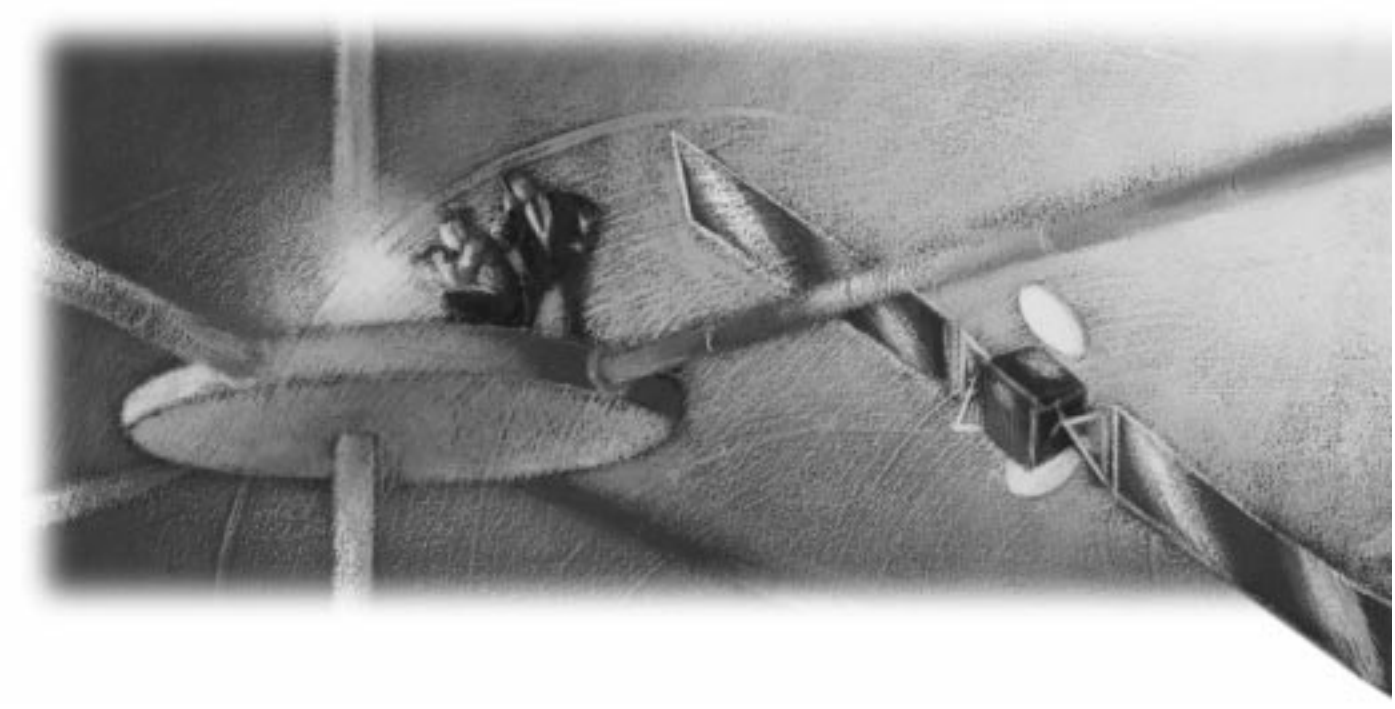


Objectivity/C++ Programmer’s Guide

Release 6.0

Objectivity/C++ Programmer's Guide

Part Number: 60-CPPGD-0

Release 6.0, October 2, 2000

Contents

Part 4 FINDING PERSISTENT OBJECTS

Part 5 SPECIAL TOPICS

About This Book

This book, *Objectivity/C++ Programmer's Guide*, describes how to develop Objectivity / C++ applications. Objectivity / C++ enables a C++ application to create, store, and manipulate persistent data in an Objectivity / DB federated database.

You should use this book in conjunction with the Objectivity / C++ programmer's reference, which a detailed description of each public construct in the Objectivity / C++ programming interface. If you are also defining a federated-database schema, you should read the Objectivity / C++ Data Definition Language book.

