the same in the schema as in Java.

Schema Class Names

The schema identifies each class with a unique class name that is set when the class description is added to the schema.

NOTE A class name in the schema can contain a maximum of 487 characters.

An Objectivity for Java application associates a *schema class name* with each persistence-capable class whose objects it read or writes. The schema class name is the name of the corresponding class in the federated database schema.

NOTE Every application that reads or writes persistent objects of a given class must use the same schema class name (even if they use a different class name internally in their source code).

If two applications use different schema class names for the same class, the schema will contain two descriptions of the class, one with the schema class name used by the first application and one with the schema class name used by the second application. The federated database will consider the two classes to be distinct.

Whenever a Java application reads or writes a persistent object, Objectivity for Java maps between the name of the object's persistence-capable Java class and the schema class name for that class. The default mapping creates a schema class name from the package-qualified name of a Java class. An application can override the default for any class by registering a *custom schema class name* for that class.

Default Schema Class Name

Objectivity for Java creates a default schema class name from the package-qualified name of the class, using the underscore character (_) as the separator between package names and the unqualified class name. For example, the package-qualified class name of the Java class `Truck` in the package `Vehicles` within the package `RentalFleet` is:

```
RentalFleet.Vehicles.Truck
```

That `Truck` class has the default schema class name:

```
RentalFleet_Vehicles_Truck
```

Default schema class names are appropriate for most applications that interoperate only with other Java applications. However, you should register custom schema class names under the following circumstances:

- If your application interacts with applications written in different languages, you may want to use a custom schema class name for any persistence-capable class that your application adds to the schema.
- If your application accesses persistent objects of a class that is already in the schema, it needs to use the existing schema class name for that class. If the existing name is different from the default, you must register the existing name as the custom schema class name for the class. This situation arises when the class was added to the schema by a C++ or Smalltalk application or by a Java application that registered a custom schema class name.
- If any persistence-capable class is nested deeply in packages with long names, so that its default schema class name exceeds the 487-character limit, you should define a custom schema class name.

Custom Schema Class Name

When you register a custom schema class name for a class, you instruct Objectivity for Java to map the Java class name to the custom schema class name

instead of to the default schema class name. The mapping remains in effect throughout the particular connection in which you register the custom schema class name.

The default schema class names are unnatural to C++ and Smalltalk applications because the Objectivity / DB interfaces for those languages do not have the notion of qualified name spaces analogous to Java's packages. If your application interoperates with applications written in different source languages, you can facilitate interoperability by using a custom schema class name for the persistence-capable classes that your application adds to the schema.

To register a custom schema class name for a persistence-capable class, call the `setSchemaClassName` method of the connection object, specifying the Java class and its custom schema class name.

NOTE You should set all custom schema class names for your application *before* making calls to `registerClass` or performing any operation that would cause class descriptions to be added to the schema automatically.

The `setSchemaClassName` method defines the application's mapping between the Java class name and the schema class name; it does not directly modify the schema. The mapping remains in effect only during the current connection; it is never stored explicitly in the federated database.

EXAMPLE This code fragment registers a custom schema class name for the Java class `RentalFleet.Vehicles.Truck`, overriding the default name (`RentalFleet_Vehicles_Truck`) with the custom name `Truck`.

```
Connection connection = Connection.open(...);
connection.setSchemaClassName(
// Package-qualified name of class
      "RentalFleet.Vehicles.Truck",
// Custom schema class name
      "Truck");
```

Remember that all applications must use the same schema class name for a given class. For example, an application that uses the custom name `Truck` will read and write objects identified in the federated database as `Truck` objects. If a second application uses the default name `RentalFleet_Vehicles_Truck`, it will read and write a completely different set of objects, which are identified in the federated database as `RentalFleet_Vehicles_Truck` objects.

Default Mapping for Java Types

Objectivity / DB supports a wide range of data types for persistent data. A given Objectivity / DB type may be mapped to more than one Java data type. A given Java data type may be mapped to more than one Objectivity / DB data type. However, each Java type has a single default, preferred mapping.

When a Java application adds a class description to the schema, the Objectivity / DB data type of each persistent field in the class description is set to the default mapping for the field's Java data type.

The tables in the following subsections give the default mappings of Java types to Objectivity / DB types. For a description of the Objectivity / DB field data types, see Chapter 21, "Objectivity / DB Data Types".

Java Primitive Types

The following table shows the default mapping of Java primitive types to Objectivity / DB types. Note that a Java primitive type is always mapped to a Objectivity / DB primitive type (never to an Objectivity / DB class).

Java Primitive Type	Default Objectivity/DB Type
`byte`	`int8`
`short`	`int16`
`int`	`int32`
`long`	`int64`
`boolean`	`uint8`
`char`	`uint16`
`float`	`float32`
`double`	`float64`

Java Classes

The following table shows the default mapping of Java classes to Objectivity / DB types. With the exception of the two Java string classes, the default representation for a Java class is an object reference to an internal persistent object of an internal Objectivity / DB class. As a consequence, object identity is usually preserved when data is transferred between a Java application and a database.

Java Class	Default Objectivity/DB Type
`String`	`ooUTF8String`
`StringBuffer`	`ooUTF8String`
`java.util.Date`	`ooRef(oojDate)`
`java.sql.Date`	`ooRef(oojDate)`
`java.sql.Time`	`ooRef(oojTime)`
`java.sql.Timestamp`	`ooRef(oojTimestamp)`
`AppClass`, where *`AppClass`* is an application-defined persistence-capable class whose schema class name is *`PCclass`*.	`ooRef(`*`PCclass`*`)`
`APIclass`, where *`APIclass`* is a persistence-capable class defined in the public Objectivity for Java programming interface (such as `ooContObj` or `ooMap`).	`ooRef(`*`APIclass`*`)`
`PCinterface` where *`PCinterface`* is an interface (implemented by one or more persistence-capable classes).	`ooRef(ooObj)`

NOTE Object identity of values in `String` and `StringBuffer` fields is *not* preserved. For example, suppose that two Java objects reference the same `String` object in persistent fields. If the two objects are saved in the database and later read by a Java application, the two retrieved objects will reference different `String` objects.

Java Array Types

The following table shows the default mapping of Java arrays to Objectivity / DB types.

Java Array Type	Default Objectivity/DB Type
`byte[]`	`ooRef(oojArrayOfInt8)`
`short[]`	`ooRef(oojArrayOfInt16)`
`int[]`	`ooRef(oojArrayOfInt32)`
`long[]`	`ooRef(oojArrayOfInt64)`
`boolean[]`	`ooRef(oojArrayOfBoolean)`
`char[]`	`ooRef(oojArrayOfCharacter)`
`float[]`	`ooRef(oojArrayOfFloat)`
`double[]`	`ooRef(oojArrayOfDouble)`
`String[]`	`ooRef(oojArrayOfObject)` where array elements are of type `oojString`
`java.util.Date[]`	`ooRef(oojArrayOfObject)` where array elements are of type `oojDate`
`java.sql.Date[]`	`ooRef(oojArrayOfObject)` where array elements are of type `oojDate`
`java.sql.Time[]`	`ooRef(oojArrayOfObject)` where array elements are of type `oojTime`
`java.sql.Timestamp[]`	`ooRef(oojArrayOfObject)` where array elements are of type `oojTimestamp`
`AppClass``[]`, where *`AppClass`* is an application-defined persistence-capable class whose schema class name is *`PCclass`*.	`ooRef(oojArrayOfObject)` where array elements are of type `ooRef(`*`PCclass`*`)`

Java Array Type	Default Objectivity/DB Type
APIclass[], where *APIclass* is a persistence-capable class in the public Objectivity for Java programmer interface (for example, ooContObj or ooMap).	ooRef(oojArrayOfObject) where array elements are of type ooRef(*APIclass*)
PCinterface[], where *PCinterface* is an interface (implemented by one or more persistence-capable classes).	ooRef(oojArrayOfObject) where array elements are of type ooRef(ooObj)

The default representation for every Java array type is an object reference to an internal persistent object of an internal Objectivity / DB array class. As a consequence, array identity is preserved when data is transferred between a Java application and a database. For example, suppose that two Java objects reference the same array in persistent fields. If the two objects are saved in the database and later read by a second Java application, the two retrieved objects will also reference the same array. A change in the elements of the array will affect both referencing objects.

NOTE The Objectivity / DB representation of a variable-sized array cannot distinguish between null (no array) and an array with zero elements. As a consequence, an array of no elements is stored in (and retrieved from) the database as a null object reference.

15

Schema Evolution and Object Conversion

At some point during the lifetime of your Objectivity/DB application, you may find it necessary to modify the definition of one or more persistence-capable classes. The corresponding class descriptions in the schema of your federated database must be modified to be consistent with the new Java class declarations. If the federated database contains any persistent objects of the modified classes, those objects must be converted to make them consistent with the new class descriptions in the schema.

If such changes occur during the prototyping or development phases of a project, you could simply modify your Java class declarations as necessary and create a new federated database to use for test purposes or delete objects of the changed classes from your federated database. After your application is deployed, however, the federated database at each installation site will contain objects created by the application's end users. It would not be practical to delete the deployed federated database or to delete objects from it.

In This Chapter

Schema Evolution

Objectivity / DB provides mechanisms and tools to assist you with modifying the schema, distributing the changed schema to your installation sites, and integrating the changes into their existing federated databases. In Objectivity for Java, class descriptions in the schema are changed dynamically by a running application. The schema policy of the connection determines whether such modifications are allowed.

Deployed applications typically prohibit modifications to schema descriptions. If a new version of your deployed application uses modified persistence-capable classes, you may need to deliver the new version to installation sites along with a separate "update application" that performs the necessary schema evolution. The update application can set its schema policy to permit changes to schema descriptions and then register each of the modified classes. Registering a class updates its corresponding class description in the schema. Your update application might also convert all affected objects in the federated database. Objectivity for Java can perform simple conversions for you. If you make extensive changes to a class, however, you may need to implement your own mechanism for converting the existing objects of that class.

Schema evolution is the process of modifying class descriptions in the schema of your federated database to be consistent with new Java class declarations. Schema evolution becomes necessary when you modify the Java declaration of a persistence-capable class for which a corresponding class description already exists in the schema. Updates to the schema description of a particular class can be triggered in either of two ways: automatically when the schema descriptions are needed, or under explicit control of the application. In either case, Objectivity for Java searches the schema for the application's schema class name for the class. If it finds the name, it compares the schema description associated with that name with the Java class declaration.

If the schema description is incompatible with the Java class declaration, Objectivity for Java automatically replaces the existing schema description with a new description generated from the Java class. Because schema class descriptions are shared by Java, C++, and Smalltalk applications, it is possible that the incompatibility was introduced in a different the language environment. However, the resolution of the incompatibility will be based on the Java class definition.

WARNING You must be very careful when performing schema evolution in a concurrent environment in which multiple processes may access elements of a given class. In particular, you should avoid having one process evolve a class description in the schema while other running processes are using objects of that class.

Ideally, schema evolution should be performed when no other processes are running. After schema evolution is complete, all processes should use class definitions corresponding to the new schema description. If this approach is not practical, you should have a plan for switching each process to the new class definition in an orderly manner.

Class Modifications

If you change the name of your Java class, you do not need to update the class description in the schema. Instead, you just modify your application to use the same schema class name for your new Java class as it used for your old Java class. The schema will continue to use the same class name; all existing objects of the class will remain in the federated database, identified by the same schema class name with which they were created.

If you make any of the following changes to a Java class, the corresponding class description must be updated in the schema:

- Change the inheritance path of the class by replacing its superclass or the superclass of any of its ancestor classes.
- Add or remove persistent fields or relationships.
- Change the order of the fields of the class. Static, transient, and final fields are ignored; however, if you change the relative order of the persistent and relationship fields, the schema must be updated.
- Modify the properties of a relationship, which include the related class.
- Change the data type of a persistent field to a Java type that is incompatible with the corresponding Objectivity / DB type in the schema.
- Change the access control of a persistent field when your application's schema policy requires field access control to be the same in the schema description and the Java class.

Automatic Schema Update

Class descriptions in the schema are modified automatically as needed. When Objectivity for Java needs the schema description of a Java class, it triggers a schema comparison; the schema description is updated automatically if it is found to be incompatible with the Java class declaration.

Objectivity for Java needs the schema description for a particular persistence-capable class, `PCclass`, when:

- An instance of `PCclass` is made persistent.
- The application scans a storage object for persistent objects of the class `PCclass`.
- The description of a subclass of `PCclass` is added to, or modified in, the schema.

- The description of another class is added to, or modified in, the schema and that class references `PCclass` in a persistent field.
- The description of another class is added to, or modified in, the schema and that class has a relationship in which `PCclass` is the related class.
- An index is defined for objects of `PCclass`.

Explicit Schema Update

After you modify persistence-capable classes, you can explicitly trigger the necessary schema modifications. For example, you might write an update application to update schema descriptions for all Java classes that have been modified since the last release of a deployed application. You could deliver the update application with the new release of the deployed application. Separating schema modifications into an update application allows the deployed application to run with a schema policy that prohibits changes to schema descriptions.

To make any necessary updates to the schema's description of a persistence-capable class, you call the `registerClass` method of the connection object. The parameter to `registerClass` is the package-qualified name of the Java class that was modified. This method checks the schema descriptions of the specified class, its superclass and other ancestor classes, and all classes to which it is related based on its relationship definitions. If the class has persistent fields that reference other persistence-capable classes, `registerClass` checks their descriptions recursively. Any schema description that is found to be inconsistent with the corresponding Java class declaration is updated automatically.

NOTE Before you call `registerClass`, you should set the schema class names for your application's persistence-capable classes to the class names that are used in the schema.

Schema Comparison

Whenever Objectivity for Java needs a current schema description for a class (because the application is attempting to read or write an object of the class or because of an explicit call to `registerClass`), the existing class description in the schema is compared to the corresponding Java class declaration. Three outcomes are possible. The schema description is:

- Identical to the description that would be generated from the current Java class declaration.
- Compatible with the Java class declaration. The schema description specifies the same inheritance path, relationships, and fields as the Java class declaration. The Objectivity / DB type of every field can be mapped to the Java type of the corresponding field.

- Incompatible with the Java class declaration. At least one of the following differences exists:
  - The Java class has a different inheritance path than is specified in the schema description.
  - The schema description has a relationship that is missing from the Java class.
  - The Java class has a relationship that is missing from the schema description.
  - A relationship in the schema description is defined differently than is the corresponding relationship of the Java class.
  - The schema description has a field that is missing from the Java class.
  - The Java class has a field that is missing from the schema description.
  - The order of the fields and relationships in the schema description differs from the order of the persistent fields and relationships in the Java class.
  - A field in the schema description has an Objectivity / DB type that cannot be mapped to the corresponding Java field type. Chapter 19, "Schema Matching for Interoperability," describes the Java types that are allowed mappings for the various Objectivity / DB types.
  - A field in the schema description has a different access control than the corresponding field of the Java class, and the application's schema policy requires field access control to be the same in the schema description and the Java class.

If the schema comparison detects an incompatible schema description, Objectivity for Java generates a new schema description from the Java class declaration. The content of the new descriptions is discussed in "Content of a Schema Class Description" on page 263. Note that *all* fields in the new schema description use the default mapping for the corresponding Java type, even if the Objectivity / DB type of the field in the old schema description was an allowed mapping for the Java type.

The schema retains the original schema description for the evolved class in addition to the new one. Objectivity / DB uses the original schema description when it needs to access an object that has not been converted to the new schema description.

NOTE If the application's schema policy prohibits changes to schema descriptions, the schema remains unchanged; an `ObjySchemaException` is thrown.

Object Conversion

After a class description in the schema is changed, all objects of that class must be converted. If the class has subclasses, objects of its subclass and other descendant classes must also be converted. *Object conversion* is the process of converting persistent objects in a federated database to make their data consistent with the modified schema. The objects that must be converted are called the *affected objects*.

Objectivity for Java can perform simple conversions for you. If you make extensive changes to a class, however, you may need to implement your own mechanism for converting the existing objects of that class.

NOTE Changing the access modifier of any field or adding or deleting a *non-inline* relationship *does not* cause object conversion.

Conversion of Persistent Data

Objectivity / DB converts the persistent data of an affected object, `affectedObject`, as follows:

- A field or inline relationship that was added to the schema description is added to `affectedObject`. If the field has a primitive data type, the field of `affectedObject` is set to the default value (typically 0). If the field has a class data type, the new field of `affectedObject` is set to a null object reference.
- A field or inline relationship that was deleted from the schema description is removed from `affectedObject`.
- If a field or inline relationship was renamed, the change is treated the same as if the existing field or relationship had been deleted and a new field or relationship had been added. Thus, the old field or relationship is removed from `affectedObject`. The new field of `affectedObject` is initialized with a default value.

  If you want to rename a field without losing data, you should evolve the class in two passes, converting the objects between the two passes. In the first schema-evolution pass, you add a new field or inline relationship with the new name. You then convert objects by copying data from the old field to the new field. In the second schema-evolution pass, you remove the old field.
- Fields and inline relationships of `affectedObject` are reordered, as necessary, to match the order in the new schema description.
- An inline relationship whose definition was changed in the schema description is removed from `affectedObject` if any of the following properties of the relationship was changed:
  - Related class

- Inverse relationship
- Directionality
- Cardinality

However, if the only changes were to the relationship's delete or lock propagation, copying, or versioning behaviors, `affectedObject` retains the relationship.

- If the storage mode of a relationship was changed from non-inline to inline, inline to non-inline, short inline to long inline, or long inline to short inline the storage mode of relationships involving `affectedObject` are so modified. Whenever relationships are converted to *short inline* from any other format, references contained by the converted objects are set to null if the referenced objects are *not* in the same container.
- If the data type of a field was changed:
  - If one numeric data type is changed to a different numeric data type, the value in the field of `affectedObject` is converted to the new type. Precision may be lost if the new type is smaller that the original type. For example, precision may be lost when converting from `float64` to `float32`.
  - If one interface type, call it `A`, is changed to a different interface type, call it `B`, the object referenced in the field of `affectedObject` is retrieved. If that object's class implements the interface B, the field is set to reference the retrieved object. Otherwise, the field is set to null.
  - If an interface type is changed to the type `ooObj`, the object referenced in the field of `affectedObject` is retrieved. If that object's class is `ooObj` or a descendant class of `ooObj`, the field is set to reference the retrieved object. Otherwise, the field is set to null.
  - If the type `ooObj` is changed to an interface type, the object referenced in the field of `affectedObject` is retrieved. If that object's class implements the interface, the field is set to reference the retrieved object. Otherwise, the field is set to null.
  - If one interface array type, call it `A[]`, is changed to a different interface array type, call it `B[]`, each object in the array field of `affectedObject` is retrieved. If a retrieved object's class implements the interface `B`, the corresponding element of the array is set to reference the retrieved object. Otherwise, that array element is set to null.
  - If an interface array type is changed to the type `ooObj[]`, each object in the array field of `affectedObject` is retrieved. If a retrieved object's class is `ooObj` or a descendant class of `ooObj`, the corresponding element of the array is set to reference the retrieved object. Otherwise, that array element is set to null.
  - If the type `ooObj[]` is changed to an interface array type, each object in the array field of `affectedObject` is retrieved. If a retrieved object's class

implements the interface, the corresponding element of the array is set to reference the retrieved object. Otherwise, that array element is set to null.

- If the data type of a field is changed in any other way, the value in that field of `affectedObject` is removed. If the new data type is a primitive type, the field of `affectedObject` is set to the default value (typically 0). If the new data type is a class, the new field of `affectedObject` is set to a null object reference.

During object conversion, inherited fields are treated the same as fields defined in the class of `affectedObject`.

NOTE If you change a Java class in a way that requires more modification to objects than Objectivity / DB performs, you must implement your own update application to perform the necessary additional conversion.

Automatic Conversion

When your application retrieves an affected object by scanning a storage object for all objects of the class, Objectivity / DB automatically converts the retrieved object. When you retrieve an affected object in any other way, Objectivity / DB converts the retrieved object when you call its `fetch` method. In either case, the conversion is performed in application memory. If the local representation of the converted object belongs to a session whose open mode is read / write, the converted object is written to the federated database immediately. If the session's open mode is read only, however, the object in the federated database remains unconverted.

Explicit Conversion

You can explicitly trigger the conversion of all the affected objects in a particular storage object by calling the `convertObjects` method of that storage object. Doing so allows you to select the conversion granularity to federated database, database, or container.

For performance reasons, you may prefer to update objects explicitly rather than waiting for them to be converted when they are accessed. Explicit conversion eliminates the overhead of read-only sessions that convert the affected objects in memory without writing the converted objects to the federated database.

16

Autonomous Partitions

This chapter describes how to use Objectivity for Java to perform autonomous-partition administration tasks. You may perform these tasks only if you have purchased and installed Objectivity / DB Fault Tolerant Option (Objectivity / FTO). For a conceptual discussion of autonomous partitions, see the Objectivity / FTO and Objectivity / DRO book.

The description of each task indicates which partitions must be available to perform the task. If a task can be performed through an Objectivity / DB or Objectivity / FTO tool as an alternative to using the programming interface, the task description identifies that tool.

Unless otherwise indicated, autonomous-partition tasks are also valid in an Objectivity / DB Data Replication Option (Objectivity / DRO) environment (see Chapter 17, “Database Images”).

In This Chapter

Understanding Autonomous Partitions

An *autonomous partition* is an independent piece of a federated database. Each autonomous partition is self-sufficient in case a network or system failure occurs in another partition. Although data physically resides in database files, each autonomous partition *controls* access to particular databases (or database images) and containers.

Each autonomous partition can perform most database functions independently of other autonomous partitions, because each partition has all the system resources necessary to run an Objectivity/DB application, including a boot file, a lock server, and a system database file. The system database file contains schema information and a global catalog of all autonomous partitions, their locations, and the databases they contain. When you create, modify, or delete a partition, however, Objectivity/DB needs access to all partitions in the federated database so that it can update each partition's global catalog.

Specifying the Boot Autonomous Partition

Must have access to: The desired boot partition

To open a connection to a federated database with a particular autonomous partition as the boot autonomous partition for your application, pass the pathname of that partition's boot file to the `Connection.open` static method.

Controlling Access to Offline Partitions

By default, applications enforce the offline status of partitions. If you want a session to be able to access offline partitions other than your application's boot autonomous partition, call the `setOfflineMode` method of the session to set the offline mode for your application. Specify the offline mode `IGNORE` to ignore the offline status of partitions.

If you later want to return to enforcing the offline status of partitions, call the session's `setOfflineMode` method again, specifying the offline mode `ENFORCE`.

To check whether the application is enforcing or ignoring the offline status of partitions, call the `getOfflineMode` method of the session.

Creating an Autonomous Partition

Must have access to: All autonomous partitions

Tool alternative: `oonewap` (see the Objectivity / FTO and Objectivity / DRO book)

The first autonomous partition in a federated database is created implicitly when you create the federated database. You must create any additional partitions explicitly.

To create a new autonomous partition, call one of the `newAP` methods of the federated database. These methods create:

- A system database file
- A journal file
- A boot file
- An autonomous partition in the federated database locked for write
- An instance of `ooAPObj` in your application

The simpler `newAP` method requires you to specify the system name of the partition, the name of the lock server host for the partition, and the host and directory path where the partition's system database file is to be located. It creates the partition's boot file and the journal file in the same directory, on the same host as the partition's system database file.

The second `newAP` method additionally requires you to specify the host and directory path for the partition's boot file and for its journal file.

When you commit or checkpoint the transaction in which you create a partition, the partition is created in the federated database. If you instead abort the transaction, the partition becomes a dead object and no physical partition is created.

EXAMPLE This code fragment creates a session and retrieves its associated federated database. It then calls the federated database's hasAP method to check whether an autonomous partition named VehiclesPartition exists and if so, retrieves it from the federated database. You should check whether the partition exists before doing the lookup, because the lookup operation will throw an exception if a partition with the name VehiclesPartition does not exist in the federated database. If the partition does not exist, the example creates a new partition in the federated database.

```
Session session = new Session();
session.begin();
ooFDObj vrcFD = session.getFD();
if (vrcFD.hasAP("VehiclesPartition"))
   ooAPObj vehiclesPartition =
          vrcFD.lookupAP("VehiclesPartition");
else {
   vehiclesPartition = vrcFD.newAP("VehiclesAP");
   System.out.println("Created partition \"VehiclesAP\".");
}
session.commit();
```

Retrieving a Partition

You can retrieve a partition from the federated database that contains the partition, from a database that the partition contains, or from a container that the partition controls.

Getting the Boot Autonomous Partition

Must have access to: The boot autonomous partition

Call the getBootAP method of the federated database to retrieve the autonomous partition whose boot file was used to open a connection to the federated database. This autonomous partition is the boot autonomous partition for the application.

Getting an Autonomous Partition by System Name

Must have access to: The desired autonomous partition

Call the lookupAP method of the federated database to retrieve the autonomous partition with a specified system name. This method throws an exception if the federated database does not have a partition with the specified name. You can call the hasAP method of the federated database to check whether the partition exists.

Iterating Over All Partitions

Must have access to: All autonomous partitions

To obtain an iterator that finds all partitions in the federated database, call the `containedAPs` method of the federated database.

Finding the Partition that Contains a Database

Must have access to: The containing partition

If a single image of a database exists, you can find the autonomous partition that contains the database by calling the `getContainingPartition` method of the database.

Finding the Partition that Controls a Container

Must have access to: The controlling partition

To find the autonomous partition that controls a container, call the container's `getControlledBy` method.

Getting and Changing Attributes of a Partition

The following attributes of a partition are set when the partition is created:

- System name
- Lock server host
- System database file host
- System database file path
- Boot file host and boot file path
- Journal file directory host and journal file directory path
- Offline status (set to online by default)

You can get any of these attributes and you can change the offline status.

Getting the Attributes of a Partition

Must have access to: The partition of interest

Tool alternative: `oochange` with the `oochange` tool with the `-ap` or `-id` flag and no other parameters (see the the Objectivity / DB administration book)

You can use the following methods of an autonomous partition to get its attributes:

Member Function	Gets Attribute
`getName`	System name
`getLockServerHost`	Lock server host
`getSystemDBFileHost`	Host for system database file
`getSystemDBFilePath`	Path for directory of system database file
`getBootFileHost`	Host for boot file
`getBootFilePath`	Path for directory of boot file
`getJournalDirHost`	Host for journal file
`getJournalDirPath`	Path for directory of journal file
`isOnline`	Offline status

Changing the Offline Status

Must have access to: The autonomous partition of interest

Tool alternative: `oochange` with the `-ap` or `-id` option (see the Objectivity / DB administration book)

To change the offline status of a partition, call its `setOnline` method. Pass false as the parameter to make the partition offline; pass true as the parameter to make the partition online.

Getting and Changing Controlled Objects

An autonomous partition controls access to:

- All the databases it contains
- Any container whose control has been transferred to the partition
- All containers in the databases it contains except those containers whose control has been transferred to a different partition

---

NOTE Any method that changes the control of a database or container from a source partition to a destination partition ignores the offline status of those two partitions.

Contained Databases

Objectivity for Java creates all databases in the initial autonomous partition of the federated database. You can:

- Move a database from one partition to another
- Iterate over all databases in a partition

Moving a Database to a Different Partition

Must have access to: All autonomous partitions

Tool alternative: `oochangedb` with the `-movetoap` option (see the Objectivity/DB administration book)

You can move a database to a different partition unless there is more than one image of the database. To move a database to a different partition, call the `changePartition` method of the database. This method creates a new image of the database in the destination autonomous partition and deletes the database from its current partition. If there are multiple images of the database, or if the database has been updated during the partition-changing transaction, an exception is thrown.

This method changes logical containment and updates the global catalog in all autonomous partitions. If you also want to change the physical location of a database file, you must do so using the Objectivity/DB `oochangedb` tool with the `-host` and/or `-filepath` flags.

Iterating Over Databases in a Partition

Must have access to: The partition of interest

To obtain an iterator that finds all databases in an autonomous partition, call the `imagesContainedIn` method of the partition.