Audience

This book assumes that you are familiar with programming in C++.

Organization

- Part 1 introduces the concepts and processes that are fundamental to the Objectivity / DB object-oriented database management system and to the Objectivity / C++ programming interface to Objectivity / DB.
- Part 2 describes the classes and mechanisms that control Objectivity / C++ application processes.
- Part 3 describes the classes and mechanisms that Objectivity / C++ provides for modeling persistent data, and the auxiliary classes through which an Objectivity / C++ application works with persistent objects.
- Part 4 explains how to organize persistent objects to minimize search and how to use the organization to find the persistent objects when they are needed.

- Part 5 covers special topics that are of interest to some, but not all, users of the Objectivity/C++ programming interface.

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 INTRODUCTION

This part introduces the concepts and processes that are fundamental to the Objectivity / DB object-oriented database management system and to the Objectivity / C++ programming interface to Objectivity / DB.

1

Objectivity/DB Basics

This chapter provides an introduction to the Objectivity / DB object-oriented database management system. It introduces:

- Objectivity / DB architecture
- Steps involved in developing an Objectivity / DB application

Objectivity/DB Architecture

Objectivity / DB provides database-management services for storing and finding objects created by applications written in any of the following object-oriented programming languages: C++, Java, and Smalltalk. Each such application is written using an Objectivity / DB programming interface (Objectivity / C++, Objectivity for Java, or Objectivity / Smalltalk); objects stored by an application written in one language can be found by applications written in other languages.

This section gives an overview of the components of an Objectivity / DB application, independent of the programming interface in which it is written:

- The processes involved in an Objectivity / DB application.
- Transactions, the mechanism for organizing operations on Objectivity / DB objects.
- The kinds of Objectivity / DB objects accessible from within an application and the purpose of each.
- The Objectivity / DB object model.
- 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, applications have database services built directly into them instead of relying on a back-end server process. This integration of database services is generally accomplished by dynamically

loading Objectivity / DB libraries into the same process space as the Objectivity / DB application.

Objectivity / DB provides simultaneous, multiuser access to databases that can be distributed across a network. A group of such databases using a common object model (or *schema*) is organized into a unit, called a *federated database* or *federation*. 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 memory representations of objects, which must be obtained 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 *update 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 29, “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 particular federated database.

NOTE An Objectivity / DB process can interact with only one federated database.

A transaction is effectively a subsection of an application, the extent of which is designated by four operations: begin, checkpoint, commit, 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. An application can *checkpoint* a transaction, which saves modifications to the federated database without ending the transaction. A transaction ends when it is committed or aborted. At that time, the locks on any Objectivity / DB objects are released, the application’s memory representations of Objectivity / DB objects may no longer be consistent with the

objects in the federated database, and further system-defined operations on an Objectivity / DB object will signal an error.

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.

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 federated database or none is performed. Thus, several operations, on one or more of the objects contained in the federated database, appear to all users as a single, indivisible operation.
- *Consistency* means that the transaction takes the federated 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 that 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 federated database and henceforth visible to any other concurrent users of the federated database. If the transaction aborts, none of the changes are made in the federated database.
- *Durability* means that the effects of committed transactions are preserved in the event of system failures such as crashes or memory exhaustion.

For additional information, see Chapter 4, “Transactions”.

Objectivity/DB Objects

There are four types of standard 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 an application is normally represented as a basic object. Each basic object is stored 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 stored within a database.

A *database* consists of system-created containers and containers created by applications. 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 applications. Each federated database has a *system database*, which contains the schema (object model) with descriptions of the classes whose objects are stored in the federated database. The schema is language independent, which means that objects of classes defined using the Objectivity / DB programming interface for one language can be accessed and managed from applications that use the Objectivity / DB programming interfaces for other languages.

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.

Storage Objects and Persistent Objects

Objectivity / DB objects are organized into a containment or *storage hierarchy*. 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. Figure 1-1 illustrates the kind of object at each level of the storage hierarchy.

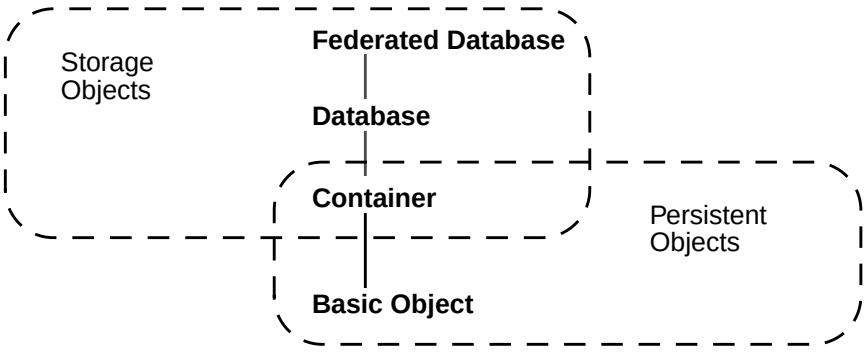


Figure 1-1 Objectivity/DB Objects

The Objectivity / DB objects that *contain* other objects (federated database, database, and container) 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. Within an application, the memory representation for a storage object is essentially a proxy for forwarding messages to Objectivity / DB, which locates and manipulates the actual storage object on disk.

The Objectivity / DB objects that can contain application-specific data are called *persistent objects*. Basic objects are persistent objects, as are containers (which are also storage objects). Objectivity / DB defines the persistence behavior for persistent objects—namely, the ability to store the objects' data persistently, to link related persistent objects together, to look up an individual persistent object, and to follow links from a persistent object to find its related persistent objects.

Any application-specific behavior is defined by the object's class, which also defines the object's application-specific persistent data. Within an application, the memory representation for a persistent object contains its persistent data.

Storage and persistent objects continue to exist after an application terminates. They can be shared among applications, with locking managed by Objectivity / DB. Except for a federated database, all types of objects can be created and deleted dynamically by an application program.

For additional information, see Chapter 8, "Storage Objects," and Chapter 9, "Persistent Objects".

Autonomous Partitions

A separately purchased option to Objectivity / DB, namely Objectivity / DB Fault Tolerant Option (Objectivity / FTO), adds another kind of Objectivity / DB object, called an *autonomous partition*. 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. The remainder of this chapter addresses only storage objects and persistent objects. Chapter 27, "Autonomous Partitions," describes autonomous partitions.

Identifiers

Every Objectivity / DB object has an *identifier* that uniquely distinguishes it among other objects of the same type within the same containing storage object.

All storage objects and autonomous partitions are given identifiers when they are created. The federated database identifier is an integer, specified by the creator of the federated database. Databases, containers, and autonomous partitions also have integer identifiers. These identifiers are used to identify their respective Objectivity / DB objects to the lock server.

The identifier for a basic object is called its *object identifier* or *OID*. An object identifier corresponds to the location of the object within the federated database;

it is unique within the entire federated database, not just within the object's container. An object's OID can change during the lifetime of the object only if the object is moved to a different container. After an object is moved or deleted, its previous object identifier may be reused for a new object.

The identifier of a database, container, or autonomous partition can be expressed in the object-identifier format, so object identifiers provide a general mechanism for uniquely identifying any object in a federated database. Objectivity / DB uses object identifiers instead of memory addresses to identify objects, because object identifiers 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.

Working With Objectivity/DB Objects

You create a federated database with an administrative tool. When an application accesses a federated database, it obtains a memory representation through which it performs operations on the federated database.