Controlled Containers

A newly created container comes under the control of its database's containing partition. You can use Objectivity for Java to:

- Transfer control of a container to a specified partition
- Return control to the containing partition
- Clear a partition of all the containers whose control has been transferred to the partition
- Iterate over all containers controlled by a partition

Transferring control of a container causes the container to be moved physically to the system database file of the destination partition. Returning control causes the

container to be moved physically to the database file of its database. Changing control of a container does not affect the container's logical containment relationships.

NOTE You cannot transfer or return control of a container that has been modified until you commit the changes.

Transferring Control of a Container

Must have access to:

- The autonomous partition of the container's database
- The autonomous partition that currently controls the container (if control has already been transferred)
- The autonomous partition to which control is being transferred

Tool alternative: `oochangecont` (see the Objectivity / FTO and Objectivity / DRO book)

To transfer control of a container, call the `transferControl` method of the container. The destination partition can be any partition, even one that does not contain an image of the container's database.

Returning Control of a Container

Must have access to:

- The autonomous partition that currently controls the container
- The autonomous partition of the container's database

Tool alternative: `oochangecont` (see the Objectivity / FTO and Objectivity / DRO book)

To return control of a container to the autonomous partition that contains the container's database, call the `returnControl` method of the container.

Clearing an Autonomous Partition

Must have access to:

- The autonomous partition being cleared
- The "home partition" for each container whose control has been transferred to the partition being cleared

Tool alternative: `ooclearap` (see the Objectivity / FTO and Objectivity / DRO book)

Clearing an autonomous partition releases the partition's control of all containers whose control has been transferred to the partition. The control of each such container is returned to the autonomous partition of that container's database.

To clear a partition, call the `returnAll` method of the partition.

Iterating Over Containers Controlled by a Partition

Must have access to: The partition of interest

To obtain an iterator that finds all containers controlled by an autonomous partition, call the `containersControlledBy` method of the partition. Indirectly controlled containers, which the partition controls by virtue of containing their databases, are not included.

Using a Partition as a Scope Object

An autonomous partition may serve as a scope object. See "Scope Names" on page 215 for information on how scope naming works. A scope object uses the hashing mechanism of a hashed container to associate each name in the name scope with the appropriate object. An autonomous partition uses the roots container of the default database of its federated database to store names.

When you name a transient object in the scope of an autonomous partition, the object is made persistent. If the transient object is a container, it is clustered in the default database of its federated database; if the object is a basic object, it is clustered in the roots container of the default database of its federated database. See Chapter 12, "Clustering Objects," for more information about clustering.

The following methods of an autonomous partition are used to manage its scope:

- `ooAPObj.nameObj`
- `ooAPObj.lookupObj`
- `ooAPObj.lookupObjName`
- `ooAPObj.unnameObj`

Deleting a Partition

Must have access to: All autonomous partitions

Tool alternative: `oodeleteap` (see the Objectivity / FTO and Objectivity / DRO book)

To delete an autonomous partition, call its `delete` method.

Deleting an autonomous partition:

- Clears the partition, returning control of any containers whose control has been transferred to the partition to be deleted. Returning control physically relocates each container to its containing database.
- (*DRO*) Deletes any replicated database images that the partition contains.
- Deletes the autonomous partition system database.
- Deletes the autonomous partition boot file.

You cannot delete an autonomous partition if it contains any databases (or the last image of any database); you must first move such databases to another partition.

Until the current transaction is committed, the local representation of the partition continues to exist in your application's memory. When you commit or checkpoint the transaction in which you delete a partition, the physical partition is deleted from the federated database. If the partition controls the last image of a database, an exception will be thrown and the partition is left unchanged.

WARNING A partition deleted in a transaction is marked dead when the deletion occurs. A deleted partition is not restored if you abort the transaction. In other words, this operation cannot be undone by aborting the transaction.

17

Database Images

This chapter describes how to use Objectivity for Java to perform database replication tasks. You may perform these tasks only if you have purchased and installed Objectivity / DB Data Replication Option (Objectivity / DRO). For a conceptual discussion of database images, see the Objectivity / FTO and Objectivity / DRO book.

Before you can install and use Objectivity / DRO, you must install both the Objectivity / DB Fault Tolerant Option (Objectivity / FTO) and the Advanced Multithreaded Server (AMS). You must have at least two autonomous partitions in your federated database. For information on installing AMS, see the *Installation and Platform Notes* for your operating system. For information on using AMS, see the Objectivity / DB administration book.

The description of each task indicates which partitions must be available to perform the task. If a task can be performed through an Objectivity / DB or Objectivity / DRO tool as an alternative to using the programming interface, the task description identifies that tool.

In This Chapter

Understanding Database Images

Objectivity / DRO enables you to create and manage multiple replicas of a database, called *database images*. If an application's boot partition contains an image of the database, the application will use that image; otherwise, the application reads a single image in a different partition. If one image of a particular database becomes unavailable due to a network or machine failure, work may continue with an available image.

All images of a database share the same system name and database identifier, but each image is controlled by a different autonomous partition (see Chapter 16, "Autonomous Partitions," for information on autonomous partitions) and each image has a distinct *weight*, which is used to determine whether a *quorum* of images exists. In general, tasks affecting database images require that a quorum of the database images be available (an image is available if the containing partition is available).

All images of a database are either read-only or read-write (see "Making a Database Read-Only" on page 94). If you make one database image read-only, *all* images are automatically made read-only. While a database is read-only, you cannot add, delete, or change the properties of an individual image.

In general, Objectivity / DB tools and Objectivity for Java methods operate the same whether or not you have installed Objectivity / DRO and created multiple images of databases. Several methods of `ooDBObj`, however, can be used to return information if there is one image of a database, but throw an exception if the federated database contains multiple images of the target database. When this is the case, an alternative method is available for use with multiple images, as shown in the following table.

Method for Single Image	Method for Multiple Images
`getContainingPartition`	`containingImage`
`filename`	`getImageFileName`
`hostName`	`getImageHostName` `getImagePathName`

Enabling Nonquorum Reads

Must have access to: At least one partition containing an image of the database

By default, applications cannot read or write a replicated database unless they have access to a quorum of its images. An application can enable reading a database even when a quorum of images is not available.

To allow your application to read a database even if a quorum of images is not available, call the `setNonQuorumReadAllowed` method of the database, passing true as the parameter. Nonquorum reads will be disabled automatically at the end of the transaction.

To require a quorum before your application can read a database, call its `setNonQuorumReadAllowed` method, passing false as the parameter.

To test whether your application can read from the database if a quorum is not available, call the `isNonQuorumReadAllowed` method.

To test whether your application is currently reading from the database without a an available quorum, call the `isNonQuorumRead` method.

Creating a Database Image

Must have access to: All autonomous partitions

Tool alternative: `oonewdbimage` (see the Objectivity / FTO and Objectivity / DRO book)

You can create an image of a particular database in any autonomous partition that does not already contain an image of that database. Before you create a database image in an autonomous partition on a new host machine, you must start an AMS server on that host. All images of all replicated databases must reside on host machines that are running AMS.

To replicate a database that has only one image, AMS must be running on both the host machine on which the original database file resides and the host machine that is to contain the new image.

To create a database image, call the `replicate` method of the database. The parameters identify the autonomous partition in which to create the new image, the host machine and directory path for the image's database file, and the quorum weight for the new image.

If the database is read-only, you must change it back to read-write before you can create a new image of it (see "Making a Database Read-Only" on page 94).

EXAMPLE This example creates an image of a database named `DB1` with a weight `3` in an autonomous partition named `Partition2`.

```
ooFDObj fd = session.getFD();
session.begin();
db = fd.lookupDB("DB1");
ap = fd.lookupAP("Partition2");
db.replicate(ap, "", "", 3);
session.commit();
```

Getting and Changing Attributes of an Image

The following attributes of a database image are set when the image is created:

- The host where the image's database file is located
- The directory path where the image's database file is located
- The weight to be used in quorum calculations

You can get any of these attributes and you can change the weight of the image.

Getting the Attributes of an Image

Must have access to: The partition containing the image

Tool alternative: `oochangedb` with the `-db` or `-id` flag, the `-ap` flag, and no other flags (see the Objectivity / DB administration book)

You can use the following methods of a database to get the attributes of a database image; the parameter specifies the autonomous partition containing the image of interest.

Member Function	Gets Attribute
`getImageHostName`	Host where database file is located
`getImagePathName`	Pathname of directory where database file is located
`getImageFileName`	Fully qualified name of the database file
`getImageWeight`	Weight

Changing the Weight of an Image

Must have access to: All autonomous partitions

Tool alternative: `oochangedb` (see the Objectivity / DB administration book)

To change the weight of a database image, call the `setImageWeight` method of a database. The parameters specify the autonomous partition containing the image and the new weight for that image. This method throws an exception if the specified partition does not contain an image of the database.

If the database is read-only, you must change it back to read-write before you can change its weight (see “Making a Database Read-Only” on page 94).

Checking Number and Availability of Images

Checking Replication

Must have access to: At least one partition containing an image of the database

To test whether the federated database contains multiple images of a database, call the `isReplicated` method of the database.

To find out how many images of a database exist, call its `getImageCount` method.

Checking Availability

Must have access to: At least one partition containing an image of the database

To test whether a database is available, call its `isAvailable` method. The result is true if a quorum of images is physically accessible.

To test whether a particular partition contains an image of a database, call the `hasImageIn` method of the database, passing the partition as the parameter.

To test whether the image of a database in a particular partition is available, call the `isImageAvailable` method, passing the partition as the parameter. The result is true if the specified partition has an image of the database and that partition is accessible to the current process.

Getting and Setting the Tie Breaker

You can add, remove, or retrieve the tie-breaker partition for a database.

Setting the Tie-Breaker Partition

Must have access to: All autonomous partitions

Tool alternative: `oonewdbimage` with the `-tiebreaker` flag (see the Objectivity / FTO and Objectivity / DRO book)

To add a tie-breaker partition for a database, or to change the tie-breaker to be a different partition, call the `setTieBreaker` method of the database. The parameter is the autonomous partition that is to serve as the tie breaker during quorum negotiation; it can be any partition that does not already contain an image of the database. The host for the tie-breaker partition must be running a lock server.

Removing the Tie-Breaker Partition

Must have access to: All autonomous partitions

Tool alternative: `oodeletedbimage` with the `-tiebreaker` flag (see the Objectivity / FTO and Objectivity / DRO book)

To remove the tie-breaker partition for a database, call the `setTieBreaker` method of the database, passing null as the parameter.

Getting the Tie-Breaker Partition

Must have access to: The tie-breaker partition

To retrieve the tie-breaker partition for a database, call the `getTieBreaker` method of the database.

Iterating Over Partitions That Contain an Image

Must have access to: All autonomous partitions that contain an image of the database

To obtain an iterator that finds all partitions that contain an image of a particular database, call the `containingImage` method of the database.

Deleting a Database Image

Must have access to: All autonomous partitions

Tool alternative: `oodeletedbimage` (see the Objectivity / FTO and Objectivity / DRO book)

To delete a particular image of a database, call the `deleteImage` method of the database; the parameter indicates the autonomous partition that contains the image to be deleted. This method allows you to specify whether the deletion should proceed if the specified partition contains the last remaining image of the database. If the given partition does not contain an image of the database, this method throws an exception.

If the database is read-only, you must change it back to read-write before you can delete an image of it (see “Making a Database Read-Only” on page 94).

Resynchronizing Database Images

Must have access to: A quorum of images for any database to be resynchronized

After a hardware or network failure is corrected, the federated database must be restored to a consistent state. Your installation should have a recovery application that is run when autonomous partitions that were inaccessible become accessible again. The recovery procedure should resynchronize every database with an image in the restored partitions.

To resynchronize the images of a particular database, call the `negotiateQuorum` method of the database. The parameter is a lock mode indicating the type of lock to obtain for the database. This method forces recalculation of the quorum for the database, which causes Objectivity / DB to synchronize out-of-date images with images that were updated while one or more partitions were unavailable. If the application does not have access to a quorum, the images are not resynchronized.

The negotiateQuorum method should be used only after the federated database has been restored to a consistent state by the oocleanup administrative tool (see the Objectivity/DB administration book).

EXAMPLE This example first checks that all partitions are available. If so, it resynchronizes all databases in the federated database.

```
void resynch (ooFDObj fd) {
   ooAPObj ap ;
   ooDBObj db ;

   // Initialize an iterator for all partitions
   Iterator apItr = fd.containedAPs() ;
   while(apItr.hasNext()) {
      ap = (ooAPObj)apItr.next() ;
      if (!ap.isAvailable()) {
         System.out.println(ap.getName() +
                              " is not available");
         return ;
      }
   }
   // All partitions are available

   // Initialize an iterator for all databases
   Iterator dbItr = fd.containedDBs() ;

   // Use the iterator to resynchronize each database
   try {
      while(dbItr.hasNext()) {
         db = (ooDBObj)dbItr.next() ;
         db.negotiateQuorum(oo.READ);
         System.out.println(db.getName() +
                              " has been resynchronized ");
      }
   }
   catch (ObjyRuntimeException e1) {
      System.out.println("negotiateQuorum failed");
   }
}
```

18

In-Process Lock Server

You can improve performance in certain Objectivity for Java applications by using an *in-process lock server.*

Before you can use an in-process lock server, you must purchase and install Objectivity / DB In-Process Lock Server Option (Objectivity / IPLS).

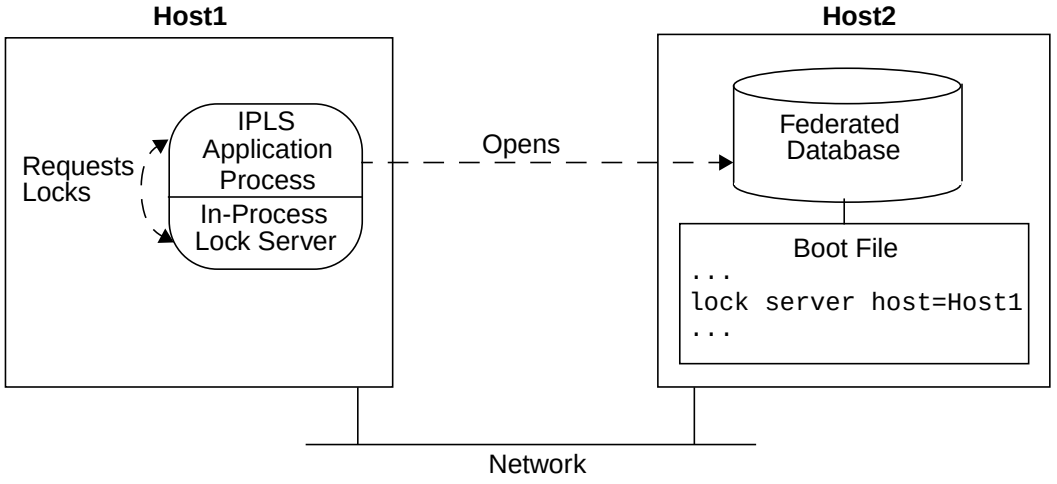
In This Chapter

Understanding In-Process Lock Servers

When multiple applications access a federated database, the access rights for those applications are coordinated by a lock server that runs as a separate process. If, however, all or most lock requests originate from a single, multithreaded application, the application can improve its runtime speed by starting an *in-process lock server.* An application that starts an in-process lock server is called an *IPLS application*.

An in-process lock server is just like a standard lock server, except that it runs in the IPLS application process. This enables the IPLS application to request locks through simple method calls without having to send these requests to an external process.

NOTE Like any other application, an IPLS application always uses the lock server that is specified by the federated database. Consequently, an IPLS application uses its own in-process lock server only if the connected federated database names the application's host as the lock server host, as shown in the following figure.



When an in-process lock server is started, the IPLS application becomes the lock-server process for the workstation on which it is running. Consequently, if a federated database names this workstation as its lock server host, all applications accessing that federated database will send their lock requests to the application running the in-process lock server. The in-process lock server creates a separate *listener thread* to service requests from external applications.

A large number of lock requests from external applications could reduce the performance of the IPLS application; normally an in-process lock server is consulted only by the application that started it.

Just as you cannot run two lock-server processes on the same host, you cannot run two IPLS applications on the same host; an in-process lock server cannot be started if any lock-server process or IPLS application is already running on the same host.

NOTE You use a separate lock-server process during development—for example, while you are creating the federated database. You typically modify the application to start an in-process lock server as a later step—for example, while tuning the application's runtime speed.

Starting an In-Process Lock Server

You start an in-process lock server after creating a session object and before starting the first transaction:

1. Open a connection to the federated database.
2. Create a session object (but do *not* start a transaction).
3. Call the call the `startInternalLS` method of the connection object.

An in-process lock server can be started only if no other lock-server process or IPLS application is currently running on the same host. You can check whether a lock server (external or in-process) is running on your host by calling the `checkLS` method of the connection object.

NOTE When you install Objectivity/DB, you normally configure the workstation to start the standard lock server automatically every time the machine is rebooted. You should reconfigure any workstation where you plan to run an IPLS application.

Stopping an In-Process Lock Server

You stop an in-process lock server by calling the `stopInternalLS` method of the connection object at the end of the IPLS application, and after committing or aborting all transactions in all sessions.

The `stopInternalLS` method safely shuts down an in-process lock server so that you can terminate the IPLS application without harming any external applications that may be using the in-process lock server. By default, this method waits indefinitely for other applications to terminate their transactions, stopping the in-process lock server when all active transactions are finished. You can optionally specify a finite wait period, after which `stopInternalLS` returns, even if transactions are not terminated. If active transactions do not finish and the wait period expires, the method optionally stops the in-process lock server or allows it to continue running so you can try again later.

Example IPLS Application

This example shows a simple outline for an IPLS application. The static method `main` of the `IPLSInit` class opens a connection to the federated database and creates the first session object, then starts the in-process lock server after checking

whether any other lock server is running on the current host. The application creates other sessions and performs its Objectivity / DB operations. After all transactions have ended, it stops the in-process lock server before closing the connection to the federated database.

```
import com.objy.db.*;     //Import Objectivity for Java exceptions
import com.objy.db.app.*; //Import Objectivity for Java classes
…
public class IPLSInit {
    public static void main(String args[]) {
        Connection connection;
        Session session1;
        …
        // Open a connection to a federated database
        try {
            connection = Connection.open("myFD", oo.openReadWrite);
        } catch (DatabaseNotFoundException e1) {
            System.out.println( "\nFederated database not found.");
            return;
        // Create first session
        session1 = new Session();
        // Find out whether another lock server is running
        // on the current host
        if (connection.checkLS("myHost")) {
           … // Advise user that another lock server is running
           … // Proceed or not according to user choice
        }
        // Start the lock server
        if (connection.startInternalLS()) {
            … // Create other sessions, if desired
            … // Perform Objectivity/DB operations in transactions
            // After all transactions have ended, wait for 5 minutes
            // (300 seconds) to allow the in-process lock server to
            // finish servicing any transactions of external
            // applications; then stop the in-process lock server
            connection.stopInternalLS(300, true);
        }
        // Close the connection
        try {
            connection.close();
        } catch (DatabaseClosedException e3) {
            System.out.println("\nConnection already closed.");
            return;
        }
    } // End main
    …
} // End class IPLSInit
```

19

Schema Matching for Interoperability

An Objectivity / DB federated database can be shared by Java applications and applications written in C++ or Smalltalk. Each application maps data for an object between the Objectivity / DB data types specified in the schema description for its class and data types native to the application. This chapter explains how to define a persistence-capable Java class that is compatible with an existing schema class description. You can ignore this chapter if your application doesn't interoperate with applications written in different languages.

In This Chapter

Interoperability

If you develop a Java application to interoperate with applications in other languages, you must be sure that your Java classes and the corresponding classes in the other languages are all compatible with the schema of the shared federated database.

- If the schema does not already contain a description for a particular class, be sure that the class description that gets added to the schema contains Objectivity / DB types that are supported by all application languages. You should first decide on the Objectivity / DB field types, then define the corresponding classes in each application language.

  Refer to Objectivity documentation for the various languages to find out how each language adds a class description to the schema and what Objectivity / DB types each language supports.
- If a class description already exists in the schema, define your Java class to be compatible with the existing class description. You can use the Objectivity / DB objects and types browser to see an existing class description.

  Some field types and inheritance relationships preclude interoperability because they are not supported by all application languages. Java does not support the following characteristics; if a schema description of a class contains any of these characteristics, you cannot define a compatible Java class:
  - More than one immediate superclass (multiple inheritance)
  - A field containing a fixed-size array of any data type
  - A field containing an embedded application-specific class

---

NOTE The Objectivity / C++ programming interface allows an application to partition class definitions among multiple schemas in the same federated database. However, Objectivity for Java supports only a single default schema, so a Java application cannot access classes that reside in a nondefault (named) schema.

Selecting the Class Name

Your application can use whatever naming convention you prefer for persistence-capable Java classes. The schema identifies each class with a unique class name. A Java application also has a schema class name for each persistence-capable class. When the application reads or writes a persistent object, Objectivity for Java maps between the Java name of the object's class and the application's schema class name for the class.

If a class name in the schema is a legal Java class name, you may use the same name for the corresponding Java class, but you are not required to do so. The only requirement is that your application's schema class name is the same as the class name in the schema.

- If your class is in the default package and its Java name is the same as the class name in the schema, this requirement is satisfied automatically; the default schema class name is the fully qualified class name with the dots replaced by underscores.
- Otherwise, your application must use a custom schema class name for the class. To set a custom schema class name, you call the `setSchemaClassName` method of the connection object, specifying as parameters the Java class name and the corresponding class name used in the schema.

NOTE You should set the custom schema class names for your application *before* performing any operations that read or write persistent objects. For more information, see "Schema Class Names" on page 264.

Defining the Inheritance Hierarchy

When you define a Java class corresponding to an existing schema class definition, the inheritance hierarchy for your Java class must be the same as the inheritance hierarchy reflected in schema class descriptions. In particular, the superclass for your class should be the Java class whose schema class name appears as the superclass in the class definition. You can use the Objectivity / DB objects and types browser to see the superclass specified in an existing class description.

WARNING Objectivity / DB supports multiple inheritance; persistence-capable C++ classes may have more than one immediate superclass. Because Java does not support multiple inheritance, you cannot define a Java class that is equivalent to any class in the schema that uses multiple inheritance.

Defining the Relationships

When you define a Java class corresponding to an existing schema class definition, you should define a relationship corresponding to each relationship in the class description. You can use the Objectivity / DB objects and types browser to see the relationships in an existing class description.

Define each relationship of your Java class as described in the schema. The relationship fields of your Java class should be in the order in which the corresponding relationships appear in the class description. The related class should be the Java class whose schema class name appears in the schema class description. The following properties of the relationship should be the same in your class as in the schema:

- The relationship name.
- The name of the inverse relationship (if any).
- Directionality.
- Cardinality.
- Delete propagation behavior.
- Lock propagation behavior.
- Copying behavior.
- Versioning behavior.
- Inline, short inline, or non-inline.

Defining the Persistent Fields

When you define a Java class corresponding to an existing schema class definition, you should define a persistent field corresponding to each field in the class description. The fields of your Java class should have the same names as the field names in the class description; they should be declared in your class in the order in which they appear in the class description.

Because different languages support different field access control settings, you do not need to define each Java field with the same access control as appears in the schema. Instead, you should use the field access control that makes sense for your Java application and set the schema policy of your application to disregard field access control when comparing a Java class with the corresponding class description in the schema.

You must select a Java type for each field that is compatible with the corresponding Objectivity / DB data type in the class description. You can use the Objectivity / DB objects and types browser to see the Objectivity / DB type of each field in an existing class description.

Objectivity / DB supports a wide range of data types for persistent data. A given Objectivity / DB type may be mapped to more than one Java data type. However, each Objectivity / DB type has one or more recommended mappings; some Objectivity / DB types can be mapped to additional Java types, but these mappings are not recommended. If at all possible, you should select the Java type of a field from the recommended mappings for the Objectivity / DB type of the field.

Mapping Objectivity/DB Types to Java Types

This section groups Objectivity / DB data types according to the classification presented in Chapter 21, “Objectivity / DB Data Types”. For each group of Objectivity / DB data types, a table lists the recommended mappings. When you define a Java class to be compatible with a class description in the schema, you should use these tables to select a recommended Java type corresponding to the Objectivity / DB type of each field. Any notes or warnings about potential problems appear before the corresponding table of mappings.

WARNING Each Objectivity / DB field is defined to contain either a single value or a fixed-sized array of values of the same type. Because Java does not support fixed-sized arrays, you cannot define a Java class that is equivalent to a class in the schema with a field that contains a fixed-sized array.

Object-References

This section describes the Java types that are compatible with Objectivity / DB object references when the referenced class is an application-defined persistence-capable class or a persistence-capable class in the public Objectivity for Java programmer interface. Object references to Objectivity / DB internal persistence-capable classes are described in later sections.

Recommending Mappings

The following table lists the recommended Java type for Objectivity / DB object-reference types.

Objectivity/DB Object-Reference Types	Recommended Java Type
ooRef(*AppClass*) ooShortRef(*AppClass*) where *AppClass* is an application-defined persistence-capable class.	*PCclass* where *PCclass* is a persistence-capable class that matches the schema description for *AppClass* and for which the application uses the schema class name *AppClass*.
ooRef(*APIclass*) ooShortRef(*APIclass*) where *APIclass* is a persistence-capable class in the public Objectivity for Java programmer interface (for example, ooContObj or ooMap).	*APIclass*

Alternative Mappings

A field whose declared Java data type is an interface is represented in the schema by the Objectivity / DB type ooRef(ooObj). As a consequence, you may map a regular or short reference to ooObj to any interface type. Of course, you should only use an interface that is implemented by some persistence-capable class.

The following table lists the Java classes that are allowed mappings for Objectivity / DB object-references to ooObj.

Objectivity/DB Reference Types	Allowed Java Type
ooRef(ooObj) ooShortRef(ooObj)	*PCinterface* where *PCinterface* is an interface (implemented by one or more persistence-capable classes)

Numeric and Character Data

The following table lists the recommended Java types for each of the Objectivity / DB primitive numeric types that are used for character, integer, and floating-point data.

Objectivity/DB Primitive Type	Recommended Java Types
int8	byte
int16	short
int32	int
int64	long
uint8	boolean short
uint16	character int
uint32	long
uint64	long
float32	float
float64	double
char	byte

Strings

The schema description for a field containing a string specifies an embedded Objectivity / DB non-persistence-capable class. If a class C has a field whose type is a non-persistence-capable string class, an instance of that string class is embedded within the data for a persistent object of class C. Although Java has no way to

embed one instance within the data for another, Objectivity for Java can map these embedded instances into standard Java references to Java string classes.

The following table lists the recommended Java types for each Objectivity / DB non-persistence-capable string class.

Objectivity/DB Embedded String Class	Recommended Java Types
`ooVString`	`String`[1] `StringBuffer`[1]
`ooUTF8String`	`String` `StringBuffer`

1. Unicode characters will be stored and retrieved correctly, but they may not be rendered correctly by some tools, such as the Objectivity/DB browser.

The `ooVString` class is intended for strings of ASCII characters only; if a string with Unicode characters is stored as an `ooVString` and then retrieved by an application, the retrieved string is not guaranteed to be rendered correctly. In contrast, the `ooUTF8String` class can represent unicode characters that will be rendered correctly on all platforms supporting unicode.

NOTE If a schema class description contains an `ooVString` field, you should avoid using unicode characters in strings stored in the corresponding field of objects of your Java class.

Objectivity / DB also has a persistence-capable string class, `oojString`, that is used *only* for the elements of a string array of class `ArrayOfObject`; you should map references to a string array to the Java type `String[]` as described in “Arrays” on page 310. Because schema descriptions do not contain fields that are object references to the class `oojString`, you do not need to map such object-reference types to a Java string class except in the context of the containing array.

WARNING Objectivity / C++ applications can define special non-persistence-capable optimized string classes named `ooString_`*N* where *N* is the number of characters for which strings of the class are optimized. If a C++ persistence-capable class has a field of type `ooString_`*N*, an instance of `ooString_`*N* is embedded within the data for a persistent object of the class. Because Java has no way to embed one instance within the data for another, you cannot define a Java class that is equivalent to any class in the schema with a field whose Objectivity / DB type is `ooString_`*N*.

Date and Time Data

The schema description for a field containing date or time data specifies either an embedded Objectivity / DB non-persistence-capable class or an object-reference type for which the referenced class is an internal Objectivity / DB persistence-capable class.