Basic lifecycle operations on the other kinds of Objectivity / DB objects provide for creation, deletion, and property management. Additional operations support the transfer of objects between an application and a federated database. Thus, before you can access an Objectivity / DB object in an application, you must:

1. *Find* the object in the federated database—for example, by looking up an application-assigned name or by traversing to it from an associated object. Finding an object obtains the object identifier that uniquely identifies it within the federated database. This operation is also called *getting a reference to* an object.
2. Make the found object available to the application:
   - Obtain a memory representation of the object, including any application-specific data.
   - Obtain an appropriate lock on the object. A read lock indicates to Objectivity / DB that you need read-only access to an object. An update lock indicates that you intend to modify the object.

   Until the lock on the object is released, all subsequent operations on the object are directed to the memory representation, that is, these operations do not require the application to find the object in the federated database.

The operations for finding and making an object available are realized differently within each programming interface to Objectivity / DB:

- In Objectivity / C++, one operation finds an object (gets a reference to it), and another operation *opens* it (represents the entire object in memory and locks it). In many cases, a single function performs both operations. The memory representation of the object exists within a special *cache* managed by Objectivity / DB.
- In Objectivity for Java and Objectivity / Smalltalk, one operation *retrieves* an object (finds it and obtains a basic memory representation for it as a Java or Smalltalk object). A separate operation *fetches* any application-specific persistent data into the memory representation. Locking is usually performed implicitly by fetch or various other operations on a retrieved object.

Locking Objectivity/DB Objects

Objectivity / DB *locks* various objects to maintain consistency during simultaneous access by multiple transactions in different processes or threads. When a transaction has a read lock on an object, the application can safely read the object; when a transaction has an update lock, the application can safely modify the locked object.

A persistent object must be locked before it can be accessed. Locking is performed in a variety of ways, depending on the programming interface to Objectivity / DB. In most cases, a persistent object is locked *implicitly* by operations that obtain its application-specific data (for example, open or fetch); in such cases, the lock is acquired only at the point at which it is needed. Alternatively, you can reserve access to an object in advance by locking it *explicitly*. Explicit locking can reduce concurrency (because objects tend to be locked for longer periods of time), but ensures access to objects when you need it.

Containers are the actual units of locking for persistent objects. When a basic object is locked, the container in which it resides is locked, effectively locking all basic objects in the container. This is a performance advantage for a transaction that needs to access multiple objects in the same container; such a transaction can obtain the necessary permissions through a single lock request.

In general, databases and federated databases are shared resources, so two containers in the same database may be locked by two different, concurrent transactions. Furthermore, two or more concurrent transactions can lock the same container, provided that the locks are compatible. Read locks are always compatible, so two transactions can obtain read locks on the same container; update locks are always incompatible, so two transactions cannot lock the same container for update. In other cases, Objectivity / DB applies a *concurrent access policy* to determine whether a requested lock is compatible with existing locks held by other transactions; see “Concurrent Access Policies” on page 34.

If an application needs to “freeze” all of the containers in a database or in a federated database, it can explicitly lock the database or federated database. When a federated database or a database is locked explicitly, its contents cannot be modified by any other transaction. Furthermore, a federated database can be locked so that no other transaction can either read or modify it. Locking at this level is normally necessary only for administrative tasks that require exclusive access to the data for a period of time.

For additional information about locking, see Chapter 6, “Locking and Concurrency”.

Concurrent Access Policies

Objectivity / DB allows multiple transactions to read a container simultaneously and prevents multiple transactions from updating a container simultaneously. It supports two concurrent access policies to control whether one transaction can update a container while one or more transactions are reading the same container. The two concurrent access policies are *standard* and *multiple readers, one writer (MROW)*. The standard access policy prevents a transaction from reading a container that is being modified by another transaction; the MROW policy permits a transaction to read the last-committed or checkpointed version of a container that is being modified. The 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.

Persistence-Capable Classes

Every persistent object is an instance of some *persistence-capable class*. A persistence-capable class has persistence behavior that enables applications to store instances of the class persistently in an Objectivity / DB federated database. Instances of persistence-capable classes are normally *persistent*, that is, stored in the federated database. Many persistence capable classes also support instantiation to create transient objects. A *transient* object exists only within the memory of the process that creates it.