Embedded Data and Time Classes

If a class C has a field whose type is a non-persistence-capable date or time class, an instance of that date or time class is embedded within the data for a persistent object of class C. Although Java has no way to embed one instance within the data for another, Objectivity for Java can map these embedded instances into standard Java references to Java date or time classes. Mapping an Objectivity non-persistence-capable class to a Java class can lead to two differences between data in Java memory and the corresponding data after it has been written to the federated database:

- Java object identity is lost in the federated database.
- Null Java values may be replaced by 0 in the federated database.

For a discussion of these differences, see “Data in the Federated Database” on page 326.

NOTE If a class description contains a field with an embedded non-persistence-capable date or time class, objects of your corresponding Java class should not rely on object identity in that field or use null values in that field. (The loss of object identity should not cause problems, because all Java date and time classes are immutable.)

The following table lists the recommended Java types for each Objectivity / DB non-persistence-capable date and time class.

Objectivity/DB Embedded Date or Time Class	Recommended Java Types
`ooSQLdate`	`java.util.Date`[1, 2] `java.sql.Date`[1, 2]
`ooSQLnull_date`	`java.util.Date`[1] `java.sql.Date`[1]
`ooSQLtime`	`java.sql.Time`[1, 2]
`ooSQLnull_time`	`java.sql.Time`[1]

Objectivity/DB Embedded Date or Time Class	Recommended Java Types
ooSQLtimestamp	java.sql.Timestamp[1, 2]
ooSQLnull_timestamp	java.sql.Timestamp[1]

1. Does not preserve object identity.
2. Null values will be stored as zero and retrieved as zero.

Object References to Data and Time Classes

The following table lists the recommended Java types for object references to each Objectivity / DB persistence-capable date and time class.

Objectivity/DB Object-Reference to Date or Time Class	Recommended Java Types
ooRef(oojDate)	java.util.Date java.sql.Date
ooRef(oojTime)	java.sql.Time
ooRef(oojTimestamp)	java.sql.Timestamp

Arrays

The schema description for a field containing a variable-length array specifies either an embedded non-persistence-capable Objectivity / DB array class or an object-reference type for which the referenced class is an internal Objectivity / DB persistence-capable array class.

Embedded Array Classes

Java preserves array identity; Objectivity / DB's persistence-capable array classes preserve identity but its non-persistence-capable array classes do not. If a non-persistence-capable array class is mapped to a Java array, array identity is lost when a Java array is written to the federated database. For example, suppose that two Java objects reference the same array in persistent fields. If the two objects are saved in the database and later read by a Java application, the two retrieved objects will reference the different arrays.

NOTE If a class description contains a field with an embedded non-persistence-capable array class, objects of your corresponding Java class should not rely on array identify in that field.

Recommended Mappings

The following table lists the recommended Java type for each Objectivity / DB non-persistence-capable array class.

Objectivity/DB Embedded Array Classes	Recommended Java Type
`ooVArray(int8)`	`byte[]`[1]
`ooVArray(int16)`	`short[]`[1]
`ooVArray(int32)`	`int[]`[1]
`ooVArray(int64)`	`long[]`[1]
`ooVArray(uint8)`	`boolean[]`[1]
`ooVArray(uint16)`	`char[]`[1]
`ooVArray(float32)`	`float[]`[1]
`ooVArray(float64)`	`double[]`[1]
`ooVArray(ooRef(`*`AppClass`*`))` `ooVArray(ooShortRef(`*`AppClass`*`))` where *`AppClass`* is an application-defined persistence-capable class.	*`PCclass`*`[]`[1] where *`PCclass`* is a persistence-capable class that matches the schema description for *`AppClass`* and for which the application uses the schema class name *`AppClass`*.
`ooVArray(ooRef(`*`APIclass`*`))` `ooVArray(ooShortRef(`*`APIclass`*`))` where *`APIclass`* is a persistence-capable class in the public Objectivity for Java programmer interface (for example, `ooContObj` or `ooMap`).	*`APIclass`*`[]`[1]
1. Does not preserve array identity.	

Alternative Mappings

A field whose declared Java data type is an interface is represented in the schema by the Objectivity / DB type `ooRef(ooObj)`. As a consequence, you may map an array of object references to `ooObj` to an array of any interface type. Of course, you should only use an interface that is implemented by some persistence-capable class.

The following table lists the Java classes that are allowed mappings for Objectivity / DB non-persistence-capable array classes involving object references to `ooObj`.

Objectivity/DB Embedded Array Classes	Allowed Java Types
`ooVArray(ooRef(ooObj))` `ooVArray(ooShortRef(ooObj))`	*`PCinterface`*`[]`[1] where *`PCinterface`* is an interface (implemented by one or more persistence-capable classes)
1. Does not preserve array identity.	

Object-References to Array Classes

An object reference to an Objectivity / DB array class corresponds to a Java array. In the case of an array of objects, the class of the array elements determines the appropriate Java array type.

Recommended Mappings

The following table lists the recommended Java type for object references to the Objectivity / DB persistence-capable array classes.

Objectivity/DB Object Reference to Array Class	Recommended Java Types
`ooRef(oojArrayOfInt8)`	`byte[]`
`ooRef(oojArrayOfInt16)`	`short[]`
`ooRef(oojArrayOfInt32)`	`int[]`
`ooRef(oojArrayOfInt64)`	`long[]`
`ooRef(oojArrayOfBoolean)`	`boolean[]`
`ooRef(oojArrayOfCharacter)`	`char[]`
`ooRef(oojArrayOfFloat)`	`float[]`
`ooRef(oojArrayOfDouble)`	`double[]`
`ooRef(oojArrayOfObject)` where array elements are of type `oojString`	`String[]`
`ooRef(oojArrayOfObject)` where array elements are of type `oojDate`	`java.util.Date[]` `java.sql.Date[]`

Objectivity/DB Object Reference to Array Class	Recommended Java Types
`ooRef(oojArrayOfObject)` where array elements are of type `oojTime`	`java.sql.Time[]`
`ooRef(oojArrayOfObject)` where array elements are of type `oojTimestamp`	`java.sql.Timestamp[]`
`ooRef(oojArrayOfObject)` where array elements are of type `ooRef(`*`AppClass`*`)` and *`AppClass`* is an application-defined persistence-capable class	*`PCclass`*`[]` where *`PCclass`* is a persistence-capable class that matches the schema description for *`AppClass`* and for which the application uses the schema class name *`AppClass`*.
`ooRef(oojArrayOfObject)` where array elements are of type `ooRef(`*`APIclass`*`)` and *`APIclass`* is a persistence-capable class in the public Objectivity for Java programmer interface (for example, `ooContObj` or `ooMap`)	*`APIclass`*`[]`

Alternative Mappings

A field whose declared Java data type is an interface is represented in the schema by the Objectivity / DB type `ooRef(ooObj)`. As a consequence, you may map an array of object references to `ooObj` to an array of any interface type. Of course, you should only use an interface that is implemented by some persistence-capable class.

The following table lists the Java classes that are allowed mappings for object references to the Objectivity / DB array classes involving references to `ooObj`.

Objectivity/DB Object Reference to Array Class	Allowed Java Types
`ooRef(oojArrayOfObject)` where array elements are of type `ooRef(ooObj)`	*`PCinterface`*`[]` where *`PCinterface`* is an interface (implemented by one or more persistence-capable classes)

Part 2 REFERENCE

20

Predicate Query Language

You perform a predicate query using a *predicate string* with the scan methods of storage objects and to-many relationships. A predicate query finds the objects in a specified group that satisfy a particular condition; the predicate string specifies that condition. Predicate strings use a simple expression language that supports standard operators and constant literals and that has the ability to refer to public fields of persistent objects in Objectivity/DB databases. A predicate string can contain references to fields of the object being tested, literals, and operators that act on the values of those fields and literals.

The predicate query language includes standard arithmetic, relational, and logical operators as well as string matching operators that test whether a string matches regular expressions. The language does not provide any other operators, the ability to declare variables, or the ability to call methods of the scanned class. The language ignores white space and new lines, which serve to separate tokens.

In This Chapter

Object Fields

Within a predicate string, any unquoted sequence of alphanumeric characters (for example, `employeeName`) is interpreted as a field name or the name of a to-one relationship. Predicate strings can refer to an object's persistent fields of the following Java types:

Category	Types

Java primitive type	`char`
	`byte`
	`short`
	`int`
	`long`
	`float`
	`double`
	`boolean`

Java string class	`String` `StringBuffer`

Predicates can test numeric, character, and string values only. To test a field of one of the following types against a literal value, you must specify an *integer* literal.

- `boolean` (use 1 for true, 0 for false)
- `byte`

NOTE Predicates *cannot* refer to fields representing dates and times, fields that reference other persistent objects, to-many relationships, or fields of related objects.

Literals

The query language accepts literals of the following types:

- A character literal is a single-quoted 8-bit character (for example, `'Z'`).
- A string literal is a double-quoted string (for example, `"John Doe"`). When you include a string literal within a Java `String` that contains your predicate, you must use the backslash character (\) before the double-quote characters (") that delimit the string literal. This point is illustrated in "Examples" on page 322.
- An integer literal has the same syntax as in Java (for example, `123`).
- A floating-point literal has the same syntax as in Java (for example, `98.765`).

- There is no support for a unicode literal.

Operators

Objectivity / DB supports the operators listed in the following sections; their precedence is the same as the precedence of the equivalent Java operators. Parentheses (and) can be used to override the normal precedence, as in Java.

Arithmetic Operators

Arithmetic operators produce numeric values; their operands can be numeric literals and fields whose Java types are represented as Objectivity / DB numeric types. (“Default Mapping for Java Types” on page 267 explains how the Java types of persistent fields are mapped to Objectivity / DB types.)

+	Addition, unary plus
-	Subtraction, unary minus
*	Multiplication
/	Division
%	modulus (remainder)

Relational Operators

Relational operators produce boolean values. Equality and inequality operators can compare two expressions of the same supported type. The other operators can compare numeric, character, and string types only.

=, ==	Equality
<>, !=	Inequality
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

String Matching Operators

String matching operators produce `boolean` values. You use string matching operators to compare a string to a pattern. Typically the left operand is a `String` or `StringBuffer` field and the right operand is a string literal containing a regular expression. (To compare exact strings for equality, inequality, and so on, you use the relational operators described in the previous section.)

`=~`	Matches, case sensitive
`!~`	Does not match, case sensitive
`=~~`	Matches, case insensitive
`!~~`	Does not match, case insensitive

NOTE All string-matching operators match the *entire string* in the left operand against the regular expression in the right operand. To match a prefix, suffix, or substring, the pattern must explicitly include wildcard characters at the beginning and / or end; see “Regular Expressions” on page 320.

Logical Operators

Logical operators take boolean operands and return boolean values. Typical operands are expressions that use relational or string matching operators.

`AND,&&`	Conjunction
`OR,||`	Disjunction
`NOT,!&`	Negation

The words `AND`, `OR`, and `NOT` are reserved words in the language; you cannot refer to a field with one of these names in a predicate. You can mix upper and lower case in these reserved words; for example, you can specify the && operator with any of the keywords `AND`, `and`, or `And`.

Regular Expressions

For string comparisons, you can use regular expressions. Objectivity / DB implements its regular expressions based on the POSIX extended regular expression library. Objectivity / DB also defines string matching operators that test whether strings match regular expressions.

In a regular expression, the characters in the following table have special meanings; note that no regular expression matches the newline character. All other characters are literals that match themselves. For example, the character A in a regular expression matches the character A in a string. Comparison of literals is case-insensitive; the character A also matches the character a.

Metacharacter	Description
.	Matches any single character. Loses its special meaning when used within [].
\	Used as a prefix to override any special meaning of the following character. Loses its special meaning when used within []. **Note:** Within a Java string, you must enter \\ to produce a single \ character in your predicate.
[]	Used to bracket a sequence of characters or character ranges; matches any single character in the sequence or in one of the specified ranges. If the first character in the sequence is the caret character (^), this pattern matches any character except the characters in the sequence and the specified ranges. **Note**: Within [], you can use [to match the opening bracket character ([), but you must use \] to match the closing bracket character (]).
-	When used within [], indicates a range of consecutive ASCII characters. For example, [0-5] is equivalent to [012345]. Loses its special meaning if it is the first or last character within [], or the first character after an initial ^. No special meaning when used outside [].
*	Used as a postfix to cause the preceding pattern to be matched zero or more times. Loses its special meaning when used within [].
+	Used as a postfix to cause the preceding pattern to be matched one or more times. Loses its special meaning when used within [].
^	When used as the first character within [], causes the bracketed pattern to match any character not specified within []. When used as the first character of a regular expression, matches the beginning of the string; this use is redundant because a regular expression matches the beginning of the string by default. No special meaning in other locations in a regular expression.
$	When used as the last character of a regular expression, matches the end of the string. No special meaning in other locations in a regular expression.

Metacharacter	Description
( )	Used to group patterns into a single pattern (often used with the | operator).
|	OR operator in regular expressions; when used between two patterns, matches either one of the patterns.

Unlike other languages that match strings against regular expressions, the Objectivity / DB query predicate language matches a regular expression against the *entire* string—as if the regular expression had a ^ inserted at the beginning and a $ at the end. For example, the following patterns are equivalent. They all match strings that begin with the characters "De" and end with the characters "er":

```
De.*er
^De.*er
^De.*er$
De.*er$
```

To match a prefix, suffix, or substring of the left operand, the regular expression must explicitly include wildcard characters.

- To match a prefix, end the pattern with .*. For example, the following pattern matches any string that begins with the characters "fun".

  ```
  fun.*
  ```

- To match a suffix, begin the pattern with .*. For example, the following pattern matches any string that ends with the characters "fun".

  ```
  .*fun
  ```

- To match a substring, begin and end the pattern with .*. For example, the following pattern matches any string that contains the characters "fun".

  ```
  .*fun.*
  ```

Examples

This section illustrates predicates that can be used in queries that retrieve objects of the following Vehicle class.

```
public class Vehicle extends ooObj {
   protected String license;
   protected String type;
   protected int doors;
   protected int transmission;
   protected boolean available;
```

```
   // Transmission values
   public static final int MANUAL = 0;
   public static final int AUTOMATIC = 1;
}
```

These predicates test fields against literal values:

```
license == "ca1234" and type = "H"
type == "J" && available
doors > 4
NOT available
```

These predicates demonstrate *invalid* use of expressions:

```
license == ca1234   // You forgot the quotes around ca1234
doors >* 2          // Unrecognized operator. What is >* ?
```

Using String Literals

If your predicate includes a string literal, the literal must be delimited by double-quote characters ("). To include a double-quote character within a Java string, you must precede it with a backslash character (\). The following Java statement sets the `String` variable `pred` to the first predicate in the preceding example:

```
String pred = "license = \"ca1234\" and type = \"H\"";
```

Using Static Values

If your class uses static fields to define constant values for a persistent field, you may not use those constants directly in a predicate. For example, to scan for vehicles with manual transmission, you cannot use the constant `MANUAL` in a predicate.

You can put the numeric equivalent of the constant in the predicate:

```
transmission == 0
```

Alternatively, you can construct a predicate using the constant:

```
String pred = new String("transmission = " +
                         String.valueOf(Vehicle.MANUAL));
```

Testing Boolean Fields

The easiest way to test a boolean field is to use the field name itself to test that the field is true, and the field name preceded by the `NOT` operator to test that it is false.

This predicate tests for vehicles that are available:

```
available
```

This predicate tests for vehicles that are not available (that is, vehicles that have been rented):

```
NOT available
```

If you chose to specify a literal value for a boolean field, remember that you may not use the Java values `true` and `false`; instead, you must use the integer values `1` and `0`.

Thus, these two predicates are equivalent:

```
available
available = 1
```

These two predicates are equivalent:

```
NOT available
available = 0
```

Using Regular Expressions

The following predicates use regular expressions in string comparisons.

```
license =~ ".*1.99.$"   // anyString 1 anyChar 99 anyChar
license !~~ "1.*99"     // 1 anyString 99 anyString
license =~ "ca[0-9]+"   // ca oneOrMoreDigits anyString
license =~ "(abc | def)"
                 // A string beginning with either abc or def
```

Because a regular expression matches the whole string, the following predicates are equivalent ways to test for a license that begins with the characters `"ca"`:

```
license =~ "ca.*"
license =~ "^ca.*"
```

21

Objectivity/DB Data Types

The schema of a federated database specifies the Objectivity / DB type of every persistent field of every persistence-capable class whose objects can be stored in the federated database. When an application reads or writes a persistent object, Objectivity / DB automatically maps data between its own data types and the data types native to the application.

This chapter describes the language-independent data types used in the schema of an Objectivity / DB federated database. Chapter 14, “Schema Management,” explains how a class description in the schema can be generated from a Java class declaration. Chapter 19, “Schema Matching for Interoperability,” explains how to define a Java class corresponding to an existing class description in the schema.

In This Chapter

Data in the Federated Database

A federated database contains persistent objects stored in containers in its component databases. The data for a persistent object consists of a value for each of its persistent fields.

Objectivity / DB stores field values in the federated database using language-independent types for numeric, character, and string scalars, scalars related to date and time, scalar references to persistent objects, and arrays of most of those scalar types. A given Objectivity / DB type may be used to store values of one or more Java types, one or more C++ types, one or more Smalltalk types, and one or more SQL types.

If the declared data type of a Java field is an application-defined persistence-capable class or a persistence-capable class in the public Objectivity for Java programmer interface, its value is stored as an object reference. Fields of most other supported Java types can have values of two alternative forms:

- An object reference to an internal persistent object of some internal Objectivity / DB persistence-capable class.

  Object references maintain object identity and can represent missing data (the absence of any value).

- A value of a primitive type character, integer, or floating-point type, or an instance of an internal Objectivity / DB non-persistence-capable class.

  These types do not maintain identity of values; most of them cannot represent missing data.

Object Identity

Objectivity / DB can uniquely identify any object of a persistence-capable class; it cannot uniquely identify a value of a primitive data type or an instance of an Objectivity / DB non-persistence-capable class. This difference becomes apparent when values are transferred between the federated database and an application: persistence-capable classes preserve object identity and other data types do not.

For example, if an array of integers (of the Java type `int[]`) is mapped by the schema to the array class `oojArrayOfInt32`, all objects in memory that reference the same array will be mapped to persistent objects that reference the same array in the federated database. Any modification to that array will affect all persistent objects in memory that reference the array; when you commit the transaction that modified the array, the change will affect all persistent objects in the federated database that reference the corresponding `oojArrayOfInt32`. On the other hand, if the Java array is mapped by the schema to the Objectivity / DB non-persistence-capable class `ooVArray(int32)`, different persistent objects in memory that reference the array will lose the identity of this array when they are stored in the database and later retrieved. Each retrieved persistent object will

reference a different array, and a change to the array referenced by one retrieved object will not affect the array referenced by a different retrieved object.

Missing Data

Missing data for a particular field may be indicated by a null value in that field; null indicates the absence of any value. All persistence-capable classes use a null object reference to represent missing data. Some other types have a representation for null and others do not. For example, the non-persistence-capable class `ooSQLnull_date` can represent the absence of any date value, but the primitive type `int16` cannot represent the absence of any integer value.

Object-Reference Types

The following table describes the Objectivity / DB types that store references to persistent objects.

Objectivity/DB Reference Type	Description
`ooRef(`*`PCclass`*`)`, where *`PCclass`* is a persistence-capable class.	Reference to a persistent object of class *`PCclass`*; identifies the referenced object with a full (64-bit) object identifier (OID)
`ooShortRef(`*`PCclass`*`)`, where *`PCclass`* is a persistence-capable class.	Reference to a persistent object of class *`PCclass`*; identifies the referenced object with a short (32-bit) OID

Objectivity / DB provides two alternative ways to store a reference to a persistent object of a given class. The `ooRef` types identify the referenced object by a full 64-bit identifier, which specifies the object's database, container, page within the container, and slot on the page. The `ooShortRef` types identify the referenced object by a 32-bit identifier, which specifies only the page within the container and the slot number on the page. The referenced object is assumed to be in the same container and database as the referencing object.

Numeric and Character Types

The following table describes the Objectivity / DB primitive types used to store numeric and character values.

Objectivity/DB Primitive Type	Description of Value
`int8`	8-bit signed integer
`int16`	16-bit signed integer
`int32`	32-bit signed integer
`int64`	64-bit signed integer
`uint8`	8-bit unsigned integer or Boolean value
`uint16`	16-bit unsigned integer or character
`uint32`	32-bit unsigned integer
`uint64`	64-bit unsigned integer
`float32`	Floating-point number
`float64`	Double-precision floating-point number

String Classes

A character string may be stored with either of two Objectivity / DB non-persistence-capable string classes: `ooVString` or `ooUTF8String`. The difference is that `ooVString` is intended for strings of ASCII characters only. If a string with Unicode characters is stored as a `ooVString` and then retrieved by an application, the retrieved string is not guaranteed to be rendered correctly. In contrast, `ooUTF8String` can represent Unicode characters that will be rendered correctly on all platforms supporting Unicode.

Objectivity/DB Non-Persistence-Capable Class	Description of Value
`ooVString`	Variable-length string of ASCII character
`ooUTF8String`	Variable-length string of Unicode characters

Objectivity / DB also has a persistence-capable string class `oojString`, which is used *only* for the elements of a string array.

Objectivity/DB Persistence-Capable Class	Description of Value
`oojString`	Internal persistent object wrapping a string of type `ooUTF8String` **Note:** This type is used *only* for the elements of an array of the class `oojArrayOfObject` corresponding to a Java array of type `String[]`.

WARNING Objectivity / C++ applications can define special non-persistence-capable optimized string classes named `ooString_`*N* where *N* is the number of characters for which strings of the class are optimized. Java applications cannot access any field whose Objectivity / DB type is `ooString_`*N*.

Date and Time Classes

The following table describes the Objectivity / DB non-persistence-capable classes used to store date and time values.

Objectivity/DB Non-Persistence-Capable Class	Description of Value
`ooSQLdate`	Calendar date; no representation of null
`ooSQLnull_date`	Calendar date; can represent null
`ooSQLtime`	Clock time; no representation of null
`ooSQLnull_time`	Clock time; can represent null
`ooSQLtimestamp`	A point in time to the nearest millisecond; no representation of null
`ooSQLnull_timestamp`	A point in time to the nearest millisecond; can represent null

The following table describes the Objectivity / DB persistence-capable classes used to store date and time values.

Objectivity/DB Persistence-Capable Class	Description of Value
`oojDate`	Internal persistent object wrapping calendar date
`oojTime`	Internal persistent object wrapping clock time
`oojTimestamp`	Internal persistent object wrapping a point in time to the nearest nanosecond

Objectivity / DB has three alternative types for storing date / time values:

- Non-persistence-capable classes whose names begin with the `ooSQL` prefix; these types cannot represent null values.
- Non-persistence-capable classes whose names begin with the `ooSQLnull_` prefix; these types can represent null values.
- Persistence-capable classes whose names begin with the `ooj` prefix; like all classes, these types preserve object identity and can represent null values. Objects of these types are stored more efficiently than values of the two alternative structure types.

  Note that the persistence-capable class used to store a timestamp is more precise than the two alternative classes; the persistence-capable class specifies the time to the nanosecond whereas the two non-persistence-capable classes express the time to the millisecond.

Array Classes

The following table describes the Objectivity / DB non-persistence-capable array types.

Objectivity/DB Non-Persistence-Capable Array Class	Description of Value
`ooVArray(int8)`	Variable-length array of elements of type `int8`
`ooVArray(int16)`	Variable-length array of elements of type `int16`
`ooVArray(int32)`	Variable-length array of elements of type `int32`

Objectivity/DB Non-Persistence-Capable Array Class	Description of Value
`ooVArray(int64)`	Variable-length array of elements of type `int64`
`ooVArray(uint8)`	Variable-length array of elements of type `uint8`
`ooVArray(uint16)`	Variable-length array of elements of type `uint16`
`ooVArray(float32)`	Variable-length array of elements of type `float32`
`ooVArray(float64)`	Variable-length array of elements of type `float64`
`ooVArray(ooRef(`*`PCclass`*`))`, where *`PCclass`* is a persistence-capable class.	Variable-length array of elements of type `ooRef(`*`PCclass`*`)`
`ooVArray(ooShortRef(`*`PCclass`*`))`, where *`PCclass`* is a persistence-capable class.	Variable-length array of elements of type `ooShortRef(`*`PCclass`*`)`

The following table describes the Objectivity / DB persistence-capable array classes.

Objectivity/DB Array Class	Description
`oojArrayOfInt8`	Internal persistent object wrapping variable-length array of elements of type `int8`
`oojArrayOfInt16`	Internal persistent object wrapping variable-length array of elements of type `int16`
`oojArrayOfInt32`	Internal persistent object wrapping variable-length array of elements of type `int32`
`oojArrayOfInt64`	Internal persistent object wrapping variable-length array of elements of type `int64`
`oojArrayOfFloat`	Internal persistent object wrapping variable-length array of elements of type `float32`
`oojArrayOfDouble`	Internal persistent object wrapping variable-length array of elements of type `float64`
`oojArrayOfObject`	Internal persistent object wrapping variable-length array of object references to objects of the same class

Part 3 PROGRAMMING

22

Exceptions

Exceptions defined by the Objectivity for Java programming interface are contained in the `com.objy.db` package. Checked exceptions are derived from `ObjyException`. Unchecked exceptions are derived from `ObjyRuntimeException`.

Exceptions occur for three main reasons:

- You have used the programming interface incorrectly. For example, locking a basic object that is not persistent throws an `ObjectNotPersistentException`, and calling `begin` on a session that was already open throws a `TransactionInProgressException`.

  It is up to you to decide whether to catch exceptions resulting from programming errors or let the program terminate and correct the program logic.
- A method was, for some reason, unable to do what it was trying to do. An example would be failing to obtain a lock because of a conflicting lock. In this case, the method would throw a `LockNotGrantedException`.
- An error, such as a resource failure, has occurred within Objectivity / DB and it must asynchronously terminate a transaction. This kind of situation would cause a `TransactionAbortedException`.

In This Chapter

Exception Information Objects

Exceptions thrown by Objectivity for Java may arise from errors within the Objectivity / DB kernel, or from errors detected by the programming interface. If the error originated in the Objectivity / DB kernel, the exception will have an associated vector of exception information objects; exception information objects implement the `ExceptionInfo` interface. If the errors were detected by the programming interface, the vector will be null.

The exception classes provide two methods relevant to exception information objects: `reportErrors` prints the ID and description of any kernel errors, and `errors` retrieves the vector of exception information objects. The order of the exception information objects in the vector is the order in which the Objectivity / DB kernel reported the errors.

Each exception information object has an error identifier, an error level, and a message describing the error. The error identifier is unique to each error, while the level number merely distinguishes between categories of error, such as fatal errors compared to warnings.

Examples

The following example catches a number of exceptions that could be thrown while scanning a container for objects that satisfy a predicate expression:

- A container's scan method automatically requests a read lock on the container. If the session cannot obtain the read lock, the method would throw a `LockNotGrantedException`.
- If the predicate supplied to the scan method is invalid, the method will throw an `ObjyRuntimeException`. The developer has the option of modifying the application to ensure that the predicate is always valid, or simply catching the exception and providing information about the exception back to whomever supplied the predicate.
- If lock waiting is enabled and a deadlock occurs, the method will throw a `TransactionAbortedException`.

EXAMPLE This example catches any of these exceptions with a general handler for the exception superclass `ObjyRuntimeException`.

```
session.begin();
Iterator itr;
try {
   itr = classAContainer.scan("ClassA", predicate);
} catch (ObjyRuntimeException e) {
   e.printErrors();
   session.abort();
}
session.commit();
```

This example illustrates how to retrieve and print out the properties of the exception information objects of the `ObjyRuntimeException` thrown by the `scan` method in the previous example.

```
session.begin();
Iterator itr;
try {
   itr = classAContainer.scan("ClassA", predicate);
} catch (ObjyRuntimeException e) {
   // Get Vector of exception information objects
   Vector errors = e.errors();
   // Make sure there are exception information objects
   if (errors != null) {
      // Get Enumeration from Vector
      Enumeration errs = errors.elements();
      ExceptionInfo ei;
         while (errs.hasMoreElements()) {
         ei = (ExceptionInfo)errs.nextElement();
         System.out.println("Id: " + ei.getId());
         System.out.println("Level: " + ei.getLevel());
         System.out.println("Message: " + ei.getMessage());
         }
   }
   session.abort();
   return;
}
session.commit();
```

23

Getting Started

This chapter contains the source code for the programming example presented in Chapter 1, “Getting Started”.

In This Chapter

Example

The example application is intended to be used by the agents and managers of a vehicle rental company. The operations supported are:

- Add a vehicle to the rental company’s fleet of vehicles.
- Delete a vehicle from the fleet.
- List:
  - All the vehicles in the fleet.
  - Only the vehicles that satisfy a predicate.
- Rent a vehicle.
- Return a rented vehicle.

The application is implemented by four classes:

- `Vehicle` (see page 343), which implements a vehicle that can be rented and returned.
- `Fleet` (see page 341), which implements a fleet of vehicles.
- `VrcInit` (see page 361), which initializes the rental company database.

- `Vrc` (see page 348), which implements the interactive application for accessing the database of the vehicle rental company.

To execute this example, you need to:

1. Compile the files `Vehicle.java`, `Fleet.java`, `VrcInit.java`, and `Vrc.java` located in the `GettingStarted` subdirectory of the programming samples directory. See the *Installation and Platform Notes* for your operating system for the location of the samples directory for your platform.
2. Start an Objectivity / DB lock server.
3. Create a federated database called `Vrc` in the `GettingStarted` sample directory.
4. Execute `VrcInit` to initialize the federated database.

See the Objectivity/DB administration book for information on the tools used to create a federated database and start a lockserver.

Fleet.java

```
////////////////////////////////////////////////////////////////////////
//
// Fleet - a fleet of rental vehicles
// An array of reference in a persistent field links a fleet to its
// vehicles.
//
// Field access methods hide the array implementation.
//
// Copyright (c) 1998 Objectivity, Inc.  All Rights Reserved
//
// UNPUBLISHED PROPRIETARY SOURCE CODE OF OBJECTIVITY, INC.
//
// The copyright notice above does not evidence any actual
// or intended publication of such source code.
//
////////////////////////////////////////////////////////////////////////

import com.objy.db.*;     //Import Objectivity for Java exceptions
import com.objy.db.app.*; //Import Objectivity for Java classes
import java.util.NoSuchElementException;

public class Fleet extends ooObj {
    static final int FLEET_SIZE = 1000;

    // Persistent fields
    protected Vehicle[] vehicles  = new Vehicle[FLEET_SIZE];
    protected int numberOfVehicles;

    // Constructor
    public Fleet() {
        this.numberOfVehicles = 0;
    }

    // All access methods must be called during a transaction

    // Field access methods
    public int getNumVehicles() {
        fetch();
        return this.numberOfVehicles;
    }

    public void addVehicle(Vehicle newMember) {
        markModified();
        if (findVehicle(newMember.getLicense()) == null) {
```

```
                this.vehicles[this.numberOfVehicles] = newMember;
                this.numberOfVehicles++;
            }
        }

        public void deleteVehicle(Vehicle vehicle) {
            markModified();
            int i = 0;
            while ((i < this.numberOfVehicles) &&
                   (this.vehicles[i] != vehicle)) {
                i++;
            }
            if (i != this.numberOfVehicles) {
                // Vehicle was found; remove it
                for (int j = i + 1; j < this.numberOfVehicles; j++, i++) {
                    this.vehicles[i] = this.vehicles[j];
                }
                this.vehicles[this.numberOfVehicles-1] = null ;
                this.numberOfVehicles--;
            }
        }

        public Vehicle findVehicle(String license) {
            fetch();
            if (this.numberOfVehicles == 0) {
                return null;
            }
            for (int i = 0; i < this.numberOfVehicles; i++) {
                if ((this.vehicles[i].getLicense()).equals(license)) {
                    return this.vehicles[i];
                }
            }
            return null;
        }

    } // End Fleet class
```

Vehicle.java

```
////////////////////////////////////////////////////////////////////////
//
// Vehicle - vehicle in a rental fleet
// A reference in a persistent field links a vehicle to its fleet.
//
// Copyright (c) 1998 Objectivity, Inc.  All Rights Reserved
//
// UNPUBLISHED PROPRIETARY SOURCE CODE OF OBJECTIVITY, INC.
//
// The copyright notice above does not evidence any actual
// or intended publication of such source code.
//
////////////////////////////////////////////////////////////////////////

import com.objy.db.*;     //Import Objectivity for Java exceptions
import com.objy.db.app.*; //Import Objectivity for Java classes
import com.objy.db.iapp.*;
import java.util.*;

public class Vehicle extends ooObj {
    // Persistent fields
    protected String license;
    protected String type;
    protected int doors;
    protected int transmission;
    protected boolean available;
    protected Fleet fleet;

    // Legal values for transmission field
    public static final int MANUAL = 0;
    public static final int AUTOMATIC = 1;

    // Transient field
    protected transient int dailyRate;

    // Constructor
    public Vehicle(String license, String type,
                   int doors, int transmission, int rate) {
        this.license = license;
        this.type = type;
        this.doors = doors;
        this.transmission = transmission;
        this.available = true;
        this.dailyRate = rate;
```

```
}

// All access methods must be called during a transaction

// Field access methods to get persistent field values
public String getLicense() {
    fetch();
    return this.license;
}

public String getType() {
    fetch();
    return this.type;
}

public int getDoors() {
    fetch();
    return this.doors;
}

public int getTransmission() {
    fetch();
    return this.transmission;
}

public boolean isAvailable() {
    fetch();
    return this.available;
}

public Fleet getFleet() {
    fetch();
    return this.fleet;
}

// Field access methods to set fleet field
public void setFleet(Fleet fleet) {
    markModified();
    this.fleet = fleet;
}

// Field access method to set available field
public void rentVehicle() {
    markModified();
    this.available = false;
}
```

```
public void returnVehicle() {
    markModified();
    this.available = true;
}

// Utility to print description of new vehicle
public String toShortString() {
    // This method must be called during a transaction
    fetch();
    StringBuffer buffer = new StringBuffer();
    buffer.append("License:" + license);
    buffer.append(" Class:" + type);
    buffer.append(" Doors:" + doors);
    if (transmission == MANUAL)
        buffer.append(" MANUAL");
    else
    buffer.append(" AUTOMATIC");
    buffer.append(" Rate:" + dailyRate);
    String strng = new String(buffer);
    return strng;
}

// Utility to print description of this vehicle
public String toString() {
    // This method must be called during a transaction
    fetch();
    StringBuffer buffer = new StringBuffer();
    if (available)
        buffer.append("AVAILABLE");
    else
        buffer.append("RENTED");
    buffer.append(" License:" + license);
    buffer.append(" Class:" + type);
    buffer.append(" Doors:" + doors);
    if (transmission == MANUAL)
        buffer.append(" MANUAL");
    else
    buffer.append(" AUTOMATIC");
    buffer.append(" Rate:" + dailyRate);
    String strng = new String(buffer);
    return strng;
}

// Static utility to obtain properties maintained by
// Objectivity for Java
public static void printInfo(Vehicle vehicle) {
    // This method must be called during a transaction
```

```
        // Get vehicle's container
        ooContObj cont = vehicle.getContainer();
        // Get vehicle's database
        ooDBObj db = cont.getDB();
        // Get vehicle's federated database
        ooFDObj fd = db.getFD();
        // Get vehicle's session
        Session session = vehicle.getSession();
        // Get vehicle's object identifier
        ooId oid = vehicle.getOid();

        // Print the location in the storage hierarchy
        System.out.println("Vehicle " + vehicle.getLicense() +
                           " has OID " + oid.getStoreString() +
                           " and is stored in:");
        String nullStr = "";
        if (cont.getName() == nullStr) {
            System.out.println("    an unnamed container in");
        }
        else {
          System.out.println("   container " + cont.getName() + " in");
        }
        System.out.println("        database " + db.getName() + " in");
      System.out.println("         federated database " + fd.getName());

        // Print information about the session
        String access, txType;
        if (session.getOpenMode() == oo.openReadOnly) {
            access = "read only";
        }
        else {
            access = "read/write";
        }
        if (session.getMrowMode() == oo.MROW) {
            txType = "MROW";
        }
        else {
            txType = "nonMROW";
        }
        System.out.println("The session is open for " +
                           access + " access by " +
                           txType + "transactions");
    }

    public void activate(ActivateInfo activateInfo) {
        // Handle fetch errors
```

```
            if (activateInfo.hasFetchErrors()) {
                // Mark modified so that new values of persistent fields
                // are written to the database.
                markModified();
                // Retrieve vector of fetch errors
                Vector errors = activateInfo.getFetchErrors();
                Enumeration errs = errors.elements();
                while (errs.hasMoreElements()) {
                    FetchErrorInfo ffi =
                        (FetchErrorInfo)errs.nextElement();
                    // Get components of fetch failed information object
                    // and print error message.
                    String fieldName = ffi.getFieldName();
                    System.out.println(ffi.getErrorMessage());
                    // Set persistent field at which error occurred
                    if (fieldName.equals("fleet")) {
                        try {
                            fleet =
                            (Fleet)getContainer().getDB().lookup("Fleet");
                        }
                        catch(ObjectNameNotFoundException e) {
                            System.out.println
                            ("\nCouldn't set fleet for vehicle: "
                            + toString());
                        }
                    }
                }
            }
            // Set default transient field
            dailyRate = doors * 10;
        }
    } // End Vehicle class
```

Vrc.java

```
//////////////////////////////////////////////////////////////////////
//
// Vrc - implements the interactive application for accessing the
// database of the vehicle rental company. This class:
// * Opens a connection to a federated database called "Vrc" for
//   read/write access.
// * Creates a session and gets its associated federated database.
// * Retrieves a database called "VehiclesDB".
// * Retrieves a container called "VehiclesContainer".
// * Accepts requests for add, delete, list, rent
//   and return operations on the fleet.
// * Dispatches the request to the appropriate method.
// * If the request is null, closes the database and exits.
// * Closes the federated database.
//
// Copyright (c) 1998 Objectivity, Inc.  All Rights Reserved
//
// UNPUBLISHED PROPRIETARY SOURCE CODE OF OBJECTIVITY, INC.
//
// The copyright notice above does not evidence any actual
// or intended publication of such source code.
//
//////////////////////////////////////////////////////////////////////

import com.objy.db.*;     //Import Objectivity for Java exceptions
import com.objy.db.app.*; //Import Objectivity for Java classes
import java.io.IOException;
import java.io.BufferedReader;
import java.io.InputStreamReader;
public class Vrc {
    Connection connection;
    Session session;
    ooFDObj vrcFD;
    ooDBObj vehiclesDB;
    ooContObj vehiclesContainer;
    Fleet fleet;
    Vehicle vehicle;
    Iterator itr;

    static public void main(String args[]) {
        Vrc vrc = new Vrc();
        vrc.initialize();
        vrc.handleRequest();
    }
```

```
public void initialize() {
    // Open a connection to a federated database
    try {
        connection = Connection.open("Vrc", oo.openReadWrite);
    } catch (DatabaseNotFoundException exception) {
       System.out.println("\nFederated Database \"Vrc\" not found" +
                    " - use oonewfd to create federated database.");
        return;
    } catch (DatabaseOpenException exception) {
        System.out.println("\nConnection to federated database" +
                           " \"Vrc\" already open.");
        return;
    }

    // Create session
    session = new Session();
    // Get federated database from session
    vrcFD = session.getFD();

    // Get the vehicles database
    session.begin();
    if (vrcFD.hasDB("VehiclesDB")) {
        vehiclesDB = vrcFD.lookupDB("VehiclesDB");
        session.commit();
    }
    else {
        System.out.println("\nDatabase \"VehiclesDB\" not found" +
                           " - run VrcInit.");
        session.abort();
        return;
    }
    // Get the fleet
    session.begin();
    try {
        fleet = (Fleet)vehiclesDB.lookup("Fleet");
        session.commit();
    } catch (ObjectNameNotFoundException e4) {
        System.out.println("Could not find fleet named root.");
        session.abort();
        return;
    }
    // Get the vehicles container
    session.begin();
    if (vehiclesDB.hasContainer("VehiclesContainer")) {
        vehiclesContainer =
```

```
                vehiclesDB.lookupContainer("VehiclesContainer");
            }
            else {
                System.out.println("\nContainer \"VehiclesContainer\" " +
                                   "not found" + " - run VrcInit.");
                session.abort();
                return;
            }
            session.commit();

        }

        public void handleRequest() {
            String request;
            if (vrcFD == null || vehiclesDB == null
                || fleet == null || vehiclesContainer == null)
                return;
            // Loop to get request
            System.out.println("\n--- WELCOME TO THE OBJECTIVITY " +
                               "CAR RENTAL DEMO PROGRAM ---");
            while ( (request = getString("\nEnter request" +
                      " (add, delete, list, rent, return, quit)")) != null) {
            // Dispatch to appropriate method
                if (request.equalsIgnoreCase("add") ||
                            request.equalsIgnoreCase("a"))
                    addVehicle();
                else if (request.equalsIgnoreCase("delete") ||
                            request.equalsIgnoreCase("d"))
                    deleteVehicle();
                else if (request.equalsIgnoreCase("list") ||
                            request.equalsIgnoreCase("l"))
                    listVehicles();
                else if (request.equalsIgnoreCase("rent") ||
                            request.equalsIgnoreCase("ren"))
                    rentVehicle();
                else if (request.equalsIgnoreCase("return") ||
                            request.equalsIgnoreCase("ret"))
                    returnVehicle();
                else if (request.equalsIgnoreCase("quit") ||
                            request.equalsIgnoreCase("q"))
                    break;
                else System.out.println("\nPlease enter a valid request.");
            }
            // Terminate the session and close the federated database
            // when the request is null
            try {
                connection.close();
```

```
        } catch (DatabaseClosedException e) {
            System.out.println("\nConnection already closed.");
            System.out.println("\nGoodbye!");
            return;
        }
        System.out.println("\nGoodbye!");
    }

    public void quit() {
        // Close connection
        try {
            connection.close();
        } catch (DatabaseClosedException exception) {
            System.out.println("Connection already closed.");
        }
    }

    public int numberOfVehicles() {
        session.begin();
        int number = fleet.getNumVehicles();
        session.commit();
        return number;
    }
    public Iterator scanVehicles(String predicate) {
        Iterator itr;
        try {
            itr = vehiclesContainer.scan("Vehicle", predicate);
        } catch (ObjyRuntimeException e) {
            e.reportErrors();
            return null;
        }
        if (!itr.hasNext()) {
            System.out.println("\nVehicle with: " + predicate +
                                " not found.");
            return null;
        }
        else
            return itr;
    }

    public void addVehicle() {
        String input = "";
        String license = "";
        String type = "";
        int doors = 0;
        int trans = 0;
        int rate = 0;
```

```
boolean gotLicense = false;
while (gotLicense == false) {
    // Loop to get license
    while ( (input = getString("Enter license: " )) == null );
    license = input.toUpperCase();
    String predicate = new String("license == \"" +
                                    license + "\"");
    session.begin();
    // Check to see if this license already exists
    try {
        itr = vehiclesContainer.scan("Vehicle", predicate);
    } catch (ObjyRuntimeException e) {
        e.reportErrors();
        session.abort();
        return;
    }
    if(itr.hasNext()) {
        System.out.println("\nVehicle with license: " +
                            license + " in this database.");
    }
    else
        gotLicense = true;
    session.commit();
}

// Get vehicle class
boolean gotClass = false;
while ( !gotClass ) {
    while ( (input = getString("Enter class: ")) == null );
    type = input.toUpperCase();
    gotClass = true;
}

// Get doors
boolean gotDoors = false;
while (!gotDoors) {
 while ( (input = getString("Enter number of doors: ")) == null);
    if ((doors = getInt(input)) != 0) {
        gotDoors = true;
    }
}

// Get transmission type
boolean gotTrans = false;
while (!gotTrans) {
    while ( (input = getString("Enter transmission type" +
```

```
                                " - manual or automatic: ")) == null);
            if (input.equalsIgnoreCase("manual")
                                    || input.equalsIgnoreCase("m")) {
                trans = Vehicle.MANUAL;
                gotTrans = true;
            }
            else if (input.equalsIgnoreCase("automatic")
                    || input.equalsIgnoreCase("a")
                    || input.equalsIgnoreCase("auto")) {
                trans = Vehicle.AUTOMATIC;
                gotTrans = true;
            }
        }

        // Get rate
        boolean gotRate = false;
        while (!gotRate) {
            while ( (input = getString("Enter daily rate: ")) == null);
            if ((rate = getInt(input)) != 0) {
                gotRate = true;
            }
        }

        session.begin();
        vehicle = new Vehicle(license, type, doors, trans, rate);
        try {
            vehiclesContainer.cluster(vehicle);
        } catch (LockNotGrantedException e) {
            e.reportErrors();
            session.abort();
            return;
        }
        fleet.addVehicle(vehicle);
        System.out.println("\nAdded Vehicle: " + vehicle.toString());
        session.commit();
    }

    public void deleteVehicle() {
        // Scan for and return vehicle that matches license
        String input = "", license = "";
        boolean gotLicense = false;
        while (gotLicense == false) {
            // Loop to get license
            while ( (input = getString("Enter license: " )) == null );
            gotLicense = true;
            license = input.toUpperCase();
        }
```

```
        String predicate = new String("license = \"" + license + "\"");
        System.out.println("\nLooking for vehicle with license: "
                           + license);

        session.begin();
        Iterator itr = scanVehicles(predicate);
        if (itr == null) {
            session.abort();
            return;
        }
        while(itr.hasNext()) {
            vehicle = (Vehicle)itr.next();
            if (!vehicle.isAvailable()) {
             System.out.println("\nCannot delete vehicle with license: "
                                + license + ", it is rented.");
                session.abort();
                return;
            }
            System.out.println("\nDeleted: " + vehicle.toString() );
            fleet.deleteVehicle(vehicle);
            vehicle.delete();
            break;
        }
        session.commit();
    }

    public void rentVehicle() {
        // Scan for and return vehicle that matches license
        String input = "", license = "";
        boolean gotLicense = false;
        while (gotLicense == false) {
            // Loop to get license
            while ( (input = getString("Enter license: " )) == null );
            gotLicense = true;
            license = input.toUpperCase();
        }
        String predicate = new String("license = \"" + license + "\"");
        System.out.println("\nLooking for vehicle with license: "
                           + license);
        session.begin();
        Iterator itr = scanVehicles(predicate);
        if (itr == null) {
            session.abort();
            return;
        }
        while(itr.hasNext()) {
            vehicle = (Vehicle)itr.next();
```

```
            if (!vehicle.isAvailable()) {
                System.out.println("\nVehicle with license: " +
                                   license + " is already rented.");
                session.abort();
                return;
            }
            vehicle.rentVehicle();
           System.out.println("\nRented: " + vehicle.toShortString() );
            break;
        }
        session.commit();
    }

    public void returnVehicle() {
        // Scan for and rent first vehicle that matches license
        String input = "", license = "";
        boolean gotLicense = false;
        while (gotLicense == false) {
            // Loop to get license
            while ( (input = getString("Enter license: " )) == null );
            license = input.toUpperCase();
            gotLicense = true;
        }
        String predicate = new String("license = \"" + license + "\"");
        System.out.println("\nLooking for vehicle with license: "
                           + license);
        session.begin();
        Iterator itr = scanVehicles(predicate);
        if (itr == null) {
            session.abort();
            return;
        }
        while(itr.hasNext()) {
            vehicle = (Vehicle)itr.next();
            if (vehicle.isAvailable()) {
               System.out.println("\nVehicle with license: " + license +
                                  " is not rented.");
                session.abort();
                return;
            }
            vehicle.returnVehicle();
           System.out.println("\nReturned: " + vehicle.toShortString());
            break;
        }
        session.commit();
    }
```

```
public void listVehicles() {
    // List vehicles by predicate
    String field = "";
    String predicate = "";
    String userPredicate = "";
    StringBuffer buffer = new StringBuffer();
    StringBuffer userBuffer = new StringBuffer();

    // Get field to scan on
    boolean listAll = false;
    boolean gotField = false;
    boolean getField = false;
    boolean first = true;
    String input;
    String searchPrompt = "\nDo you want to enter a search field?";
    String fieldPrompt = "\nEnter field to search \n" +
             " (license, class, doors, transmission, or available)";
    while (!getField) {
        while ( ( input = getString(searchPrompt)) == null);
            if (input.equalsIgnoreCase("no")
                || input.equalsIgnoreCase("n")) {
                getField = true;
                if (first)
                    listAll = true;
                break;
            }
       searchPrompt = "\nDo you want to enter another search field?";
        while (!gotField) {
            while ( (input = getString(fieldPrompt)) == null );
            if (input.equalsIgnoreCase("license")
                || input.equalsIgnoreCase("l")) {
                field = new String("license");
                gotField = true;
            }
            else if (input.equalsIgnoreCase("class")
                || input.equalsIgnoreCase("c")) {
                field = new String("type");
                gotField = true;
            }
            else if (input.equalsIgnoreCase("doors")
                || input.equalsIgnoreCase("d")) {
                field = new String("doors");
                gotField = true;
            }
            else if (input.equalsIgnoreCase("transmission")
                || input.equalsIgnoreCase("trans")
                || input.equalsIgnoreCase("t")) {
```

```
            field = new String("transmission");
            gotField = true;
        }
        else if (input.equalsIgnoreCase("available")
            || input.equalsIgnoreCase("a")) {
            field = new String("available");
            gotField = true;
        }
        else if (input.equalsIgnoreCase("q")) {
            break;
        }
    }
    if (!gotField)
        return;

    gotField = false;
    // Get value to search for
    String value = "";
    int intValue = 0;
    String fieldValue = "";
    if (field.equals("transmission"))
        fieldValue = " (manual or automatic)";
    else if (field.equals("license") || field.equals("type"))
        fieldValue = " (a string)";
    else if (field.equals("available"))
       fieldValue = " (true or false)";
    else
       fieldValue = " (an integer)";
    while ( (value = getString("\nWhat value of "
                               + field.toUpperCase()
                               + fieldValue +
                         " do you want to search for? ")) == null);
    value = value.toUpperCase();

    if (!first) {
        buffer.append(" && ");
        userBuffer.append(" && ");
    }
    else
        first = false;

    // Field is a string
    if (field.equals("license") || field.equals("type")) {
        buffer.append(field + " = \"" + value + "\"");
        userBuffer.append(field + " = \"" + value + "\"");
    }
    else {
```

```
 // Field is a constant int
// Convert Java constant to Objectivity/DB representation
 if (field.equals("transmission")) {
     boolean gotTrans = false;
     while (!gotTrans) {
         if ((intValue = getTrans(value)) != -1) {
             gotTrans = true;
             break;
         }
         while ( (value = getString("\nWhat value of "
                         + field.toUpperCase()
                         + fieldValue +
                " do you want to search for? ")) == null);
     }
 }
 // Field is a boolean
// Convert Java boolean to Objectivity/DB representation
 else if (field.equals("available")) {
     boolean gotBool = false;
     while (!gotBool) {
         if ((intValue = getBool(value)) != -1) {
             gotBool = true;
             break;
         }
         while ( (value = getString("\nWhat value of "
                         + field.toUpperCase()
                         + fieldValue +
                " do you want to search for? ")) == null);
         value = value.toUpperCase();
     }
 }
 // Field is an integer
 // Check that it's valid
 else if ( field.equals("doors")) {
     boolean gotInt = false;
     while (!gotInt) {
         if ((intValue = getInt(value)) != 0) {
             gotInt = true;
             break;
         }
         while ( (value = getString("\nWhat value of "
                         + field.toUpperCase()
                         + fieldValue +
                " do you want to search for? ")) == null);
     }
 }
 userBuffer.append(field + " = " + value);
```

```
                buffer.append(field + " = " + intValue);
            }
        }

        if (!listAll) {
            predicate = new String(buffer);
            userPredicate = new String(userBuffer);
            System.out.println("\nSearching for vehicles with "
                               + userPredicate + "\n");
        }
        else
            System.out.println("\nSearching for ALL vehicles.\n" );

        session.begin();
        Iterator itr = scanVehicles(predicate);
        if (itr == null) {
            session.abort();
            return;
        }
        while(itr.hasNext()) {
            vehicle = (Vehicle)itr.next();
                System.out.println(vehicle.toString());
        }
        session.commit();
    }


    public String getString(String prompt) {
        String  line = "";

        System.out.println(prompt);
        BufferedReader in = new
                    BufferedReader(new InputStreamReader(System.in));
        try { line = in.readLine(); }
        catch(java.io.IOException e) {
            System.out.println("\nError reading input, exiting...");
            return null;
        }
        if (line == null)
            return null;
        else if (line.length() == 0)
            return null;

        return line;
    }

    public int getInt(String str) {
```

```
            int returnInt = 0;
            try {
                Integer tempInt = new Integer(str);
                returnInt = tempInt.intValue();
                return returnInt;
            } catch (NumberFormatException e1) {
                System.out.println("\nPlease enter an integer.");
                return returnInt;
            }
        }

        public int getBool(String str) {
            if (str.equals("TRUE") || str.equals("T"))
                return 1;
            else if (str.equals("FALSE") || str.equals("F"))
                return 0;
            else
                return -1;
        }

        public int getTrans(String str) {
            int intValue = -1;
            if (str.equalsIgnoreCase("manual")
                || str.equalsIgnoreCase("m")) {
                intValue = Vehicle.MANUAL;
            }
            else if (str.equalsIgnoreCase("automatic")
                || str.equalsIgnoreCase("a")
                || str.equalsIgnoreCase("auto")) {
                intValue = Vehicle.AUTOMATIC;
            }
            return intValue;
        }
    } // End Vrc class
```

VrcInit.java

```
////////////////////////////////////////////////////////////////
//
//  VrcInit - initializes the rental company database.
//  This class:
//  * Opens a connection to a federated database called "Vrc"
//    for read/write access.
//  * Creates a session and gets its associated federated database.
//  * Creates a database named "VehiclesDB".
//  * Creates a container named "VehiclesContainer".
//  * Adds a couple of Vehicle objects to the database; vehicles
//    become persistent by being clustered in a container.
//  * Closes the federated database.
//
// Copyright (c) 1998 Objectivity, Inc.  All Rights Reserved
//
// UNPUBLISHED PROPRIETARY SOURCE CODE OF OBJECTIVITY, INC.
//
// The copyright notice above does not evidence any actual
// or intended publication of such source code.
//
////////////////////////////////////////////////////////////////

import com.objy.db.*;     //Import Objectivity for Java exceptions
import com.objy.db.app.*; //Import Objectivity for Java classes
import java.util.*;
import java.io.IOException;
import java.io.BufferedReader;
import java.io.InputStreamReader;

public class VrcInit {
    public static void main(String args[]) {
        Connection connection;
        Session session;
        ooFDObj vrcFD;
        ooDBObj vehiclesDB;
        ooContObj vehiclesContainer;
        Fleet fleet;
        Vehicle vehicle;

        // Open a connection to a federated database
        try {
            connection = Connection.open("Vrc", oo.openReadWrite);
        } catch (DatabaseNotFoundException e1) {
           System.out.println("\nFederated database \"Vrc\" not found" +
```

```
                    " - use oonewfd to create federated database.");
        return;
    } catch (DatabaseOpenException e) {
        System.out.println("\nConnection to federated database" +
                           " \"Vrc\" already open.");
        return;
    }

    // Create session
    session = new Session();
    session.begin();

    // Turn off schema manager informational messages
    (connection.getSchemaPolicy()).setVerbose(false);

    // Retrieve federated database from session
    vrcFD = session.getFD();

    // Get or create database

    if (vrcFD.hasDB("VehiclesDB"))
              vehiclesDB = vrcFD.lookupDB("VehiclesDB");
    else {
        try {
            vehiclesDB = vrcFD.newDB("VehiclesDB");
        } catch (LockNotGrantedException e) {
            e.reportErrors();
            session.abort();
            return;
        }
        System.out.println("\nCreated database \"VehiclesDB\".");
    }
    session.commit();

    session.begin();
    try {
        fleet = (Fleet)vehiclesDB.lookup("Fleet");
    } catch (ObjectNameNotFoundException e4) {
        try {
            fleet = new Fleet();
            vehiclesDB.bind(fleet, "Fleet");
        } catch (ObjectNameNotUniqueException e5) {
            System.out.println("Could not add fleet named root.");
            session.abort();
            return;
        }
    }
```

```
session.commit();

// Get or create container
session.begin();
if (vehiclesDB.hasContainer("VehiclesContainer"))
              vehiclesContainer =
               vehiclesDB.lookupContainer("VehiclesContainer");
else {
    // Create a container
    vehiclesContainer = new com.objy.db.app.ooContObj();
   // Make the container persistent by adding it to a database.
    try {
        vehiclesDB.addContainer(vehiclesContainer,
                                "VehiclesContainer", 0, 5, 10);
    } catch (LockNotGrantedException e) {
        e.reportErrors();
        session.abort();
        return;
    }
 System.out.println("Created container \"VehiclesContainer\".");
}
session.commit();

// Add a couple of vehicles
session.begin();

// Create a new vehicle of class G, 4 doors, manual transmission
vehicle = new Vehicle("CA1234", "G", 4, Vehicle.MANUAL, 40);
// Make the vehicle persistent by clustering it in a container
vehicle.setFleet(fleet);
vehiclesContainer.cluster(vehicle);
fleet.addVehicle(vehicle);
System.out.println("\nAdded vehicle: "
                   + vehicle.toShortString());

// Create a new vehicle of class H, 4 doors, automatic transmission
vehicle = new Vehicle("CA7654", "H", 4, Vehicle.AUTOMATIC, 40);
// Make the vehicle persistent by clustering it in a container
vehicle.setFleet(fleet);
vehiclesContainer.cluster(vehicle);
fleet.addVehicle(vehicle);
System.out.println("Added vehicle: "
                   + vehicle.toShortString());

// Add an index over the license field
if (!vehiclesContainer.hasIndex("VehiclesIndex"))
    vehiclesContainer.addUniqueIndex("VehiclesIndex",
```

```
                                                "Vehicle", "license");
            session.commit();

            // Close connection
            try {
                connection.close();
            } catch (DatabaseClosedException e3) {
                System.out.println("\nConnection already closed.");
                return;
            }
        }
        static public String getString(String prompt) {
            String  line = "";

            System.out.println(prompt);
            BufferedReader in = new
                        BufferedReader(new InputStreamReader(System.in));
            try { line = in.readLine(); }
            catch(java.io.IOException e) {
                System.out.println("\nError reading input, exiting...");
                return null;
            }
            if (line == null)
                return null;
            else if (line.length() == 0)
                return null;

            return line;
        }
    } // End VrcInit class
```

24

Application Objects

This chapter contains the source code for the programming examples presented in Chapter 2, “Application Objects”.