Non-persistence-capable classes do not have persistence behavior; instances of those classes cannot be stored as independent persistent objects. However, an instance of a non-persistence-capable class may be stored in the federated database indirectly if it is embedded in the data of a persistent object. An application cannot find such an embedded object independently of its containing persistent object.

Each Objectivity / DB programming interface allows applications to define their own persistence-capable classes; the interfaces also provide persistence-capable collection classes.

Application-Defined Classes

Applications define persistence-capable classes for the basic objects they want to store in a federated database. Each persistence-capable class is a class in the application's programming language (C++, Java, or Smalltalk); applications can create both transient and persistent instances of the class.

An application that needs to associate data with containers can define persistence-capable container classes as well.

Before instances of a persistence-capable class can be added to a federated database, a language-independent description of the class must be added to the federated database schema. The mechanism for adding class descriptions to the schema depends on the programming interface to Objectivity / DB.

NOTE Objectivity / DB does not save the "behavior" of persistence-capable classes—that is, the member functions of a C++ class or the methods of a Java or Smalltalk class.

For additional information, see "Persistence-Capable Classes" on page 140.

Persistent-Collection Classes

Objectivity / DB defines persistence-capable collection classes. An instance of one of these classes is called a *persistent collection*. A persistent collection is an aggregate object that groups other persistent objects together. Objectivity / DB supports persistent collections of the following kinds:

- *Sets* and *lists* have persistent objects as elements.
- *Object maps* have key-value pairs as elements; each key and each value is a persistent object.
- *Name map*s have key-value pairs as elements; each key is a string (or name) and each value is a persistent object.

All instances of a persistent collection class are persistent.

For additional information, see Chapter 11, "Persistent Collections".

Objectivity/DB Object Model

In the Objectivity / DB object model, a persistence-capable class has *attributes* and *associations*. The attributes of a class constitute its component data. The associations of a class describe how an instance of that class can be related to instances of other specified classes.

Attributes

A persistent object has a value for each attribute defined by its class. The attributes' values express the state of the object. Attributes correspond to standard data members of a C++ class, fields of a Java class, or instance variables of a Smalltalk class.

The Objectivity / DB object model supports attributes of the following data types:

- Primitive types (integers, floating-point numbers, characters, and so on).
- Reference types in which the referenced class is persistence capable.
- Embedded-class types in which the embedded class (or structure) is non-persistence-capable.

A given attribute can hold either a single value of its type, or a fixed-size array of values of its type. The value of a primitive attribute is a number or character, The value of a reference attribute is a reference to a persistent object, which acts as a link to the referenced object. The value of an embedded-class type is an instance of the embedded class.

The different programming interfaces support this model to varying degrees. Objectivity / C++ supports the complete Objectivity / DB object model. Because of restrictions in the Java and Smalltalk languages, however, Objectivity for Java and Objectivity / Smalltalk do not support embedded-class attributes, attributes containing fixed-size arrays of values, or attributes with template types.

For additional information, see "Attributes" on page 141.

Associations

The persistence-capable class that defines an association is called the *source class* for that association. The association indicates how an instance of the source class can be related to one or more instances of a *destination class*. The destination class can be any persistence-capable class, including the source class itself.

At runtime, associations can be formed, each one linking a particular instance of the source class, called the *source object*, to an instance of the destination class, called the *destination object*. A *to-one* association can link a given source object to a single destination object; a *to-many* association can link a a given source object to multiple destination objects.

Associations link persistent objects together just as reference attributes do. Associations, however, enable a higher level of functionality. For example, when you define an association, you can specify:

- Whether inverse links should be maintained automatically.
- How associations are handled when an application creates a copy or new version of the source object.
- Whether deletion and locking operations on a source object should be propagated to the destination object(s).

For additional information, see “Associations” on page 145.