The `MultipleThreadsSP` programming example (see page 366) simulates a server application handling requests to look up or list root objects in a database. The server creates a new thread to handle each request. Instead of having each thread create a new session, the example shares a pool of sessions among the threads. Access to the pool of sessions is synchronized. The session pool is implemented in the `SessionPool` class (see page 372). The example also illustrates the use of a restricted thread policy: before a thread uses a session that it retrieves from the pool, it executes a join operation.

This example can be executed after you compile the files and create a federated database named `"Objects"` in the `Application` subdirectory of the samples directory.

In This Chapter

MultipleThreadsSP.java

```
//////////////////////////////////////////////////////////////////////
//
// MultipleThreadsSP.java - simulates a server application handling
// requeststo lookup or list root objects in a database.
// The server creates a new thread to handle each request.
// The thread policy is RESTRICTED. Consequently, each thread:
// * Gets a session from a shared pool of sessions
// * Joins the session
// * Begins a transaction
// * Executes the request
// * Commits the transaction
// * Leaves the session
// * Returns the session to the session pool
//
// Copyright (c) 1998 Objectivity, Inc.  All Rights Reserved
//
// UNPUBLISHED PROPRIETARY SOURCE CODE OF OBJECTIVITY, INC.
//
// The copyright notice above does not evidence any actual
// or intended publication of such source code.
//
//////////////////////////////////////////////////////////////////////
import com.objy.db.*;     //Import Objectivity for Java exceptions
import com.objy.db.app.*; //Import Objectivity for Java classes
import com.objy.db.util.ooMap;

class MultipleThreadsSP {
    public static void main(String args[]) {
        MultipleThreadsServer srv = new MultipleThreadsServer();
        srv.setDaemon(true);
        srv.start();
    }
}

class MultipleThreadsServer extends Thread {
    private Connection connection;
    private Session session;
    private SessionPool sp;
    private static int REQUEST_COUNT = 10;
    public MultipleThreadsServer() {
        try {
            connection = Connection.open("Objects", oo.openReadWrite);
        } catch (DatabaseNotFoundException e1) {
            System.out.println("Federated Database" +
```

```
                        " \"Objects\" not found.");
    return;
} catch (DatabaseOpenException e2) {
    System.out.println("Federated Database" +
                        " \"Objects\" already open.");
    return;
}
// Create session pool, shared among all operations
SessionPool sp = new SessionPool(2, 100);
Session session = new Session();
session.setRecoveryAutomatic(true);
ooFDObj fd = session.getFD();
session.begin();
ooDBObj db;
if (fd.hasDB("ObjectsDB")) {
    db = fd.lookupDB("ObjectsDB");
}
else {
    db = fd.newDB("ObjectsDB");
    ooMap map = new ooMap();
    try {
        db.bind(map, "mapObject1");
    }
    catch (ObjectNameNotUniqueException e3) {
      System.out.println("Map \"mapObject1\" already exists.");
    }
    map = new ooMap();
    try {
        db.bind(map, "mapObject2");
    }
    catch (ObjectNameNotUniqueException e3) {
      System.out.println("Map \"mapObject2\" already exists.");
    }
}
session.commit();
Thread [] workers = new Thread[REQUEST_COUNT+1];
int request = 0;
int requestCount = 0;
int mapObject;
while (requestCount < REQUEST_COUNT) {
    // Generate a random request every half second
    // Terminate after REQUEST_COUNT requests
    request = (int)Math.floor(Math.random() * 10);
    try {
        // sleep half a second between requests
        sleep(500);
    }
```

```
            catch (InterruptedException e4) {
                System.out.println("Dispatcher interrupted.");
            }

            // Dispatch to appropriate method
            if (request < 5) {
                // Look up random map object
                mapObject = (int)Math.floor(Math.random() * 3);
                String lookupStr = new String("mapObject" + mapObject);
                Thread t1 = new LookupSP(sp, lookupStr);
                workers[requestCount] = t1;
                t1.start();
            }
            else {
                // List all objects
                Thread t2 = new ListSP(sp);
                workers[requestCount] = t2;
                t2.start();
            }
            requestCount++;
            // Print out the number of sessions from the pool in use
            System.out.println("Number of sessions in use: "
                               + sp.inUseSessions() +
                               " out of: " + sp.totalSessions());
        }

        // Wait for threads to finish
        for (int i=0; i < requestCount; i++) {
            synchronized (this) {
                if (workers[i].isAlive()) {
                    try {
                        workers[i].join();
                    } catch (Exception e) {
                        e.printStackTrace();
                    }
                }
                else
                    continue;
            }
        }

        // Close the connection
        try {
            connection.close();
        } catch (DatabaseClosedException e3) {
            System.out.println("Connection already closed.");
            return;
```

```
        }
    }
}
class LookupSP extends Thread {
    private ooFDObj fd;
    private SessionPool sp;
    private String str;
    public LookupSP(SessionPool isp, String istr) {
        sp = isp;
        str = istr;
    }
    public void run() {
        Session session;
        while ((session = sp.getSession()) == null);
        try {
            // Sleep 2 seconds before executing request
            sleep(2000);
        }
        catch (InterruptedException e4) {
            System.out.println("Dispatcher interrupted.");
        }
        session.join();
        session.begin();
        ooFDObj fd = session.getFD();
        ooDBObj db = null;
        boolean gotDB = false;
        if (fd.hasDB("ObjectsDB")) {
            // Catch exception if federated database is locked
            while(!gotDB) {
                try {
                    db = fd.lookupDB("ObjectsDB");
                    gotDB = true;
                } catch (ObjyRuntimeException e) {
                }
            }
        }
        else {
            System.out.println("Database \"ObjectsDB\" not found.");
            session.abort();
            session.leave();
            sp.returnSession(session);
            return;
        }
        Object object;
        System.out.println("\nLooking up " + str);
        try {
            object = db.lookup(str);
```

```
            } catch (ObjectNameNotFoundException e4) {
                System.out.println("Object not found.");
                session.abort();
                session.leave();
                sp.returnSession(session);
                return;
            }
            if (object != null)
                System.out.println("Found object: " + str);
            else
                System.out.println("Object is null.");
            session.commit();
            session.leave();
            sp.returnSession(session);
        }
    }

    class ListSP extends Thread {
        private SessionPool sp;
        public ListSP(SessionPool isp) {
            sp = isp;
        }
        public void run() {
            Session session;
            while ((session = sp.getSession()) == null)
            try {
                // Sleep 2 seconds before executing request
                sleep(2000);
            }
            catch (InterruptedException e4) {
                System.out.println("Dispatcher interrupted.");
            }
            ooDBObj db = null;
            session.join();
            session.begin();
            ooFDObj fd = session.getFD();
            boolean gotDB = false;
            if (fd.hasDB("ObjectsDB")) {
                // Catch exception if federated database is locked
                while(!gotDB) {
                    try {
                        db = fd.lookupDB("ObjectsDB");
                        gotDB = true;
                    } catch (ObjyRuntimeException e) {
                    }
                }
            }
```

```
            else {
                System.out.println("Database \"ObjectsDB\" not found.");
                session.abort();
                session.leave();
                sp.returnSession(session);
                return;
            }
            System.out.println("\nListing all objects:");
            Iterator itr = db.rootNames();
            while(itr.hasNext()) {
                System.out.println("\t" + (String)itr.next());
            }
            session.commit();
            session.leave();
            sp.returnSession(session);
        }
    } // End MultipleThreadsSP class
```

SessionPool.java

```
//////////////////////////////////////////////////////////////////////
//
// SessionPool - implements a shared pool of sessions.
//
// Copyright (c) 1998 Objectivity, Inc.  All Rights Reserved
//
// UNPUBLISHED PROPRIETARY SOURCE CODE OF OBJECTIVITY, INC.
//
// The copyright notice above does not evidence any actual
// or intended publication of such source code.
//
//////////////////////////////////////////////////////////////////////

import com.objy.db.*;     //Import Objectivity for Java exceptions
import com.objy.db.app.*; //Import Objectivity for Java classes
import java.util.*;
public class SessionPool {
    private int inUseSessions;
    private int totalSessions;
    private int maxSessions;
    private Stack pool;

    public SessionPool(int startTotal, int maxSessions) {
        this.pool = new Stack();
        for (int i = 0; i < startTotal; i++)
            this.pool.push(new Session());
        this.inUseSessions = 0;
        this.totalSessions = startTotal;
        this.maxSessions = maxSessions;
    }
    public synchronized Session getSession() {
        if (this.pool.empty()) {
            if (totalSessions() == maxSessions())
                return null;
            this.pool.push(new Session());
            this.totalSessions++;
        }
        this.inUseSessions++;
        return (Session)this.pool.pop();
    }
    public synchronized void returnSession(Session session) {
        this.pool.push(session);
        this.inUseSessions--;
    }
```

```
    public synchronized int inUseSessions() {
        return this.inUseSessions;
    }
    public synchronized int totalSessions() {
        return this.totalSessions;
    }
    public synchronized int maxSessions() {
        return this.maxSessions;
    }
} // End SessionPool class
```

25

ODMG Application Objects

This chapter contains the source code for the programming examples presented in Chapter 3, “ODMG Application Objects”.

The `MultipleThreadsTP` programming example simulates a server application handling requests to look up root objects in a database. The server creates a new thread to handle each request and each thread gets a transaction from a shared pool of transactions. The transaction pool is implemented in the `TransactionPool` class.

This example can be executed after you compile the files and create a federated database named `"Objects"` in the `ODMGApplication` subdirectory of the samples directory.

In This Chapter

MultipleThreadsTP.java

```
//////////////////////////////////////////////////////////////////////
//
// MultipleThreadsTP.java - simulates a server application handling
// requests to lookup root objects in a database.
// The server creates a new thread to handle each request.
// The thread policy is RESTRICTED. Consequently, each thread:
// * Gets a transaction from a shared pool of transactions
// * Joins the transaction
// * Begins a transaction
// * Executes the request
// * Commits the transaction
// * Leaves the transaction
// * Returns the transaction to the transaction pool
//
// Copyright (c) 1998 Objectivity, Inc.  All Rights Reserved
//
// UNPUBLISHED PROPRIETARY SOURCE CODE OF OBJECTIVITY, INC.
//
// The copyright notice above does not evidence any actual
// or intended publication of such source code.
//
//////////////////////////////////////////////////////////////////////

import com.objy.db.*;     //Import Objectivity for Java exceptions
import com.objy.db.app.*; //Import Objectivity for Java classes
import com.objy.db.util.ooMap;

class MultipleThreadsTP {
    public static void main(String args[]) {
        MultipleThreadsServer srv = new MultipleThreadsServer();
        srv.setDaemon(true);
        srv.start();
    }
}

class MultipleThreadsServer extends Thread {
    private Database database;
    private TransactionPool tp;
    private static int REQUEST_COUNT = 20;
    public MultipleThreadsServer() {
        // Open ODMG database
        try {
            database = Database.open("Objects", oo.openReadWrite);
        } catch (DatabaseNotFoundException e1) {
```

```
    System.out.println("Federated Database" +
                       " \"Objects\" not found.");
    return;
} catch (DatabaseOpenException e2) {
    System.out.println("Federated Database" +
                       " \"Objects\" already open.");
    return;
}
// Create transaction pool, shared among all operations
TransactionPool tp = new TransactionPool(2, 100);
Transaction tx = new Transaction();
tx.begin();

ooMap map = new ooMap();
try {
    database.bind(map, "mapObject1");
}
catch (ObjectNameNotUniqueException e3) {
    System.out.println("Map \"mapObject1\" already exists.");
}
map = new ooMap();
try {
    database.bind(map, "mapObject2");
}
catch (ObjectNameNotUniqueException e3) {
    System.out.println("Map \"mapObject2\" already exists.");
}
tx.commit();
Thread [] workers = new Thread[REQUEST_COUNT+1];
int request = 0;
int requestCount = 0;
int mapObject;
while (requestCount < REQUEST_COUNT) {
    // Generate a random request every half second
    // Terminate after REQUEST_COUNT requests
    request = (int)Math.floor(Math.random() * 10);
    try {
        sleep(500);
    }
    catch (InterruptedException e4) {
        System.out.println("Dispatcher interrupted.");
    }

    // Look up random map object
    mapObject = (int)Math.floor(Math.random() * 3);
    String lookupStr = new String("mapObject" + mapObject);
    Thread t1 = new LookupTP(database, tp, lookupStr);
```

```
                workers[requestCount] = t1;
                t1.start();
                requestCount++;
                System.out.println("Number of transactions in use: "
                                   + tp.inUseTransactions() +
                                   " out of: " + tp.totalTransactions());
            }
            // Wait for threads to finish
            for (int i=0; i < requestCount; i++) {
                synchronized (this) {
                    if (workers[i].isAlive()) {
                        try {
                            workers[i].join();
                        } catch (Exception e) {
                            e.printStackTrace();
                        }
                    }
                    else
                        continue;
                }
            }
            // Close ODMG database
            try {
                database.close();
            } catch (DatabaseClosedException e3) {
                System.out.println("Database already closed.");
                return;
            }
        }
    }
    class LookupTP extends Thread {
        private Database database;
        private TransactionPool tp;
        private String str;
       public LookupTP(Database idatabase, TransactionPool itp, String istr) {
            database = idatabase;
            tp = itp;
            str = istr;
        }
        public void run() {
            Transaction tx;
            while ((tx = tp.getTransaction()) == null);
            try {
                // Sleep 2 seconds before executing request
                sleep(2000);
            }
            catch (InterruptedException e4) {
```

```
                System.out.println("Dispatcher interrupted.");
            }
            tx.join();
            tx.begin();
            Object object = null;
            boolean lookedUp = false;
            System.out.println("\nLooking up " + str);
            try {
                // Catch exception if federated database is locked
                while(!lookedUp) {
                    try {
                        object = database.lookup(str);
                        lookedUp = true;
                    } catch (ObjyRuntimeException e) {
                    }
                }
            } catch (ObjectNameNotFoundException e4) {
                System.out.println("Object not found.");
                tx.abort();
                tx.leave();
                tp.returnTransaction(tx);
                return;
            }
            if (object != null)
                System.out.println("Found object: " + str);
            else
                System.out.println("Object is null.");
            tx.commit();
            tx.leave();
            tp.returnTransaction(tx);
        }
    } // End MultipleThreadsTP class
```

TransactionPool.java

```
////////////////////////////////////////////////////////////////////////
//
// TransactionPool - implements a shared pool of transactions.
//
// Copyright (c) 1998 Objectivity, Inc.  All Rights Reserved
//
// UNPUBLISHED PROPRIETARY SOURCE CODE OF OBJECTIVITY, INC.
//
// The copyright notice above does not evidence any actual
// or intended publication of such source code.
//
////////////////////////////////////////////////////////////////////////
import com.objy.db.*;     //Import Objectivity for Java exceptions
import com.objy.db.app.*; //Import Objectivity for Java classes
import java.util.*;

public class TransactionPool {
    private int inUseTransactions;
    private int totalTransactions;
    private int maxTransactions;
    private Stack pool;

    public TransactionPool(int startTotal, int maxTransactions) {
        this.pool = new Stack();
        for (int i = 0; i < startTotal; i++)
            this.pool.push(new Transaction());
        this.inUseTransactions = 0;
        this.totalTransactions = startTotal;
        this.maxTransactions = maxTransactions;
    }
    public synchronized Transaction getTransaction() {
        if (this.pool.empty()) {
            if (totalTransactions() == maxTransactions())
                return null;
            this.pool.push(new Transaction());
            this.totalTransactions++;
        }
        this.inUseTransactions++;
        return (Transaction)this.pool.pop();
    }
    public synchronized void returnTransaction(Transaction tx) {
        this.pool.push(tx);
        this.inUseTransactions--;
    }
```

```
    public synchronized int inUseTransactions() {
        return this.inUseTransactions;
    }
    public synchronized int totalTransactions() {
        return this.totalTransactions;
    }
    public synchronized int maxTransactions() {
        return this.maxTransactions;
    }
} // End TransactionPool class
```

26

Storage Objects

The example discussed in Chapter 5, “Storage Objects,” illustrates how to use multiple containers to improve scalability and concurrency.

In This Chapter

Example

The example revisits the vehicle rental company example introduced in Chapter 1, “Getting Started”. The implementation is changed as follows:

- The `Fleet` class (see page 385) uses a map instead of a fixed-size array to maintain its link with all its contained vehicles (see page 343). Chapter 6, “Defining Persistence-Capable Classes,” discusses trade-offs between various implementations of links to multiple objects. The fleet remains a named root in the vehicles database.
- The vehicles themselves are randomly distributed among a fixed-size container pool (see page 387) of garbage-collectible containers, instead of being stored in a single container. This allows one container to be updated without preventing other containers from being accessed. The container pool is also a named root in the vehicles database.

  The vehicles are stored according to the container pool clustering strategy (see page 389). This strategy is installed in the sessions used by the database initialization class `VrcInit` (see page 361) and the interactive application class `Vrc` (see page 348).

A vehicle is deleted simply by removing its entry in the fleet map. Because the vehicles are stored in garbage collectible containers, when a vehicle is removed from its fleet, it will be garbage collected when `oogc` is executed.

All the containers are referenced from a named root (the container pool). As a consequence, they will not be garbage collected even if all their objects are deleted or garbage collected.

The files for this example are in the `Storage` subdirectory of the programming samples directory.

To execute this example, you need to:

1. Compile the files `Fleet.java`, `VrcInit.java`, `Vrc.java`, and `ContainerPool*.java` in the `Storage` subdirectory of the programming samples directory.
2. Start an Objectivity / DB lock server.
3. Create a federated database called `Vrc` in the `Storage` directory.
4. Execute `VrcInit` to initialize the federated database.

Fleet.java

```
////////////////////////////////////////////////////////////////////////
//
// Fleet - a fleet of rental vehicles
//
// A map in a persistent field links a fleet to its vehicles.
// Field access methods hide the map implementation.
//
// Copyright (c) 1998 Objectivity, Inc.  All Rights Reserved
//
// UNPUBLISHED PROPRIETARY SOURCE CODE OF OBJECTIVITY, INC.
//
// The copyright notice above does not evidence any actual
// or intended publication of such source code.
//
////////////////////////////////////////////////////////////////////////

import com.objy.db.*;     //Import Objectivity for Java exceptions
import com.objy.db.app.*; //Import Objectivity for Java classes
import com.objy.db.util.ooMap;

public class Fleet extends ooObj {

    // Persistent fields
    protected ooMap vehicles = new ooMap();
    protected int numberOfVehicles;

    // Constructor
    public Fleet() {
        this.numberOfVehicles = 0;
    }

    // All access methods must be called during a transaction

    public ooMap vehicles() {
        fetch();
        return this.vehicles;
    }

    // Field access methods
    public long getNumVehicles() {
        fetch();
        return this.vehicles.getElementCount();
    }
```

```
        public void addVehicle(Vehicle newMember) {
            fetch();
            String key = newMember.getLicense();
            if (this.vehicles.isMember(key))
               return;
            this.vehicles.add(newMember, key);
        }

        public void deleteVehicle(Vehicle vehicle) {
            fetch();
            String key = vehicle.getLicense();
            if (this.vehicles.isMember(key)) {
                this.vehicles.remove(key);
            }
        }

        public Vehicle findVehicle(String license) {
            fetch();
            if (this.vehicles.isMember(license)) {
                // Cast retrieved object to class Vehicle
                return (Vehicle)this.vehicles.lookup(license);
            }
            else {
                return null;
            }
        }

        public java.util.Iterator getAllVehicles() {
            fetch();
            return this.vehicles.elements();
        }
    } // End Fleet class
```

ContainerPool.java

```
////////////////////////////////////////////////////////////////////////
//
// ContainerPool - a pool of containers used for clustering objects
//
// Copyright (c) 1998 Objectivity, Inc.  All Rights Reserved
//
// UNPUBLISHED PROPRIETARY SOURCE CODE OF OBJECTIVITY, INC.
//
// The copyright notice above does not evidence any actual
// or intended publication of such source code.
//
////////////////////////////////////////////////////////////////////////

import com.objy.db.*;     //Import Objectivity for Java exceptions
import com.objy.db.app.*; //Import Objectivity for Java classes
import java.util.*;

public class ContainerPool extends ooObj {
    private ooGCContObj[] pool;
    private int containerCount;

    // Constructor creates a pool of the specified number of
    // containers in the specified database
    public ContainerPool(ooDBObj db, int numberOfContainers) {
        this.pool = new ooGCContObj[numberOfContainers];
        this.containerCount = numberOfContainers;
        for (int i = 0; i < numberOfContainers; i++) {
            pool[i] = new ooGCContObj();
            // Add unnamed containers to database
            db.addContainer(pool[i], "", 0, 5, 10);
        }
    }
    public ooGCContObj getContainer(int i) {
        fetch();
        if (i >= 0 && i <= getContainerCount())
            return pool[i];
        else
            return null;
    }
    public int getContainerCount() {
        fetch();
        return containerCount;
    }
    public void clusterObject(ooObj object) {
```

```
            int index = Math.abs((new Random()).nextInt()) %
                        this.getContainerCount();

            // Cluster the object with the selected container
            ooGCContObj container = this.getContainer(index);
            container.cluster(object);
        }
    } // End ContainerPool class
```

ContainerPoolStrategy.java

```
//////////////////////////////////////////////////////////////////////
//
// ContainerPoolStrategy - an example clustering strategy
//
// Copyright (c) 1998 Objectivity, Inc.  All Rights Reserved
//
// UNPUBLISHED PROPRIETARY SOURCE CODE OF OBJECTIVITY, INC.
//
// The copyright notice above does not evidence any actual
// or intended publication of such source code.
//
//////////////////////////////////////////////////////////////////////

import com.objy.db.*;     //Import Objectivity for Java exceptions
import com.objy.db.app.*; //Import Objectivity for Java classes

public class ContainerPoolStrategy implements ClusterStrategy {

           public void requestCluster(Object requestObject,
                                ClusterReason reason, Object object) {
        ooDBObj db;
        if (requestObject instanceof ooContObj)
            db = ((ooContObj)requestObject).getDB() ;
        else if (requestObject instanceof ooObj)
            db = (((ooObj)requestObject).getContainer()).getDB() ;
        else if (requestObject instanceof ooDBObj)
            db = (ooDBObj)requestObject ;
        else
            throw new ContainerPoolStrategyException
                    ("ContainerPoolStrategy malfunction.");

        try {
            ContainerPool containerPool =
                    (ContainerPool)(db.lookup("ContainerPool"));
            containerPool.clusterObject((ooObj)object);
        }
        catch (ObjectNameNotFoundException e) {
            throw new ContainerPoolStrategyException
                ("ContainerPoolStrategy malfunction.");
        }
    }
}
```

27

Defining Persistence-Capable Classes

The example in Chapter 1, “Getting Started,” described how to define a persistence capable class using inheritance and developed a simple application that accessed objects of that class. The examples discussed in Chapter 6, “Defining Persistence-Capable Classes,” illustrate:

- How to define field access methods and relationship access methods for persistence-capable classes.
- Alternative ways to implement links between associated objects:
  - Classes in the `RentalFields` package use references in persistent fields to implement links between objects; an array of references in a persistent field implements a link from one object to many associated objects.
  - In the `RentalMap` package, a map in a persistent field implements a link from one object to many associated objects.
  - Classes in the `RentalRelations` package use relationships to implement links between objects.
- How to define a persistence-capable class by implementing the `Persistent` interface.

In This Chapter

PersistentInterface Package
Vehicle.java
Delegator.java

RentalFields Package

The `RentalFields` package contains three classes:

- The `Vehicle` class represents vehicles in a rental fleet. A reference in a vehicle's `fleet` field links the vehicle to its fleet. In addition to field access methods, this class has a method `printInfo` that illustrates how to obtain properties of a persistent object that are managed by Objectivity for Java; you should not define persistent fields corresponding to any of these properties.
- The `SimpleFleet` class (see page 396) represents rental fleets. An array of references in a fleet's `vehicles` field links the fleet to the vehicles it contains. The field access methods for the `vehicles` field get and set the vehicle at a given array index.
- The `Fleet` class (see page 397) is an alternative implementation of the `SimpleFleet` class. The field access methods for the `vehicles` field hide the array implementation.

Vehicle.java

```
//////////////////////////////////////////////////////////////////////
//
// RentalFields.Vehicle - vehicle in a rental fleet
// A reference in a persistent field links a vehicle to its fleet.
//
// Copyright (c) 1998 Objectivity, Inc.  All Rights Reserved
//
// UNPUBLISHED PROPRIETARY SOURCE CODE OF OBJECTIVITY, INC.
//
// The copyright notice above does not evidence any actual
// or intended publication of such source code.
//
//////////////////////////////////////////////////////////////////////

package RentalFields;
import com.objy.db.*;     // Import Objectivity for Java exceptions
import com.objy.db.app.*; // Import Objectivity for Java classes

public class Vehicle extends ooObj {
    // Persistent fields
```

```
protected String license;
protected String type;
protected int doors;
protected int transmission;
protected boolean available;
protected Fleet fleet;

// Legal values for transmission field
public static final int MANUAL = 0;
public static final int AUTOMATIC = 1;

// Transient field
protected transient int dailyRate;

// Constructor
public Vehicle(String license, String type,
               int doors, int transmission) {
    this.license = license;
    this.type = type;
    this.doors = doors;
    this.transmission = transmission;
    this.available = true;
}

// All access methods must be called during a transaction

// Field access methods to get persistent field values
public String getLicense() {
    fetch();
    return this.license;
}

public String getType() {
    fetch();
    return this.type;
}

public int getDoors() {
    fetch();
    return this.doors;
}

public int getTransmission() {
    fetch();
    return this.transmission;
}
```

```
public boolean isAvailable() {
    fetch();
    return this.available;
}

public Fleet getFleet() {
    fetch();
    return this.fleet;
}

// Field access method to set fleet field
public void setFleet(Fleet fleet) {
    markModified();
    this.fleet = fleet;
}

// Field access methods to set available field
public void rentVehicle() {
    markModified();
    this.available = false;
}

public void returnVehicle() {
    markModified();
    this.available = true;
}

// Utility to print description of this vehicle
public String toString() {
    // This method must be called during a transaction
    fetch();
    StringBuffer buffer = new StringBuffer();
    if (available)
        buffer.append("AVAILABLE ");
    else
        buffer.append("RENTED ");
    buffer.append("License:" + license);
    buffer.append(" Type:" + type);
    buffer.append(" Doors:" + doors);
    if (transmission == MANUAL)
        buffer.append(" MANUAL ");
    else
    buffer.append(" AUTOMATIC ");
    String strng = new String(buffer);
    return strng;
}
```

```
// Static utility to obtain properties maintained by
// Objectivity for Java
public static void printInfo(Vehicle vehicle) {
    // This method must be called during a transaction

    // Get vehicle's container
    ooContObj cont = vehicle.getContainer();
    // Get vehicle's database
    ooDBObj db = cont.getDB();
    // Get vehicle's federated database
    ooFDObj fd = db.getFD();
    // Get vehicle's session
    Session session = vehicle.getSession();
    // Get vehicle's object identifier
    ooId oid = vehicle.getOid();

    // Print the location in the storage hierarchy
    System.out.println("Vehicle " + vehicle.getLicense() +
                       " has OID " + oid.getStoreString() +
                       " and is stored in:");
    if (cont.getName().equals("")) {
        System.out.println("    an unnamed container in");
    }
    else {
       System.out.println("    container " + cont.getName() + " in");
    }
    System.out.println("        database " + db.getName() + " in");
  System.out.println("         federated database " + fd.getName());

    // Print information about the session
    String access, txType;
    if (session.getOpenMode() == oo.openReadOnly) {
        access = "read only";
    }
    else {
        access = "read/write";
    }
    if (session.getMrowMode() == oo.MROW) {
        txType = "MROW";
    }
    else {
        txType = "nonMROW";
    }
    System.out.println("The session is open for " +
                       access + " access by " +
                       txType + "transactions");
}
```

```
} // End Vehicle class
```

SimpleFleet.java

```
//////////////////////////////////////////////////////////////////////
//
// RentalFields.SimpleFleet - a fleet of rental vehicles
//
// An array of reference in a persistent field links a fleet to its
// vehicles.
//
// Field access methods allow the caller to manipulate the array of
// references.
//
// Copyright (c) 1998 Objectivity, Inc.  All Rights Reserved
//
// UNPUBLISHED PROPRIETARY SOURCE CODE OF OBJECTIVITY, INC.
//
// The copyright notice above does not evidence any actual
// or intended publication of such source code.
//
//////////////////////////////////////////////////////////////////////

package RentalFields;
import com.objy.db.*;     // Import Objectivity for Java exceptions
import com.objy.db.app.*; // Import Objectivity for Java classes

public class SimpleFleet extends ooObj {
    static final int FLEET_SIZE = 1000;

    // Persistent fields
    protected Vehicle[] vehicles  = new Vehicle[FLEET_SIZE];

    // Constructor
    public SimpleFleet() {
    }

    // All access methods must be called during a transaction

    // Field access methods
    public Vehicle getVehicle(int n) {
        fetch();
        return this.vehicles[n];
    }
```

```
    public void setVehicle(int n, Vehicle newMember) {
        markModified();
        this.vehicles[n] = newMember;
    }
} // end SimpleFleet class
```

Fleet.java

```
//////////////////////////////////////////////////////////////////////
//
// RentalFields.Fleet - a fleet of rental vehicles
//
// An array of reference in a persistent field links a fleet to its
// vehicles.
//
// Field access methods hide the array implementation.
//
// Copyright (c) 1998 Objectivity, Inc.  All Rights Reserved
//
// UNPUBLISHED PROPRIETARY SOURCE CODE OF OBJECTIVITY, INC.
//
// The copyright notice above does not evidence any actual
// or intended publication of such source code.
//
//////////////////////////////////////////////////////////////////////

package RentalFields;
import com.objy.db.*;     // Import Objectivity for Java exceptions
import com.objy.db.app.*; // Import Objectivity for Java classes
import java.util.Iterator;
import java.util.NoSuchElementException;

public class Fleet extends ooObj {
    static final int FLEET_SIZE = 1000;

    // Persistent fields
    protected Vehicle[] vehicles  = new Vehicle[FLEET_SIZE];
    protected int numberOfVehicles;

    // Constructor
    public Fleet() {
        this.numberOfVehicles = 0;
    }

    // All access methods must be called during a transaction
```

```
// Field access methods
public int getNumVehicles() {
    fetch();
    return this.numberOfVehicles;
}

public void addVehicle(Vehicle newMember) {
    markModified();
    if (findVehicle(newMember.getLicense()) == null) {
        this.vehicles[this.numberOfVehicles] = newMember;
        this.numberOfVehicles++;
    }
}

public void deleteVehicle(Vehicle vehicle) {
    markModified();
    int i = 0;
    while ((i < this.numberOfVehicles)
            && (this.vehicles[i] != vehicle)) {
        i++;
    }
    if (i != this.numberOfVehicles) {
        // Vehicle was found; remove it
        for (int j = i + 1; j < this.numberOfVehicles; j++, i++) {
            this.vehicles[i] = this.vehicles[j];
        }
        this.numberOfVehicles--;
    }
}

public Vehicle findVehicle(String license) {
    fetch();
    if (this.numberOfVehicles == 0) {
        return null;
    }
    for (int i = 0; i < this.numberOfVehicles; i++) {
        if (this.vehicles[i].getLicense().equals(license)) {
            return this.vehicles[i];
        }
    }
    return null;
}

public java.util.Iterator getAllVehicles() {
    fetch();
    return new VehicleItr(this);
}
```

```
    // Internal access methods (used by VehicleItr)
    protected Vehicle getVehicle(int n) {
        fetch();
        return this.vehicles[n];
    }

    // Inner class to support getAllVehicles
    class VehicleItr implements java.util.Iterator
    {
        // Private fields
        private transient int   currentIndex;
        private transient Fleet fleet;

        // Constructor
        VehicleItr(Fleet forFleet) {
            this.fleet = forFleet;
            this.currentIndex = 0;
        }

        // Public Iterator methods
        public boolean hasNext() {
            int max = this.fleet.getNumVehicles();
            if (this.currentIndex < max)
                return true;
            else
                return false;
        }

        public Object next() {
            if (!hasNext())
              throw new NoSuchElementException("No more vehicles in the
fleet.");
         Vehicle nextVehicle = this.fleet.getVehicle(this.currentIndex);
            this.currentIndex++;
            return (Object)nextVehicle;
        }

           public void remove() {
           throw new UnsupportedOperationException();
           }
    } // end VehicleItr class
} // end Fleet class
```

RentalMap Package

The `RentalMap` package contains two classes:

- The `Vehicle` class is identical to the `RentalFields.Vehicle` class; a reference in a vehicle's `fleet` field links the vehicle to its fleet.
- The `Fleet` class (see page 402) is an alternative implementation of the `RentalFields.Fleet` class. A map referenced in a fleet's `vehicles` field links the fleet to the vehicles it contains. This class provides the same set of field access methods for the `vehicles` field as the `RentalFields.Fleet` class.

Vehicle.java

```
//////////////////////////////////////////////////////////////////////
//
// RentalMap.Vehicle - vehicle in a rental fleet
// A reference in a persistent field links a vehicle to its fleet.
//
// Copyright (c) 1998 Objectivity, Inc.  All Rights Reserved
//
// UNPUBLISHED PROPRIETARY SOURCE CODE OF OBJECTIVITY, INC.
//
// The copyright notice above does not evidence any actual
// or intended publication of such source code.
//
//////////////////////////////////////////////////////////////////////

package RentalMap;
import com.objy.db.*;     // Import Objectivity for Java exceptions
import com.objy.db.app.*; // Import Objectivity for Java classes

public class Vehicle extends ooObj {
    // Persistent fields
    protected String license;
    protected String type;
    protected int doors;
    protected int transmission;
    protected boolean available;
    protected Fleet fleet;

    // Legal values for transmission field
    public static final int MANUAL = 0;
    public static final int AUTOMATIC = 1;

    // Transient field
```

```
    protected transient int dailyRate;

    // Constructor
    public Vehicle(String license, String type,
                   int doors, int transmission) {
        this.license = license;
        this.type = type;
        this.doors = doors;
        this.transmission = transmission;
        this.available = true;
    }

    // All access methods must be called during a transaction

    // Field access methods to get persistent field values
    public String getLicense() {
        fetch();
        return this.license;
    }

    public String getType() {
        fetch();
        return this.type;
    }

    public int getDoors() {
        fetch();
        return this.doors;
    }

    public int getTransmission() {
        fetch();
        return this.transmission;
    }

    public boolean isAvailable() {
        fetch();
        return this.available;
    }

    public Fleet getFleet() {
        fetch();
        return this.fleet;
    }

    // Field access methods to set fleet field
    public void setFleet(Fleet fleet) {
```

```
            markModified();
            this.fleet = fleet;
        }

        // Field access methods to set available field
        public void rentVehicle() {
            markModified();
            this.available = false;
        }

        public void returnVehicle() {
            markModified();
            this.available = true;
        }
        // Utility to print description of this vehicle
        public String toString() {
            // This method must be called during a transaction
            fetch();
            StringBuffer buffer = new StringBuffer();
            if (available)
                buffer.append("AVAILABLE ");
            else
                buffer.append("RENTED ");
            buffer.append("License:" + license);
            buffer.append(" Type:" + type);
            buffer.append(" Doors:" + doors);
            if (transmission == MANUAL)
                buffer.append(" MANUAL ");
            else
            buffer.append(" AUTOMATIC ");
            String strng = new String(buffer);
            return strng;
        }
    } // End Vehicle class
```

Fleet.java

```
            //
            // RentalMap.Fleet - a fleet of rental vehicles
            // A map in a persistent field links a fleet to its vehicles.
            //
            // Field access methods hide the map implementation.
```

```
//
// Copyright (c) 1998 Objectivity, Inc.  All Rights Reserved
//
// UNPUBLISHED PROPRIETARY SOURCE CODE OF OBJECTIVITY, INC.
//
// The copyright notice above does not evidence any actual
// or intended publication of such source code.
//
////////////////////////////////////////////////////////////
////////////

package RentalMap;
import com.objy.db.*;     // Import Objectivity for Java exceptions
import com.objy.db.app.*; // Import Objectivity for Java classes
import com.objy.db.util.ooMap;

public class Fleet extends ooObj {

   // Persistent fields
   protected ooMap vehicles = new ooMap();
   protected int numberOfVehicles;

   // Constructor
   public Fleet() {
      this.numberOfVehicles = 0;
   }

   // All access methods must be called during a transaction

   // Field access methods
```

```
public int getNumVehicles() {
   fetch();
   return this.numberOfVehicles;
}

public void addVehicle(Vehicle newMember) {
   fetch();
   String key = newMember.getLicense();
   if (this.vehicles.isMember(key))
     return;
   this.vehicles.add(newMember, key);
   this.numberOfVehicles++;
}

public void deleteVehicle(Vehicle vehicle) {
   fetch();
   String key = vehicle.getLicense();
   if (this.vehicles.isMember(key)) {
      this.vehicles.remove(key);
      this.numberOfVehicles--;
   }
}

public Vehicle findVehicle(String license) {
   fetch();
   if (this.vehicles.isMember(license)) {
      // Cast retrieved object to class Vehicle
      return (Vehicle)this.vehicles.lookup(license);
   }
   else {
```

```
            return null;
        }
    }

    public java.util.Iterator getAllVehicles() {
        fetch();
        return this.vehicles.elements();
    }
} // End Fleet class
```

RentalRelations Package

The RentalRelations package contains two classes:

- The Vehicle class is an alternative implementation of the RentalFields.Vehicle class. A many-to-one relationship in a vehicle's fleet field links the vehicle to its fleet.
- The Fleet class (see page 408) is an alternative implementation of the RentalFields.Fleet and RentalMap.Fleet classes. A one-to-many relationship in a fleet's vehicles field links the fleet to the vehicles it contains. This class contains the same field access methods for the vehicles field as the RentalFields.Fleet and RentalMap.Vehicle classes.

Vehicle.java

```
//////////////////////////////////////////////////////////////////////
//
// RentalRelations.Vehicle - vehicle in a rental fleet
// A to-one relationship links a vehicle to its fleet.
//
// Copyright (c) 1998 Objectivity, Inc.  All Rights Reserved
//
// UNPUBLISHED PROPRIETARY SOURCE CODE OF OBJECTIVITY, INC.
//
// The copyright notice above does not evidence any actual
// or intended publication of such source code.
//
//////////////////////////////////////////////////////////////////////

package RentalRelations;
import com.objy.db.*;     // Import Objectivity for Java exceptions
import com.objy.db.app.*; // Import Objectivity for Java classes

public class Vehicle extends ooObj {
    // Persistent fields
    protected String license = "";
    protected String type = "";
    protected int doors;
    protected int transmission;
    protected boolean available;

    // Relationships
    private ToOneRelationship fleet;
    protected static ManyToOne fleet_Relationship() {
        return new ManyToOne(
```

```
                    "fleet",                    // This relationship
                    "RentalRelations.Fleet",    // Related class
                    "vehicles",                 // Inverse relationship
                    Relationship.INLINE_NONE); // Not inline
    }

    // Legal values for transmission field
    public static final int MANUAL = 0;
    public static final int AUTOMATIC = 1;

    // Transient field
    protected transient int dailyRate;

    // Constructor
    public Vehicle(String license, String type,
                   int doors, int transmission) {
        this.license = license;
        this.type = type;
        this.doors = doors;
        this.transmission = transmission;
        this.available = true;
    }

    // All access methods must be called during a transaction

    // Field access methods to get persistent field values
    public String getLicense() {
        fetch();
        return this.license;
    }

    public String getType() {
        fetch();
        return this.type;
    }

    public int getDoors() {
        fetch();
        return this.doors;
    }

    public int getTransmission() {
        fetch();
        return this.transmission;
    }

    public boolean isAvailable() {
```

```
            fetch();
            return this.available;
        }

        // Field access method to set available field
        public void rentVehicle() {
            markModified();
            this.available = false;
        }

        public void returnVehicle() {
            markModified();
            this.available = true;
        }

           // Relationship access methods
        public void setFleet(Fleet fleet) {
            fetch();
            // Remove any existing relationship
            this.fleet.clear();
            this.fleet.form(fleet);
        }

        public Fleet getFleet() {
            fetch();
            // cast to Fleet
            return (Fleet)this.fleet.get();
        }
    } // End Vehicle class
```

Fleet.java

```
////////////////////////////////////////////////////////////////////////
//
// RentalRelations.Fleet - a fleet of rental vehicles
// A to-many relationship links a fleet to its vehicles.
//
// Relationship access methods hide the implementation.
//
// Copyright (c) 1998 Objectivity, Inc.  All Rights Reserved
//
// UNPUBLISHED PROPRIETARY SOURCE CODE OF OBJECTIVITY, INC.
//
// The copyright notice above does not evidence any actual
// or intended publication of such source code.
//
```

```
//////////////////////////////////////////////////////////////////////

package RentalRelations;
import com.objy.db.*;     // Import Objectivity for Java exceptions
import com.objy.db.app.*; // Import Objectivity for Java classes

public class Fleet extends ooObj {
    // Persistent fields
    protected int numberOfVehicles;

    // Relationships
    private ToManyRelationship vehicles;
    protected static OneToMany vehicles_Relationship() {
        return new OneToMany(
                    "vehicles",                 // this relationship
                    "RentalRelations.Vehicle",  // related class
                   "fleet",                    // inverse relationship
                    Relationship.INLINE_NONE);  // Not inline
    }

    // Constructor
    public Fleet() {
        this.numberOfVehicles = 0;
    }

    // All access methods must be called during a transaction

    // Field access methods
    public int getNumVehicles() {
        fetch();
        return this.numberOfVehicles;
    }

    // Relationship access methods

    public void addVehicle(Vehicle newMember) {
        fetch();
        if (this.vehicles.includes(newMember))
            return;
        this.vehicles.add(newMember);
        this.numberOfVehicles++;
    }

    public void deleteVehicle(Vehicle vehicle) {
        fetch();
        if (this.vehicles.includes(vehicle)) {
            this.vehicles.remove(vehicle);
```

```
            this.numberOfVehicles--;
        }
    }

    public Vehicle findVehicle(String license) {
        fetch();
        String predicate = new String("license == \"" +
                                      license + "\"");
        Iterator itr = this.vehicles.scan(predicate);
        if (itr.hasNext()) {
            // Cast to Vehicle
            return (Vehicle)itr.next();
        }
        else {
            return null;
        }
    }

    public java.util.Iterator getAllVehicles() {
        fetch();
        return this.vehicles.scan();
    }
} // End Fleet class
```

PersistentInterface Package

The `PersistentInterface` package contains two classes:

- The `Vehicle` class introduced in Chapter 1, “Getting Started,” is modified to implement the `Persistent` interface, instead of inheriting from `ooObj`. Because this class implements the `Persistent` interface, vehicle objects can participate in Objectivity for Java operations that require implicit persistence behavior (such as being made a named root).
- The `Delegator` class (see page 416) is a utility class whose static methods perform persistent operations by invoking the appropriate methods of a vehicle’s persistor.

Vehicle.java

```
////////////////////////////////////////////////////////////////////////
//
// Vehicle - vehicle in a rental fleet
// A reference in a persistent field links a vehicle to its fleet.
```

```
//
// Copyright (c) 1998 Objectivity, Inc.  All Rights Reserved
//
// UNPUBLISHED PROPRIETARY SOURCE CODE OF OBJECTIVITY, INC.
//
// The copyright notice above does not evidence any actual
// or intended publication of such source code.
//
////////////////////////////////////////////////////////////////////////

package PersistentInterface ;

import com.objy.db.*;     //Import Objectivity for Java exceptions
import com.objy.db.app.*; //Import Objectivity for Java classes
import com.objy.db.iapp.*;
import java.util.*;

// Make class persistence capable
public class Vehicle implements Persistent {
    // Persistor field
    private transient PooObj persistor;

    // Persistent fields
    protected String license;
    protected String type;
    protected int doors;
    protected int transmission;
    protected boolean available;
    protected Fleet fleet;

    // Legal values for transmission field
    public static final int MANUAL = 0;
    public static final int AUTOMATIC = 1;

    // Transient field
    protected transient int dailyRate;

    // Constructor
    public Vehicle(String license, String type,
                   int doors, int transmission, int rate) {
        this.license = license;
        this.type = type;
        this.doors = doors;
        this.transmission = transmission;
        this.available = true;
        this.dailyRate = rate;
    }
```

```
// Return this object's persistor
public synchronized PooObj getPersistor() {
    return persistor;
}

// Store this object's persistor
public synchronized void setPersistor(PooObj persistor) {
    this.persistor = persistor;
}

// All access methods must be called during a transaction

// Field access methods to get persistent field values
public String getLicense() {
    Delegator.fetch(this.getPersistor());;
    return this.license;
}

public String getType() {
    Delegator.fetch(this.getPersistor());;
    return this.type;
}

public int getDoors() {
    Delegator.fetch(this.getPersistor());;
    return this.doors;
}

public int getTransmission() {
    Delegator.fetch(this.getPersistor());;
    return this.transmission;
}

public boolean isAvailable() {
    Delegator.fetch(this.getPersistor());;
    return this.available;
}

public Fleet getFleet() {
    Delegator.fetch(this.getPersistor());;
    return this.fleet;
}

// Field access methods to set fleet field
public void setFleet(Fleet fleet) {
    Delegator.markModified(this.getPersistor());;
```

```
    this.fleet = fleet;
}

// Field access method to set available field
public void rentVehicle() {
    Delegator.markModified(this.getPersistor());;
    this.available = false;
}

public void returnVehicle() {
    Delegator.markModified(this.getPersistor());;
    this.available = true;
}

public void delete() {
    Delegator.delete(this.getPersistor());;
}

// Utility to print description of new vehicle
public String toShortString() {
    // This method must be called during a transaction
    Delegator.fetch(this.getPersistor());;
    StringBuffer buffer = new StringBuffer();
    buffer.append("License:" + license);
    buffer.append(" Class:" + type);
    buffer.append(" Doors:" + doors);
    if (transmission == MANUAL)
        buffer.append(" MANUAL");
    else
    buffer.append(" AUTOMATIC");
    buffer.append(" Rate:" + dailyRate);
    String strng = new String(buffer);
    return strng;
}

// Utility to print description of this vehicle
public String toString() {
    // This method must be called during a transaction
    Delegator.fetch(this.getPersistor());;
    StringBuffer buffer = new StringBuffer();
    if (available)
        buffer.append("AVAILABLE");
    else
        buffer.append("RENTED");
    buffer.append(" License:" + license);
    buffer.append(" Class:" + type);
    buffer.append(" Doors:" + doors);
```

```
        if (transmission == MANUAL)
            buffer.append(" MANUAL");
        else
        buffer.append(" AUTOMATIC");
        buffer.append(" Rate:" + dailyRate);
        String strng = new String(buffer);
        return strng;
    }

    // Static utility to obtain properties maintained by
    // Objectivity for Java
    public static void printInfo(Vehicle vehicle) {
        // This method must be called during a transaction

        // Get vehicle's container
       ooContObj cont = Delegator.getContainer(vehicle.getPersistor());
        // Get vehicle's database
        ooDBObj db = cont.getDB();
        // Get vehicle's federated database
        ooFDObj fd = db.getFD();
        // Get vehicle's session
        Session session = Delegator.getSession(vehicle.getPersistor());
        // Get vehicle's object identifier
        ooId oid = Delegator.getOid(vehicle.getPersistor());

        // Print the location in the storage hierarchy
        System.out.println("Vehicle " + vehicle.getLicense() +
                           " has OID " + oid.getStoreString() +
                           " and is stored in:");
        String nullStr = "";
        if (cont.getName() == nullStr) {
            System.out.println("    an unnamed container in");
        }
        else {
           System.out.println("   container " + cont.getName() + " in");
        }
        System.out.println("        database " + db.getName() + " in");
      System.out.println("         federated database " + fd.getName());

        // Print information about the session
        String access, txType;
        if (session.getOpenMode() == oo.openReadOnly) {
            access = "read only";
        }
        else {
            access = "read/write";
        }
```

```
            if (session.getMrowMode() == oo.MROW) {
                txType = "MROW";
            }
            else {
                txType = "nonMROW";
            }
            System.out.println("The session is open for " +
                               access + " access by " +
                               txType + "transactions");
        }

        public void activate(ActivateInfo activateInfo) {
            // Handle fetch errors
            if (activateInfo.hasFetchErrors()) {
                // Mark modified so that new values of persistent fields
                // are written to the database.
                Delegator.markModified(this.getPersistor());
                // Retrieve vector of fetch errors
                Vector errors = activateInfo.getFetchErrors();
                Enumeration errs = errors.elements();
                while (errs.hasMoreElements()) {
                    FetchErrorInfo ffi =
                        (FetchErrorInfo)errs.nextElement();
                    // Get components of fetch failed information object
                    // and print error message.
                    String fieldName = ffi.getFieldName();
                    System.out.println(ffi.getErrorMessage());
                    // Set persistent field at which error occurred
                    if (fieldName.equals("fleet")) {
                        try {
                            fleet =

    (Fleet)Delegator.getContainer(this.getPersistor()).getDB().lookup("Fleet");
                        }
                        catch(ObjectNameNotFoundException e) {
                            System.out.println
                            ("\nCouldn't set fleet for vehicle: "
                            + toString());
                        }
                    }
                }
            }
            // Set default transient field
            dailyRate = doors * 10;
        }
    } // End Vehicle class
```

Delegator.java

```
//////////////////////////////////////////////////////////////////////
//
// Delegator - delegates explicit persistence behavior to a persistor
//
// Copyright (c) 1998 Objectivity, Inc.  All Rights Reserved
//
// UNPUBLISHED PROPRIETARY SOURCE CODE OF OBJECTIVITY, INC.
//
// The copyright notice above does not evidence any actual
// or intended publication of such source code.
//
//////////////////////////////////////////////////////////////////////

package PersistentInterface ;

import com.objy.db.*;     //Import Objectivity for Java exceptions
import com.objy.db.app.*; //Import Objectivity for Java classes
import com.objy.db.iapp.PooObj;

public class Delegator {

   // Checking that the persistor can perform the requested operation
            private static synchronized PooObj validPersistor(PooObj
persistor) {
        if (persistor == null)
           throw new ObjectNotPersistentException("Attempted persistent
operation on transient object") ;

        if (persistor.isDead())
       throw new ObjectIsDeadException("Attempted persistent operation on
dead object") ;
        return persistor ;
    }
            private static synchronized PooObj notDeadPersistor(PooObj
persistor) {
        if (persistor.isDead())
       throw new ObjectIsDeadException("Attempted persistent operation on
dead object") ;
        return persistor ;
    }

    //
    // Getting Information
```

```
        //
        public static ooContObj getContainer(PooObj persistor)
            { return validPersistor(persistor).getContainer() ; }

        public static Session getSession(PooObj persistor)
            { return validPersistor(persistor).getSession() ; }
//           { return (persistor == null) ? null : persistor.getSession(); }

        public static ooId getOid(PooObj persistor)
            { return validPersistor(persistor).getOid() ; }

        //
        // Testing
        //
        public static boolean isDead(PooObj persistor)
            { return (persistor == null) ? false : persistor.isDead(); }

        public static boolean isModified(PooObj persistor)
            { return (persistor == null) ? false :
                      notDeadPersistor(persistor).isModified(); }

        public static boolean isFetchRequired(PooObj persistor)
            { return (persistor == null) ? false :
               notDeadPersistor(persistor).isFetchRequired(); }

        public static boolean isPersistent(PooObj persistor)
            { return (persistor == null) ? false :
               notDeadPersistor(persistor).isPersistent(); }

        public static boolean isValid(PooObj persistor) {
            if ( isDead(persistor) ) return false;
            return persistor.isValid();
        }

        //
        // Persistence
        //
        public static void markFetchRequired(PooObj persistor)
            { validPersistor(persistor).markFetchRequired(); }

        public static void markModified(PooObj persistor)
        { if (persistor != null) notDeadPersistor(persistor).markModified() ; }

        public static void clearModified(PooObj persistor)
       { if (persistor != null) notDeadPersistor(persistor).clearModified() ; }

        public static void fetch(PooObj persistor)
```

```
    { if (persistor != null) notDeadPersistor(persistor).fetch() ; }

public static void fetch(PooObj persistor, int mode)
  { if (persistor != null) notDeadPersistor(persistor).fetch(mode) ; }

public static void write(PooObj persistor)
    { validPersistor(persistor).write() ; }

//
// Locking
//
public static void lock(PooObj persistor, int mode)
    { validPersistor(persistor).lock(mode) ; }

public static void lockNoProp(PooObj persistor, int mode)
    { validPersistor(persistor).lockNoProp(mode) ; }

//
// Deleting
//
public static void delete(PooObj persistor)
    { validPersistor(persistor).delete() ; }

public static void deleteNoProp(PooObj persistor)
    { validPersistor(persistor).deleteNoProp() ; }

//
// Clustering
//
public static void cluster(PooObj persistor, Object object)
    { validPersistor(persistor).cluster(object) ; }

//
// Copying
//
public static Object copy(PooObj persistor, Object near)
    { return validPersistor(persistor).copy(near) ; }

//
// Moving
//
public static void move(PooObj persistor, Object near)
    { validPersistor(persistor).move(near) ; }

//
// Working With Scope Named Objects
//
```

```
    public static void nameObj(PooObj persistor,
                               Object object, String scopeName)
        { validPersistor(persistor).nameObj(object, scopeName) ; }

    public static void unnameObj(PooObj persistor, Object object)
        { validPersistor(persistor).unnameObj(object) ; }

    public static Object lookupObj(PooObj persistor, String scopeName)
        { return validPersistor(persistor).lookupObj(scopeName) ; }

  public Object lookupObj(PooObj persistor, String scopeName, int lockMode)
    { return validPersistor(persistor).lookupObj(scopeName, lockMode) ; }

    public static String lookupObjName(PooObj persistor, Object object)
        { return validPersistor(persistor).lookupObjName(object) ; }

    public static Iterator scopedObjects(PooObj persistor)
        { return validPersistor(persistor).scopedObjects() ; }

    public static Iterator scopedBy(PooObj persistor)
        { return validPersistor(persistor).scopedBy() ; }

    public static void updateIndexes(PooObj persistor)
        { validPersistor(persistor).updateIndexes() ; }

}
```

28

Naming and Retrieving Objects

The examples discussed in Chapter 10, "Naming Persistent Objects," and Chapter 11, "Retrieving Persistent Objects," illustrate how to name and retrieve a persistent object. The examples in those chapters come from two packages:

- The `Sales` package illustrates three alternative ways for naming an object and shows how to look up objects by each kind of name. It also illustrates how to find objects by following a to-many relationship from a retrieved object.
- The `Traversal` package illustrates how to retrieve objects by traversing the storage hierarchy.

In This Chapter

Sales Package

The `Sales` package contains four classes:

- The `Interact` class contains static utility methods that are called to create and retrieve objects of the other three classes. `Interact` can be executed after you compile all the classes and create a federated database named `"Sales"` in the `NamingAndRetrieving` subdirectory of the samples directory.
- Every salesperson of the `Salesperson` class (see page 435) is a named root in the `Sales` database and can be looked up by its root name. Methods of this class allow you to retrieve the contacts related to a particular salesperson.
- Every contact of the `Contact` class (see page 438) is given a scope name in the scope of the `ContactNames` container in the `Sales` database. Contacts can be looked up by their scope names.
- Every client of the `Client` class (see page 440) is given a name in an application-specific name table. The name table itself is a named root in the `Corporate` federated database.

Interact.java

```
//////////////////////////////////////////////////////////////////////
//
// Sales.Interact - provides static utilities for creating and
// retrieving objects in a corporate database.
//
// Copyright (c) 1998 Objectivity, Inc.  All Rights Reserved
//
// UNPUBLISHED PROPRIETARY SOURCE CODE OF OBJECTIVITY, INC.
//
// The copyright notice above does not evidence any actual
// or intended publication of such source code.
//
//////////////////////////////////////////////////////////////////////

package Sales;
import com.objy.db.*;     // Import Objectivity for Java exceptions
import com.objy.db.app.*; // Import Objectivity for Java classes
import com.objy.db.util.ooMap;

public class Interact {
    private static String fdName = "Corporate";
    private static Connection connection;
    private static Session session;
    private static boolean initialized;
```

```
    public static void main(String args[]) {
        ooFDObj corporateFD = getCorporateFD();
        if (corporateFD == null) {
            return;
        }
        // Create some objects in the federated database
        Salesperson s1 = addSalesperson("Jenny",  "Ace",  1);
        Salesperson s2 = addSalesperson("George",  "Flyer",  2);
    addClient("Alpha", "100 San Antonio Rd.", "Mountain View", "CA", 94040,
s1);
       addClient("Omega", "10 Park Ave.", "New York", "NY", 10021, s2);
        addContact("Jane", "Edwards", "Alpha");
        addContact("Jack", "Angel", "Omega");
        printClientRepresentative("Alpha");
        printSalespersonForContact("Jack", "Angel", "Omega");
        printContactsForSalesperson(1);
        printClientsInState("CA");

        try {
            connection.close();
        } catch (DatabaseClosedException e3) {
            System.out.println("Connection already closed.");
        }
    } // End main method

    // Static utility to open a connection to the corporate
    // federated database and initialize user interactions
    public static ooFDObj getCorporateFD() {
       if (initialized) {
            return session.getFD();
        }
        else {
            try {
                connection = Connection.open(fdName, oo.openReadWrite);
            } catch (DatabaseNotFoundException e1) {
                System.out.println("Federated database " + fdName +
                                   " cannot be found");
                return null;
            } catch (DatabaseOpenException e2) {
                System.out.println("The connection is already open");
                return null;
            }
            // Create session object to be used by all
            // static methods of Interact.  All persistent objects
            // created or retrieved because of request by user
            // will belong to this session.
            session = new Session();
```

```
            initialized = true;
            return session.getFD();
        }
    } // End getCorporateFD method

    // Static utility to get the sales database
    public static ooDBObj getSalesDB(ooFDObj corporateFD) {
        // This method must be called during a transaction

        ooDBObj salesDB;
        if (corporateFD.hasDB("Sales")) {
            salesDB = corporateFD.lookupDB("Sales");
        }
        else {
            salesDB = corporateFD.newDB("Sales");
        }
        return salesDB;
    } // End getSalesDB method

    // Static utility to add a salesperson
    public static Salesperson addSalesperson (
        String firstName,  // First name of new salesperson
        String lastName,   // Last name of new salesperson
        int employeeID)    // Employee ID of new salesperson
    {
        ooContObj cont;

        // Get the connected federated database.
        ooFDObj corporateFD = getCorporateFD();
        if (corporateFD == null) {
            return null;
        }

        session.begin();
        // Get sales database
        ooDBObj salesDB = getSalesDB(corporateFD);

        // Create new salesperson and cluster into
        // specified container
        Salesperson salesperson = new Salesperson(firstName,
                                                  lastName,
                                                  employeeID);
        String contName = "Salespeople";
        if (contName != null) {
            if (salesDB.hasContainer(contName)) {
                cont = (ooContObj)salesDB.lookupContainer(contName);
            }
```

```
            else {
                // Create a non-garbage-collectible container
                cont = new ooContObj();
                // Make the container persistent (and unhashed)
                salesDB.addContainer(cont, contName, 0, 5, 10);
            }
            cont.cluster(salesperson);
        }
        // Create rootname from employee ID
        String rootname = String.valueOf(employeeID);

        // Make new salesperson a named root
        try {
            salesDB.bind(salesperson, rootname);
        } catch (ObjectNameNotUniqueException e4) {
            System.out.println("Employee ID " + rootname +
                               " already in use");
            session.abort();
            return null;
        }
        session.commit();
        return salesperson;
    } // End addSalesperson method

    // Static utility to add a contact
    public static Contact addContact (
        String firstName,  // First name of new contact
        String lastName,   // Last name of new contact
        String company)    // Company of new contact
    {
        ooContObj cont;
        ooContObj scope;

        // Get the connected federated database.
        ooFDObj corporateFD = getCorporateFD();
        if (corporateFD == null) {
            return null;
        }

        Client client = lookupClient(company);

        session.begin();
        // Get sales database
        ooDBObj salesDB = getSalesDB(corporateFD);
        String contName = "Contacts";

        // Create new contact and cluster into specified container
```

```
        Contact contact = new Contact(firstName, lastName, company);
        if (contName != null) {
            if (salesDB.hasContainer(contName)) {
                cont = (ooContObj)salesDB.lookupContainer(contName);
            }
            else {
                // Create a non-garbage-collectible container
                cont = new ooContObj();
                // Make the container persistent (and unhashed)
                salesDB.addContainer(cont, contName, 0, 5, 10);
            }
            cont.cluster(contact);
        }

        // Give new contact a scope name
        if (salesDB.hasContainer("ContactNames")) {
            // Get container used as scope object
            scope = salesDB.lookupContainer("ContactNames");
        }
        else {
            // Create a non-garbage-collectible container
            scope = new ooContObj();
            // Make the container persistent (and hashed)
            salesDB.addContainer(scope, "ContactNames", 5, 5, 10);
        }
        // Create scope name from name and company
        String scopename = firstName + " " + lastName + " at " +
                           company;
        // Give the contact a scope name
        try {
            scope.nameObj(contact, scopename);
        } catch (ObjyRuntimeException e) {
            System.out.println("Unable to add " + scopename +
                               " as a scope name.");
            session.abort();
            return null;
        }

        Salesperson salesperson = client.getSalesRep();
        salesperson.addContact(contact);
        session.commit();
        return contact;
    } // end addContact method

    // Static utility to add a client
    public static Client addClient (
        String companyName,   // Company name of new client
```

```
        String address,       // Street address of new client
        String city,          // City of new client
        String state,         // State of new client
        int zipCode,          // Zip code of new client
        Salesperson salesRep) // Sales representative
    {
        ooContObj clientCont;
        ooMap nameTable;

        // Get the connected federated database.
        ooFDObj corporateFD = getCorporateFD();
        if (corporateFD == null) {
            return null;
        }

        session.begin();
        // Get sales database
        ooDBObj salesDB = getSalesDB(corporateFD);

        // Get the container for clients
        if (salesDB.hasContainer("Clients")) {
            clientCont = salesDB.lookupContainer("Clients");
        }
        else {
            // Create a non-garbage-collectible container
            clientCont = new ooContObj();
            // Make the container persistent (and unhashed)
            salesDB.addContainer(clientCont, "Clients", 0, 5, 10);
            // Create an index of clients by geographical location
            clientCont.addIndex("By Location",  // name of new index
                                "Sales.Client", // class of indexed objects
                                   "state, city, zipCode");  // key fields
       }
        // Create new client and cluster into the client container
        Client client = new Client(companyName, address, city, state,
                                   zipCode, salesRep);
        clientCont.cluster(client);

        // Create or retrieve the name table
        try {
            nameTable = (ooMap) salesDB.lookup("ClientNames");
        } catch (ObjectNameNotFoundException e) {
            nameTable = new ooMap();
            clientCont.cluster(nameTable);
            try {
                salesDB.bind(nameTable, "ClientNames");
            } catch (ObjectNameNotUniqueException e4) {
```

```
                System.out.println
                    ("Can't retrieve or create the client name table");
                session.abort();
                return null;
            }
        }
        // Add client to name table
        try {
            nameTable.add(client, companyName);
        } catch (ObjyRuntimeException e) {
            System.out.println("Company " + companyName +
                               " already exists");
            session.abort();
            return null;
        }

        session.commit();
        return client;
    } // End addClient method

    // Static utility to retrieve a salesperson
    public static Salesperson lookupSalesperson (int employeeID) {
        Salesperson salesperson;

        // Get the connected federated database.
        ooFDObj corporateFD = getCorporateFD();
        if (corporateFD == null) {
            return null;
        }

        session.begin();
        // Get sales database
        ooDBObj salesDB = getSalesDB(corporateFD);

        // Create rootname from employee ID
        String rootname = String.valueOf(employeeID);
        // Look up the salesperson
        try {
            salesperson = (Salesperson) salesDB.lookup(rootname);
        } catch (ObjectNameNotFoundException e) {
            System.out.println("No salesperson with Employee ID " +
                               rootname );
            session.abort();
            return null;
        }
        session.commit();
        return salesperson;
```

```
} // End lookupSalesperson method

// Static utility to retrieve a contact
public static Contact lookupContact (
    String firstName,  // First name of contact
    String lastName,   // Last name of contact
    String company)    // Company of contact
{
    Contact contact;

    // Get the connected federated database.
    ooFDObj corporateFD = getCorporateFD();
    if (corporateFD == null) {
        return null;
    }

    session.begin();
    // Get sales database
    ooDBObj salesDB = getSalesDB(corporateFD);

    // Create scope name from name and company
    String scopename = firstName + " " + lastName + " at " +
                       company;

    // lookup the contact
    if (salesDB.hasContainer("ContactNames")) {
        // Get container used as scope object
        ooContObj scope = salesDB.lookupContainer("ContactNames");
        try {
            contact = (Contact) scope.lookupObj(scopename);
        } catch (ObjyRuntimeException e) {
            System.out.println("No contact named " + scopename);
            session.abort();
            return null;
        }
    }
    else {
        System.out.println
                    ("Contact Names container does not exist");
        session.abort();
        return null;
    }
    session.commit();
    return contact;
} // End lookupContact method

// Static utility to retrieve a client
```

```
public static Client lookupClient (String companyName) {
    ooMap nameTable;

    // Get the connected federated database.
    ooFDObj corporateFD = getCorporateFD();
    if (corporateFD == null) {
        return null;
    }

    session.begin();
    // Get sales database
    ooDBObj salesDB = getSalesDB(corporateFD);

    // Retrieve name table
    try {
        nameTable = (ooMap)salesDB.lookup("ClientNames");
    } catch (ObjectNameNotFoundException e) {
        System.out.println("Client name table does not exist");
        session.abort();
        return null;
    }

    // Look up name in table
    if (nameTable.isMember(companyName)) {
        Client client = (Client)nameTable.lookup(companyName);
        session.commit();
        return client;
    }
    else {
        System.out.println("No client named " +
                           companyName);
        session.abort();
        return null;
    }
} // End lookupClient method

// Static utility to print the name of a client company's
// sales representative
public static void printClientRepresentative (String companyName) {

    // First retrieve the client
    // Note that lookupClient starts its own transaction and
    // prints a message if the specified company is not found
    Client client = Interact.lookupClient(companyName);
    if (client == null) {
        return;
    }
```

```
        session.begin();
        // Get the client's sales representative
        Salesperson salesPerson = client.getSalesRep();
        if (salesPerson == null) {
            System.out.println("No sales representative for " +
                               companyName);
        }
        else {
            System.out.println("Sales representative for " +
                               companyName + " is " +
                               salesPerson.getFirstName() + " " +
                               salesPerson.getLastName());
        }
        session.commit();
    } // End printClientRepresentative method

    // Static utility to print the name of a contact's salesperson
    public static void printSalespersonForContact (
        String firstName,  // First name of contact
        String lastName,   // Last name of contact
        String company)    // Company of contact
    {

        // First retrieve the contact
        // Note that lookupContact starts its own transaction and
        // prints a message if the specified contact is not found
        Contact contact = Interact.lookupContact
                           (firstName, lastName, company);
        if (contact == null) {
            return;
        }
        session.begin();
        // Get the contact's salesperson
        Salesperson salesPerson = contact.getSalesperson();
        if (salesPerson == null) {
            System.out.println("No salesperson for " +
                               firstName + " " + lastName);
        }
        else {
            System.out.println("Salesperson for " +
                               firstName + " " +
                               lastName + " is " +
                               salesPerson.getFirstName() + " " +
                               salesPerson.getLastName());
        }
        session.commit();
    } // End printSalespersonForContact method
```

```
// Static utility to print the names of a salesperson's contacts
public static void printContactsForSalesperson(int employeeID) {

    // First retrieve the salesperson
    // Note that lookupSalesperson starts its own transaction and
    // prints a message if the specified salesperson is not found
   Salesperson salesperson = Interact.lookupSalesperson(employeeID);
    if (salesperson == null) {
        return;
    }
    session.begin();
    // Get the salesperson's contacts
    Iterator itr = salesperson.getAllContacts();
     if (!itr.hasNext()) {
        System.out.println("No contacts");
        session.commit();
        return;
    }
    Contact contact;
    System.out.println("Contacts for " +
                       salesperson.getFirstName() +
                       " " + salesperson.getLastName()+ " are: ");
    while (itr.hasNext()) {
        // Cast to Contact
        contact = (Contact)itr.next();
        contact.printName();
    }
    session.commit();
} // End printContactsForSalesperson method

// Static utility to print the names of all sales people
public static void printAllSalesPeople() {

    // Get the connected federated database.
    ooFDObj corporateFD = getCorporateFD();
    if (corporateFD == null) {
        return;
    }

    session.begin();
    // Get sales database
    ooDBObj salesDB = getSalesDB(corporateFD);
    Iterator itr = salesDB.scan("Sales.Salesperson");
    if (!itr.hasNext()) {
        System.out.println("No sales people");
        session.commit();
```

```
            return;
        }
        Salesperson salesperson;
        System.out.println("Salespeople are: ");
        while (itr.hasNext()) {
            // cast to Salesperson
            salesperson = (Salesperson)itr.next();
            salesperson.printName();
        }
        session.commit();
    } // End printAllSalesPeople method

    // Static utility to print the clients in the specified state
    public static void printClientsInState(String state) {
        ooContObj clientCont;

        // Get the connected federated database
        ooFDObj corporateFD = getCorporateFD();
        if (corporateFD == null) {
            return;
        }

        session.begin();
        // Get sales database
        ooDBObj salesDB = getSalesDB(corporateFD);
        // Get the container for clients
        if (salesDB.hasContainer("Clients")) {
            clientCont = salesDB.lookupContainer("Clients");
        }
        else {
            System.out.println("No clients");
            session.commit();
            return;
        }
        String predicate = new String(
                          "state == \"" + state + "\"");
        Iterator itr = clientCont.scan("Sales.Client", predicate);
        if (!itr.hasNext()) {
            System.out.println("No clients in " + state);
            session.commit();
            return;
        }
        Client client;
        System.out.println("Clients in state " + state +  " are: ");
        while (itr.hasNext()) {
            // Cast to Client
            client = (Client)itr.next();
```

```
                System.out.println(client.getCompanyName());
            }
            session.commit();
        } // End printClientsInState method

    } // Endd Interact class
```

Salesperson.java

```
//////////////////////////////////////////////////////////////////////
//
// Sales.Salesperson - a salesperson
//
// Copyright (c) 1998 Objectivity, Inc.  All Rights Reserved
//
// UNPUBLISHED PROPRIETARY SOURCE CODE OF OBJECTIVITY, INC.
//
// The copyright notice above does not evidence any actual
// or intended publication of such source code.
//
//////////////////////////////////////////////////////////////////////

package Sales;
import com.objy.db.*;     // Import Objectivity for Java exceptions
import com.objy.db.app.*; // Import Objectivity for Java classes

public class Salesperson extends ooObj {

    // Persistent Fields
    protected String firstName;
    protected String lastName;
    protected int employeeID;

    // Relationships
    private ToManyRelationship contacts;
    public static OneToMany contacts_Relationship() {
        return new OneToMany(
            "contacts",                // This relationship
            "Sales.Contact",           // Related class
            "salesperson",             // Inverse relationship
            Relationship.COPY_MOVE,    // Move to copy of object
            Relationship.VERSION_MOVE, // Move to new version
            false,                     // Don't propagate delete
            false,                     // Don't propagate locks
            Relationship.INLINE_NONE); // Store non-inline
    }

    // Constructor
    public Salesperson(String firstName,
                       String lastName,
                       int employeeID)
    {
        this.firstName = firstName;
        this.lastName = lastName;
```

```
        this.employeeID = employeeID;
    }

    // All access methods must be called during a transaction

    // Field Access Methods
    public String getFirstName() {
        fetch();
        return this.firstName;
    }
    public String getLastName() {
        fetch();
        return this.lastName;
    }
    public int getEmployeeID() {
        fetch();
        return this.employeeID;
    }

    // Relationship access methods
    public void addContact(Contact newContact) {
        fetch();
        this.contacts.add(newContact);
    }

    public Contact findContact(String firstName,
                               String lastName,
                               String company) {
        fetch();
        String predicate = new String(
                          "company == \"" + company + "\" and " +
                          "lastName == \"" + lastName + "\" and " +
                          "firstName == \"" + firstName + "\"");
        Iterator itr = this.contacts.scan(predicate);
        if (itr.hasNext()) {
            return (Contact)itr.next(); // Cast to Contact
        }
        else {
            return null;
        }
    }

    public Iterator findContactsByCompany (String company) {
        fetch();
        String predicate = new String("company == \"" + company + "\"");
        return this.contacts.scan(predicate);
    }
```

```
    public Iterator getAllContacts () {
        fetch();
        return this.contacts.scan();
    }

    // Utility method to print this salesperson's name
    public void printName() {
        // This method must be called during a transaction
        System.out.println(getFirstName() + " " + getLastName());
    }

} // End Salesperson class
```

Contact.java

```
//////////////////////////////////////////////////////////////////////
//
// Sales.Contact - a salesperson's contact in some client company
//
// Copyright (c) 1998 Objectivity, Inc.  All Rights Reserved
//
// UNPUBLISHED PROPRIETARY SOURCE CODE OF OBJECTIVITY, INC.
//
// The copyright notice above does not evidence any actual
// or intended publication of such source code.
//
//////////////////////////////////////////////////////////////////////

package Sales;
import com.objy.db.*;     // Import Objectivity for Java exceptions
import com.objy.db.app.*; // Import Objectivity for Java classes

public class Contact extends ooObj {
    // Persistent Fields
    protected String firstName;
    protected String lastName;
    protected String company;

    // Relationships
    private ToOneRelationship salesperson;
    public static ManyToOne salesperson_Relationship() {
        return new ManyToOne(
            "salesperson",              // This relationship
            "Sales.Salesperson",        // Related class
            "contacts",                 // Inverse relationship
            Relationship.COPY_COPY,     // Copy relationship to copy
            Relationship.VERSION_COPY,  // Copy to new version
            false,                      // Don't propagate delete
            false,                      // Don't propagate locks
            Relationship.INLINE_NONE);  // Store non-inline
    }

    // Constructor
    public Contact(String firstName,
                   String lastName,
                   String company){
        this.firstName = firstName;
        this.lastName = lastName;
        this.company = company;
    }
```

```
    // All access methods must be called during a transaction

    // Field Access Methods
    public String getFirstName() {
        fetch();
        return this.firstName;
    }
    public String getLastName() {
        fetch();
        return this.lastName;
    }
    public String getCompany() {
        fetch();
        return this.company;
    }

    // Relationship access method
    public void setSalesperson(Salesperson newSalesperson) {
        fetch();
        // Remove any existing relationship
        this.salesperson.clear();
        this.salesperson.form(newSalesperson);
    }

    public Salesperson getSalesperson() {
        fetch();
        // Cast retrieved object to class Salesperson
        return (Salesperson)this.salesperson.get();
    }

    // Utility method to print this contact's name
    public void printName() {
        // This method must be called during a transaction
        System.out.println(getFirstName() + " " + getLastName());
    }

} // End Contact class
```

Client.java

```
//////////////////////////////////////////////////////////////////////
//
// Sales.Client - a client company
//
// Copyright (c) 1998 Objectivity, Inc.  All Rights Reserved
//
// UNPUBLISHED PROPRIETARY SOURCE CODE OF OBJECTIVITY, INC.
//
// The copyright notice above does not evidence any actual
// or intended publication of such source code.
//
//////////////////////////////////////////////////////////////////////

package Sales;
import com.objy.db.*;     // Import Objectivity for Java exceptions
import com.objy.db.app.*; // Import Objectivity for Java classes

public class Client extends ooObj {
    // Persistent Fields
    protected String companyName;
    protected String address;
    protected String city;
    protected String state;
    protected int zipCode;
    protected Salesperson salesRep;

    // Constructor
    public Client(String companyName, String address, String city,
                  String state, int zipCode, Salesperson salesRep)
    {
        this.companyName = companyName;
        this.address = address;
        this.city = city;
        this.state = state;
        this.zipCode = zipCode;
        this.salesRep = salesRep;
    }

    // All access methods must be called during a transaction

    // Field access methods
    public String getCompanyName() {
        fetch();
        return this.companyName;
    }
```

```
    public String getAddress() {
        fetch();
        return this.address;
    }

    public String getCity() {
        fetch();
        return this.city;
    }

    public String getState() {
        fetch();
        return this.state;
    }

    public int zipCode() {
        fetch();
        return this.zipCode;
    }

    public Salesperson getSalesRep() {
        fetch();
        return this.salesRep;
    }

} // end Client class
```

Traversal Package

The `Traversal` package contains three classes, but the only one with methods is `Tester`. `Tester` can be executed after you compile all the classes and create a federated database named `"Traverse"` in the `NamingAndRetrieving` subdirectory of the samples directory.

- The `Tester.createObjects` static method creates databases, containers, and basic objects of the classes `Apple` and `Orange` in the connected federated database.
- The `Tester.traverse` static method illustrates how to traverse the storage hierarchy, retrieving every object in the federated database.
- The `Tester.retrieveAgain` static method illustrates how to retrieve an object by its OID. This method is called from `Tester.traverse` to illustrate that each session can have its own local representation of an object.
- The `Tester.main` static method creates objects in the federated database by calling `Tester.createObjects` and traverses the storage hierarchy by calling `Tester.traverse`.

Tester.java

```
//////////////////////////////////////////////////////////////////////
//
// Traversal.Tester - provides static utilities for populating a
// federated database and traversing the storage hierarchy
//
// Copyright (c) 1998 Objectivity, Inc.  All Rights Reserved
//
// UNPUBLISHED PROPRIETARY SOURCE CODE OF OBJECTIVITY, INC.
//
// The copyright notice above does not evidence any actual
// or intended publication of such source code.
//
//////////////////////////////////////////////////////////////////////

package Traversal;
import com.objy.db.*;     // Import Objectivity for Java exceptions
import com.objy.db.app.*; // Import Objectivity for Java classes

public class Tester {
    private static String fdName = "Traverse" ;
    private static Session session;
    private static boolean initialized;

    public static void main(String args[]) {
```

```
        Connection connection;
        ooFDObj fd;
        if (initialized) {
            fd = session.getFD();
            connection = Connection.current();
        }
        else {
            try {
                connection = Connection.open(fdName, oo.openReadWrite);
            } catch (DatabaseNotFoundException e1) {
                System.out.println("Federated database " + fdName +
                                   " cannot be found");
                return;
            } catch (DatabaseOpenException e2) {
                System.out.println("The connection is already open");
                return;
            }
            // Create session object to be used for all
            //   persistent operations.  All persistent objects
            //   created or retrieved will belong to this session.
            session = new Session();
            fd = session.getFD();
            initialized = true;
        }
        // Create some objects in the federated database
        createObjects(fd);
        // Traverse the storage hierarchy
        traverse(fd);
        // Close the interaction
        try {
            connection.close();
        } catch (DatabaseClosedException e3) {
            System.out.println("Connection already closed.");
            return;
        }
    } // End main method

    public static void createObjects(ooFDObj fd) {
        ooDBObj db;
        ooContObj cont;
        Apple apple;
        Orange orange;

        session.begin();
        if (!fd.hasDB("A")) {
            db = fd.newDB("A");
            apple = new Apple();
```

```
            db.cluster(apple);
            orange = new Orange();
            db.cluster(orange);
            cont = new ooContObj();
            db.addContainer(cont, "apples", 0, 5, 10);
            for (int i = 1; i < 4; i++) {
                apple = new Apple();
                cont.cluster(apple);
            }
            apple = new Apple();

            cont = new ooContObj();
            db.addContainer(cont, "oranges", 0, 5, 10);
            for (int i = 1; i < 3; i++) {
                orange = new Orange();
                cont.cluster(orange);
            }
            cont = new ooContObj();
            db.addContainer(cont, "mixed fruits", 0, 5, 10);
            orange = new Orange();
            cont.cluster(orange);
            apple = new Apple();
            cont.cluster(apple);
        }
        if (!fd.hasDB("B")) {
            db = fd.newDB("B");
            apple = new Apple();
            db.cluster(apple);
            orange = new Orange();
            db.cluster(orange);
            cont = new ooContObj();
            db.addContainer(cont, "more apples", 0, 5, 10);
            for (int i = 1; i < 3; i++) {
                apple = new Apple();
                cont.cluster(apple);
            }
            cont = new ooContObj();
            db.addContainer(cont, "more oranges", 0, 5, 10);
            for (int i = 1; i < 4; i++) {
                orange = new Orange();
                cont.cluster(orange);
            }
            cont = new ooContObj();
            db.addContainer(cont, "more mixed fruits", 0, 5, 10);
            orange = new Orange();
            cont.cluster(orange);
            apple = new Apple();
```

```
            cont.cluster(apple);
        }
        session.commit();
    } // End createObjects method

    public static void traverse(ooFDObj fd) {
        ooDBObj db;
        ooContObj cont;
        ooObj basicObject;
        Iterator dbItr;
        Iterator contItr;
        Iterator objItr;
        String indent = "";
        String contName;
        ooId basicOID;

        // Use an MROW transaction so other transactions
        // can make modifications
        session.setMrowMode(oo.MROW);
        basicOID = null;
        basicObject = null;
        session.begin();
        // Print system name of federated database
        System.out.println(fd.getName());
        // Get all databases
        dbItr = fd.containedDBs();
        while (dbItr.hasNext()) {
            db = (ooDBObj)dbItr.next(); // cast to ooDBObj
            indent = "    ";
            // Print system name of current database
            System.out.println(indent + db.getName());
            // Get all containers in current database
            contItr = db.contains();
            while (contItr.hasNext()) {
                // cast to ooContObj
                cont = (ooContObj)contItr.next();
                indent = "        ";
                // Print system name of current container (if any)
                contName = cont.getName();
                if (cont != null)
                    System.out.println(indent + contName);
                else
                    System.out.println(indent + "(container)");
                // Get all objects in current container
                objItr = cont.contains();
                indent = "            ";
                while (objItr.hasNext()) {
```

```
                    // cast to ooObj
                    basicObject = (ooObj)objItr.next();
                    // Print OID of object
                    basicOID = basicObject.getOid();
                System.out.println(indent + basicOID.getStoreString());
                } // End while more basic objects of current container
            } // End while more containers of current database
        } // End while more databases of federated database
        session.commit();
        if (basicOID != null) {
            // Examine two local representations of the same object
            session.begin();
            printInfo(basicObject);
            session.commit();
            retrieveAgain(basicOID);
        }
    } // End traverse method

    // Static utility to retrieve an object by its OID in a new session
    public static void retrieveAgain(ooId oid) {
        // Create new session
        Session session = new Session();
        session.setOpenMode(oo.openReadOnly);
        ooFDObj myFD = session.getFD();

        // Get new local representation of object with specified OID
        session.begin();
        ooObj myBasicObject = (ooObj)myFD.objectFrom(oid);
        printInfo(myBasicObject);
        session.commit();
    } // End retrieveAgain method

    // Static utility to print properties maintained by
    // Objectivity for Java
    public static void printInfo(ooObj basicObject) {
       // This method must be called during a transaction

        // Get object's container
        ooContObj cont = basicObject.getContainer();
        // Get object's database
        ooDBObj db = cont.getDB();
        // Get object's session
        Session curSession = basicObject.getSession();
        // Get object's federated database
        ooFDObj fd = curSession.getFD();
        // Get object's object identifier
        ooId oid = basicObject.getOid();
```

```
        // Print the location in the storage hierarchy
        System.out.println("Object with OID " +
                           oid.getStoreString() +
                           " is stored in:");
        if (cont.getName().equals("")) {
            System.out.println("    an unnamed container in");
        }
        else {
          System.out.println("    container " + cont.getName() + " in");
        }
        System.out.println("        database " + db.getName() + " in");
      System.out.println("         federated database " + fd.getName());

        // Print information about the session
        String access, txType;
        if (curSession.getOpenMode() == oo.openReadOnly) {
            access = "read only";
        }
        else {
            access = "read/write";
        }
        if (curSession.getMrowMode() == oo.MROW) {
            txType = "MROW";
        }
        else {
            txType = "nonMROW";
        }
        System.out.println("The session is open for " +
                           access + " access by " +
                           txType + "transactions");
    } // End of printInfo method
} // End Tester class
```

29

Clustering Objects

The examples discussed in Chapter 12, “Clustering Objects,” illustrate two application-specific clustering strategies. Both clustering strategies use container pools associated with particular databases.

In This Chapter

Container-Pool Strategy
- ContainerPool.java
- ContainerPoolStrategy.java

Cluster-By-Class Strategy
- ClusterByClassStrategy.java
- JustCreatedReason.java
- Account.java
- Employee.java
- BranchOffice.java

Container-Pool Strategy

A container pool is an object of the `ContainerPool` class. Each database has two container pools, one containing garbage-collectible containers and the other containing non-garbage-collectible containers. A database identifies its container pools with the scope names `"GCContainerPool"` and `"Non-GCContainerPool"`, respectively. When a container pool is created, parameters to its constructor specify its database, the number of containers in the pool, and whether the containers are garbage collectible.

The `ContainerPoolStrategy` class (see page 453) implements a clustering strategy that performs as follows:

- Cluster a container in the database of the requesting object.
- Cluster a basic object being made a named root in a garbage-collectible container selected from the container pool of the database of the requesting object.
- Cluster a scope-named basic object in a non-garbage-collectible container selected from the container pool of the database of the requesting object.
- Cluster any other basic object with the requesting object.

ContainerPool.java

```
//////////////////////////////////////////////////////////////////////
//
// ContainerPool - a pool of containers used for clustering objects
//
// Copyright (c) 1998 Objectivity, Inc.  All Rights Reserved
//
// UNPUBLISHED PROPRIETARY SOURCE CODE OF OBJECTIVITY, INC.
//
// The copyright notice above does not evidence any actual
// or intended publication of such source code.
//
//////////////////////////////////////////////////////////////////////

import com.objy.db.*;     // Import Objectivity for Java exceptions
import com.objy.db.app.*; // Import Objectivity for Java classes
import java.util.*;    // Import random-number generator

public class ContainerPool extends ooObj {

    // Persistent fields
    private ooContObj[] pool;
    private int containerCount;
```

```
    // Constructor creates a pool of the specified number and
    // kind of containers in the specified database
    public ContainerPool(ooDBObj db,    // Database for pool
                         int numberOfContainers,    // Number of containers
                          boolean garbageCollectible) // Type of containers
    {
        // This constructor must be used during a transaction

        ooContObj container;
        this.containerCount = numberOfContainers;
        this.pool = new ooContObj[numberOfContainers];
        if (garbageCollectible) {
            for (int i = 0; i < numberOfContainers; i++) {
                container = new ooGCContObj();
                this.pool[i] = container;
                db.addContainer(container, "", 0, 5, 10);
            }
            db.nameObj(this, "GC Container Pool");
        }
        else {
            for (int i = 0; i < numberOfContainers; i++) {
                container = new ooContObj();
                this.pool[i] = new ooContObj();
                db.addContainer(container, "", 0, 5, 10);
            }
            db.nameObj(this, "Non-GC Container Pool");
        }
    }

    // Field access methods
    private ooContObj getContainer(int i) {
        fetch();
        return this.pool[i];
    }

    private int getContainerCount() {
        fetch();
        return this.containerCount;
    }

    // Utility method to cluster the specified object with a container
    // chosen at random from the containers in this container pool
    public void clusterObject(Object object) {

        // Randomly select a container from this pool
        int index = Math.abs((new Random()).nextInt()) %
```

```
                        this.getContainerCount();
            ooContObj container = this.getContainer(index);

            // Cluster the object with the selected container
            container.cluster(object);
        }
    } // End ContainerPool class
```

ContainerPoolStrategy.java

```
//////////////////////////////////////////////////////////////////////
//
//  ContainerPoolStrategy - A clustering strategy that clusters
//  * A container in the database of requesting object
//  * A named root in a garbage-collectible container
//    selected from the container pool of the database of the
//    requesting object
//  * A scope-named object in a non-garbage-collectible
//    container selected from the container pool of the database
//    of the requesting object
//  * Any other basic object with the requesting object
//
// Copyright (c) 1998 Objectivity, Inc.  All Rights Reserved
//
// UNPUBLISHED PROPRIETARY SOURCE CODE OF OBJECTIVITY, INC.
//
// The copyright notice above does not evidence any actual
// or intended publication of such source code.
//
//////////////////////////////////////////////////////////////////////

import com.objy.db.*;     // Import Objectivity for Java exceptions
import com.objy.db.app.*; // Import Objectivity for Java classes

// If you make direct calls to requestCluster,
// you must call this method when the session that owns
// requestObject is in a transaction.
public class ContainerPoolStrategy implements ClusterStrategy {

    public void requestCluster(Object requestObject,
                               ClusterReason reason,
                               Object object)
    {
        ooDBObj db;
        String poolName;

        // Set db to the database of the requesting object
        if (requestObject instanceof ooDBObj)
            db = (ooDBObj)requestObject ;
        else if (requestObject instanceof ooContObj)
            db = ((ooContObj)requestObject).getDB() ;
        else if (requestObject instanceof ooObj)
            db = (((ooObj)requestObject).getContainer()).getDB() ;
        else
           throw new ClusteringException("Illegal requesting object.");
```

```
            if (object instanceof ooContObj) {
              // Cluster a container in the database of the requesting object
                db.cluster(object);
                return;
            }

            int reasonCode = reason.getReason();
            if ((reasonCode == ClusterReason.RELATIONSHIP) ||
                (reasonCode == ClusterReason.COLLECTION) ||
                (reasonCode == ClusterReason.REFERENCE) ||
                (reasonCode == ClusterReason.APPLICATION)) {
                // Cluster with requesting object to keep entire
                // object graph in same kind of container
                ((ooObj)requestObject).cluster(object);
                return;
            }

            // Get name of the appropriate container pool
            if (reasonCode == ClusterReason.BIND) {
                // Cluster named root in a garbage-collectible container
                poolName = "GC Container Pool";
            }
            else if (reasonCode == ClusterReason.NAMEOBJ) {
                // Cluster scope-named object in a
                // non-garbage-collectible container
                poolName = "Non-GC Container Pool";
            }
            else
               throw new ClusteringException("Illegal clustering reason.");

            // Retrieve the container pool
       ContainerPool containerPool = (ContainerPool)(db.lookupObj(poolName));

            // Cluster object in a container chosen at random
            // from the selected pool
            containerPool.clusterObject(object);
        }
    } // End ContainerPoolStrategy class
```

Cluster-By-Class Strategy

`ClusterByClassStrategy` extends `ContainerPoolStrategy` to accept an application-specific clustering reason of the `JustCreatedReason` class. Such a clustering reason indicates that an object is being made persistent because it is a newly created object of a class whose objects are always persistent. The strategy decides where to cluster such an object based on its class:

- Cluster a newly created object of the `Account` class (see page 460) in a garbage-collectible container selected from the container pool of the database named `"Accounting"`.
- Cluster a newly created object of the `Employee` class (see page 462) in a non-garbage-collectible container selected from the container pool of the database named `"Staff"`.

Objects of these two classes are made persistent as soon as they are created.

- Every `Account` object has an associated branch office. When an account is created, it is made persistent with a direct call to the `requestCluster` method of the session that owns its associated branch office; the branch office is the object requesting clustering. Note that the session of the branch office might use the default clustering strategy, the container-pool strategy, or the cluster-by-class strategy.
- Every Employee object also has an associated branch office. When an employee is created, it is made persistent with a direct call to the `requestCluster` method of a newly created cluster-by-class strategy; the branch office is the object requesting clustering.

ClusterByClassStrategy.java

```
//////////////////////////////////////////////////////////////////////
//
// ClusterByClassStrategy - A clustering strategy that clusters
// * A container in the database of requesting object
// * A named root in a garbage-collectible container
//   selected from the container pool of the database of the
//   requesting object
// * A scope-named object in a non-garbage-collectible
//   container selected from the container pool of the database
//   of the requesting object
// * A newly created object of the Account class in a
//   garbage-collectible container selected from the container pool
//   of the Accounting database
// * A newly created object of the Employee class in a
//   non-garbage-collectible container selected from the container pool
//   of the Staff database
```

```
// * Any other basic object with the requesting object
//
// A cluster reason of the JustCreatedReason class indicates that the
// object is being made persistent because it is a newly created object
// of a class whose objects are always persistent.
//
// Copyright (c) 1998 Objectivity, Inc.  All Rights Reserved
//
// UNPUBLISHED PROPRIETARY SOURCE CODE OF OBJECTIVITY, INC.
//
// The copyright notice above does not evidence any actual
// or intended publication of such source code.
//
////////////////////////////////////////////////////////////////////////

import com.objy.db.*;     // Import Objectivity for Java exceptions
import com.objy.db.app.*; // Import Objectivity for Java classes

// If you make direct calls to requestCluster, you must call this method
// when the session that owns requestObject is in a transaction.
public class ClusterByClassStrategy implements ClusterStrategy {

    public void requestCluster(Object requestObject,
                               ClusterReason reason,
                               Object object) {
        ooDBObj requestingDB;
        ooDBObj clusteringDB;

        // Set requestingDB to the database of the requesting object
        if (requestObject instanceof ooDBObj)
            requestingDB = (ooDBObj)requestObject;
        else if (requestObject instanceof ooContObj)
            requestingDB = ((ooContObj)requestObject).getDB();
        else if (requestObject instanceof ooObj)
            requestingDB =
                    (((ooObj)requestObject).getContainer()).getDB();
        else
           throw new ClusteringException("Illegal requesting object.");

        if (object instanceof ooContObj) {
          // Cluster a container in the database of the requesting object
            requestingDB.cluster(object);
            return;
        }

        int reasonCode = reason.getReason();
        if ((reasonCode == ClusterReason.APPLICATION) &&
```

```
            (reason instanceof JustCreatedReason)) {
           // Make object persistent because all objects of its class are
            // made persistent when they are created
            if (object instanceof Account) {
                // Cluster in a container from the pool of
             // garbage-collectible containers in the Accounting database
                clusteringDB = getAccountingDB(requestingDB);
               clusterInPool(object, clusteringDB, "GC Container Pool");
                return;
            }
            else if (object instanceof Employee) {
                // Cluster in a container from the pool of
              // non-garbage-collectible containers in the Staff database
                clusteringDB = getStaffDB(requestingDB);
                clusterInPool(object, clusteringDB,
                                    "Non-GC Container Pool");
                return;
            }
            else
                throw new ClusteringException("Unrecognized class.");
        }

        if (reasonCode == ClusterReason.BIND) {
            // Cluster named root in a garbage-collectible container
            clusterInPool(object, requestingDB, "GC Container Pool");
        }
        else if (reasonCode == ClusterReason.NAMEOBJ) {
            // Cluster scope-named object in a
            // non-garbage-collectible container
           clusterInPool(object, requestingDB, "Non-GC Container Pool");
        }
        else {
            // Cluster with requesting object
            ((ooObj)requestObject).cluster(object);
            return;
        }
    }

    // Static utility to get a database with the specified name
    // If the database doesn't already exits, create it and initialize
    // its container pools.
    protected static ooDBObj getDBWithPools(ooDBObj requestingDB,
                                            String name,
                                            int numberGCcontainers,
                                           int numberNonGCcontainers) {
        ooDBObj clusteringDB;
        ContainerPool newPool;
```

```
        // This method must be called when the session that
        // owns requestingDB is in a transaction.
        // Use requestingDB to be sure that the retrieved database
        // belongs to the correct session
        ooFDObj fd = requestingDB.getFD();
        if (fd.hasDB(name)) {
            clusteringDB = fd.lookupDB(name);
        }
        else {
            // Create database and initialize its container pools
            clusteringDB = fd.newDB(name);
      newPool = new ContainerPool(clusteringDB, numberGCcontainers, true);
        newPool = new ContainerPool(clusteringDB, numberNonGCcontainers,
false);
        }
        return clusteringDB;
    } // End getClusteringDB method

   // Static utility to cluster an object in the specified container pool
    // of the specified database
    protected static void clusterInPool(Object object, ooDBObj db,
                                        String poolName) {
        // This method must be called during a transaction

        // Retrieve the container pool
   ContainerPool containerPool = (ContainerPool)(db.lookupObj(poolName));

     // Cluster object in a container chosen at random from the selected pool
        containerPool.clusterObject(object);
    }

    // Static utility to get a local representation of the Accounting
   // database that belongs to the same session as the requesting database
    protected static  ooDBObj getAccountingDB(ooDBObj requestingDB) {
        // This method must be called during a transaction
       // Returns database with 50 GC containers and 1 non-GC container
        return getDBWithPools(requestingDB, "Accounting", 50, 1);
    }

   // Static utility to get a local representation of the Staff database
    // that belongs to the same session as the requesting database
    protected static ooDBObj getStaffDB(ooDBObj requestingDB) {
         // This method must be called during a transaction
       // Returns database with 10 GC containers and 20 non-GC containers
         return getDBWithPools(requestingDB, "Staff", 10, 20);
    }
```

```
} // End ClusterByClassStrategy class
```

JustCreatedReason.java

```
////////////////////////////////////////////////////////////////////////
//
// JustCreatedReason - A clustering reason indicating that an
// object is being made persistent because it is a newly created object
// of a class whose objects are always persistent.
//
// Copyright (c) 1998 Objectivity, Inc.  All Rights Reserved
//
// UNPUBLISHED PROPRIETARY SOURCE CODE OF OBJECTIVITY, INC.
//
// The copyright notice above does not evidence any actual
// or intended publication of such source code.
//
////////////////////////////////////////////////////////////////////////

import com.objy.db.*;     // Import Objectivity for Java exceptions
import com.objy.db.app.*; // Import Objectivity for Java classes

public class JustCreatedReason implements ClusterReason {
    public int getReason() {
        return ClusterReason.APPLICATION;
    }
} // End JustCreatedReason class
```

Account.java

```
////////////////////////////////////////////////////////////////////////
//
// Account - a financial account
//
// Copyright (c) 1998 Objectivity, Inc.  All Rights Reserved
//
// UNPUBLISHED PROPRIETARY SOURCE CODE OF OBJECTIVITY, INC.
//
// The copyright notice above does not evidence any actual
// or intended publication of such source code.
//
////////////////////////////////////////////////////////////////////////

import com.objy.db.*;     // Import Objectivity for Java exceptions
import com.objy.db.app.*; // Import Objectivity for Java classes

public class Account extends ooObj {
    // Clustering reason for clustering objects of this class
```

```
    private static ClusterReason reason = new JustCreatedReason();

    // Persistent fields
    protected BranchOffice branch; // Branch office managing the account

    // Constructor makes an object persistent; the branch office requests
    // it to be clustered.
    public Account(BranchOffice branch) {
       // This constructor must be used during a transaction

       this.branch = branch;

       // Cluster the new object using the clustering
       // strategy of the session that owns branch
       Session session = branch.getSession();
       session.requestCluster(branch, reason, this);
    }
} // End Account class
```

Employee.java

```
//////////////////////////////////////////////////////////////////////
//
// Employee - an employee
//
// Copyright (c) 1998 Objectivity, Inc.  All Rights Reserved
//
// UNPUBLISHED PROPRIETARY SOURCE CODE OF OBJECTIVITY, INC.
//
// The copyright notice above does not evidence any actual
// or intended publication of such source code.
//
//////////////////////////////////////////////////////////////////////

import com.objy.db.app.*;     // Import Objectivity for Java classes

public class Employee extends ooObj {
    // Clustering reason for clustering objects of this class
    private static ClusterReason reason = new JustCreatedReason();

    // Persistent fields
    // Branch office where the employee works
    protected BranchOffice branch;
    protected String name;

    // Constructor makes an object persistent;
    // the branch office requests it to be clustered.
    public Employee(BranchOffice branch, String name) {
       // This constructor must be used during a transaction

        this.branch = branch;
        this.name = name;

        // Cluster the new object using the
        // ClusterByClassStrategy clustering strategy
        ClusterByClassStrategy strategy = new ClusterByClassStrategy();
        strategy.requestCluster(branch, reason, this);
    }
} // End Employee class
```

BranchOffice.java

```
//////////////////////////////////////////////////////////////////////
//
// BranchOffice - a corporate branch office
//
// Copyright (c) 1998 Objectivity, Inc.  All Rights Reserved
//
// UNPUBLISHED PROPRIETARY SOURCE CODE OF OBJECTIVITY, INC.
//
// The copyright notice above does not evidence any actual
// or intended publication of such source code.
//
//////////////////////////////////////////////////////////////////////

import com.objy.db.*;     // Import Objectivity for Java exceptions
import com.objy.db.app.*; // Import Objectivity for Java classes

public class BranchOffice extends ooObj {
} // End BranchOffice class
```

Glossary

ACID. Acronym for the properties—atomicity, consistency, isolation, and durability—maintained when the operations within a transaction are applied to a database.

autonomous partition (FTO). A partitioning of data within a federated database that can perform most database functions independently of any other autonomous partition, even if the others are completely unavailable.

basic object. An instance of a class that is derived from `ooObj` or implements the `Persistent` interface. A basic object is the fundamental unit of storage in an Objectivity/DB database.

bidirectional relationship. A *relationship* in which two related objects have references to each other.

boot file. A file that contains information used by an application or tool to locate and open a federated database.

cache. See *Objectivity/DB cache* and *session cache*.

cardinality (of a relationship). A property of a *relationship* that specifies whether an object on one side of the relationship can be related to multiple objects on the other side. The cardinality of a relationship can be any one of the following: one to one, one to many, many to one, or many to many.

clustering. The process of assigning an object to a storage location when the object is made persistent. Clustering a basic object assigns it a location in a particular container; clustering a container assigns it a location in a particular database.

clustering strategy. An object of a class that implements the `ClusterStrategy` interface. A clustering strategy determines where to cluster objects when they are made persistent.

connection. An object (of class `Connection`) that represents a connection between an application and a federated database.

container. A grouping of basic objects in a database. Containers are the fundamental units of locking; when any basic object in a container is locked, the entire container is locked, effectively locking all basic objects in the container.

database. The second level in the Objectivity/DB storage hierarchy. A database contains one or more containers, which in turn contain fundamental units of persistent data, called basic objects. A database is physically maintained in a file.

database image (DRO). A copy of a database contained in an autonomous partition.

dead object. An object that is no longer valid for Objectivity for Java operations. When you terminate a session, all objects that the session owns become dead objects. Objects also become dead when they are deleted within a session.

deep copy. An object created by setting the value of each field to a copy of the value in the corresponding field of another object.

federated database. The highest level in the Objectivity / DB storage model. A federated database consists of a system database and one or more application-defined databases. Each federated database maintains a global schema containing all class descriptions. See *storage hierarchy*.

garbage-collectible container. A container that adheres to the garbage-collection paradigm: If an object in the container is not a named root and cannot be reached by references and / or relationships from a named root, that object is considered to be garbage.

growth factor (of a name map). The percentage by which the name map's hash table grows when it is resized. Each time the hash table is resized, the number of bins is increased by the growth factor, then rounded up to the nearest prime number.

handler method (for a persistent event). A method of a persistence-capable class that is called when a persistent event occurs to an object of that class. The handler method performs whatever application-specific processing is required to respond to the event.

index. A data structure that maintains references to objects that have been sorted according to the values in one or more fields of the object. The sorting order is determined by the ordering of the fields specified in the keys of the index.

journal file. A file that contains a log of changes made during a transaction. It is used to restore the database if a transaction is aborted or its session is terminated abnormally. Journal files are removed after the normal completion of a transaction.

maximum average density (of a name map). The average number of elements per bin allowed before the name map's hash table must be resized.

MROW. Multiple Readers, One Writer. A concurrency mechanism that allows a container to have multiple readers and one writer simultaneously.

name map. An unordered persistent collection of key-value pairs in which the key is a string and the value is a persistent object.

name scope. A set of names defined by a particular object to identify basic objects and containers; each name in the set is called a *scope name*.

named root. A persistent object that can be located by a root name, which is unique within the federated database or a particular database.

non-garbage-collectible container. A container in which all objects are assumed to be valid.

non-persistence-capable class. A class whose instances cannot be saved in a database.

object conversion. The process of converting persistent objects in a federated database to make their persistent data consistent with a new version of their class description in the schema.

object graph. A directed graph data structure consisting of objects linked to other related or associated objects. The links in a graph of persistent objects can be references in persistent fields, relationships, or memberships in persistent collections.

Objectivity/DB cache. A part of virtual memory that is allocated and managed by Objectivity / DB to allow high speed access to persistent objects. When a persistent object is retrieved from physical storage, Objectivity / DB places the page in which the object resides into the cache.

Objectivity/DB object. Any *persistent object* or *storage object*.

OID. Object identifier; a number that uniquely identifies a basic object within a federated database. An object's OID identifies its database, container within the database, logical page within the container, and logical slot on the page.

page. The minimum unit of transfer to and from disk and across networks. Objects reside in pages. The Objectivity/DB page size can be chosen by the database developer. These pages are not the same as operating system pages.

persistence-capable class. A class whose instances can be made persistent and saved in a database. All persistence-capable classes are derived from `ooObj` or implement `Persistent`.

persistent collection. An aggregate persistent object that can contain a variable number of elements; each element is either a persistent object, or a key-value pair whose value is a persistent object.

persistent data. The values in the persistent fields of a persistent object; these values are saved persistently in the federated database.

persistent event. A pre- or post-processing event; when a persistent object is involved in certain persistent operations, the object receives a persistent-event notification immediately before or after the persistent operation occurs. In response to the notification, the object can perform whatever application-specific processing is required.

persistent field. An application-defined field of a persistent object whose value is saved persistently if the object is written to a database. All nonstatic, nontransient, application-defined fields of persistence-capable classes are persistent by default.

persistent object. An object that has been assigned a storage location in the database where it will be stored. When you commit the transaction in which you create a persistent object, the object is saved in the database. A persistent object continues to exist and retains its data beyond the duration of the process that created it.

persistor. An object that provides persistence behavior on behalf of a persistent object.

property (of a persistent object). Information about the object that is considered part of its internal representation; for example, information stored in fields defined by Objectivity for Java. Properties of a persistent object are not considered part of the object's *persistent data*.

referential integrity. A characteristic of an object that ensures that the object has references only to objects that actually exist. Maintaining referential integrity requires that, when any object is deleted, all references from other objects to the deleted object are removed.

relationship. A reference or link between one object and one or more other objects. Relationships may be one-to-one, one-to-many, many-to-one, or many-to-many. They may connect objects of the same class or different classes. In Objectivity/DB, relationships are persistent objects that are stored in the database.

restricted thread policy. A session's thread-management policy that requires a Java thread to be joined to the session before the thread can interact with the session (and its associated transaction).

root dictionary. A name map used by a database or a federated database to pair root names with the corresponding objects. The objects identified in the root dictionary are the *named root* of the database or federated database.

root name. A name that uniquely identifies a persistent object to a particular database or federated database. An object can have more than one root name within the same database, and can have a root name in more than one database. Objects that have root names are a *named root* of the database or federated database.

roots container. A special container of a database where the database stores its root dictionary; any object that becomes persistent when it is made a *named root* is also stored in this container.

scalable collection. A collection that can increase in size without performance degradation.

schema. A language-independent data model that describes the classes of all persistent objects maintained in a federated database.

schema class name. The name used in the schema to identify a persistence-capable class. Each application also has a schema class name for each persistence-capable class. When the application reads or writes an object, Objectivity for Java maps between the Java class name and the application's schema class name.

schema evolution. The process of modifying the schema of a federated database so that its class descriptions are consistent with new versions of the corresponding Java class.

scope name. The name that identifies an object within the name scope defined by a particular object.

scope object. An object that defines a name scope; each scope name in the name scope uniquely identifies an object to the scope object (but not to other objects).

session. An object (of class `Session`) that represents an extended interaction between an application and the connected federated database. An application can have multiple interactions, each corresponding to a particular subtask that the application performs.

session cache. A part of virtual memory in which a particular session keeps its persistent objects, organized by their object identifiers (OIDs).

shallow copy. An object created by setting the value of each field to the value in the corresponding field of another object.

storage hierarchy. The four-level hierarchy of containment relationships between objects in a federated database. Each non-leaf object in the hierarchy is a storage object; each leaf object is a basic object. The federated database is the root of the hierarchy; its databases form the second level of the hierarchy. Below each database are the containers stored in that database; below each container are the basic objects stored in that container

storage object. Any object representing a level of the Objectivity / DB storage hierarchy. The storage objects are federated databases, databases, and containers.

system name. A name, similar to a file name, that uniquely identifies a federated database, autonomous partition, database, or container to Objectivity / DB.

thread policy. The policy by which Java threads are allowed to interact with sessions. See *restricted thread policy* and *unrestricted thread policy*.

transaction. A unit of work an application applies to a federated database. Transaction control is used to make several database requests or operations appear to all users as a single, indivisible operation.

transient field. A field whose value is not saved persistently if the object is written to a database. Transient fields are specified with the transient modifier in the class declaration.

transient object. An object that exists only within the application that created it.

unidirectional relationship. A *relationship* in which one object has a reference to the related object, but the related object has no reference back to the relating object.

unique index. An *index* in which each indexed object has a unique combination of values in its key fields.

unrestricted thread policy. A session's thread-management policy that allows any Java thread to interact with the session (and its associated transaction).

Index

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

OBJECTIVITY, INC.
301B East Evelyn Avenue
Mountain View, California 94041
USA
+1 650-254-7100
+1 650-254-7171 Fax
www.objectivity.com
info@objectivity.com