Operations on Objectivity/DB Objects

You can perform the following operations on an Objectivity / DB federated database:

- Create the hierarchy of storage objects and add persistent objects.
- Find existing storage objects and persistent objects.
- Modify persistent objects.
- Delete existing storage objects and persistent objects.

Creating Objects

You create storage and persistent objects following the levels in the Objectivity / DB storage hierarchy. Once you have created a federated database, you can create databases within it; once you have created a database, you can create containers within it; once you have created a container, you can create basic objects within it.

Objectivity / DB objects can be created by administrative tools or by an application using one of the programming interfaces. The following table summarizes the mechanisms for creating each kind of object.

Objectivity/DB Object	Create by Tool	Create by Application
Federated Database	Yes; `oonewfd`	No
Database	Yes; `oonewdb`	Yes
Container	No	Yes
Basic Object	No	Yes

The administrative tools are described in the Objectivity / DB administration book.

An application creates an Objectivity / DB object within a transaction; the new object is not visible to other processes until the application commits or checkpoints the transaction. If the transaction is instead aborted, the object is not added to the federated database.

Federated Database

You use the `oonewfd` administrative tool to create a federated database. Among the information you provide this tool is the path for the *boot file* that applications will use to access the federated database.

Database

You can create an application-specific database either from an application or from the command line with the administrative tool `oonewdb`. When you create a database, you specify a *system name* that uniquely identifies it within its federated database. Valid system names follow the rules for valid file names within the operating system.

One database is created automatically along with the federated database, namely the system database that will store the federated database schema and catalog. Applications do not access the system database directly.

Container

You create an application-specific container from an application, specifying how to *cluster* it. Clustering a container assigns it to a particular database. You may optionally specify a system name that uniquely identifies the container within its database. Giving a container a system name allows you to look it up by name.

One container is created automatically along with each database, namely the *default container*. When an application creates a basic object in a database without specifying a container for it, the object is stored in the default container.

Basic Object

You create a basic object from an application; if the object is to be persistent, you must specify where to cluster it. Clustering a basic object assigns it to a location within a particular container.

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. An important design activity is to establish a clustering strategy that meets your concurrency, space utilization, and runtime performance requirements (see “Assigning Basic Objects to Containers” on page 131).

Transient and Persistent Objects

The exact mechanism for creating persistent objects (basic objects and containers) differs among the programming interfaces to Objectivity / DB. In Objectivity / C++, a persistent object is created when its class is instantiated. In Objectivity for Java and Objectivity / Smalltalk, the class is instantiated as a transient object that is later made persistent.

Linking Objects Together

You can establish various kinds of interrelationships between the two persistent objects by creating a directional link from one object, called the *source object*, to the other, called the *destination object*. Once you have found a source object, you can follow its links to find the related destination object of each link.

The reference attributes and associations of any source object can link it to destination objects. In addition, a persistent collection is a source object that maintains links to the persistent objects that it contains.

For additional information, see Chapter 15, “Creating and Following Links”.

Preparing Objects for Individual Lookup

If you want applications to be able to find a particular persistent object (either a basic object or a container), you must explicitly provide a way to look up that object. Objectivity / DB supports several mechanisms for identifying a persistent object for individual lookup; you use different mechanisms depending on the programming interface. In all of the programming interfaces, you can:

- Give a container a system name when you create it. Applications will then be able to look up the container within its database by its system name.
- Give any persistent object a *scope name* that is unique within the context or *name scope* of a *scope object*. Once an object has a scope name, you can look it up by that name in the correct name scope.

  Any Objectivity / DB object (storage object, basic object, or autonomous partition) can be a scope object. You can name a persistent object in the scope of a storage object even if that storage object does not contain the persistent object. A persistent object can have a name in more than one name scope.
- Make any persistent object the value in a key-value pair in a name map or object map. You can look up the object by its corresponding key.

In the Objectivity for Java and Objectivity / Smalltalk programming interfaces, you can:

- Make any persistent object a *named root* in a particular database or in the federated database. The database or federated database in which the object is named is called its *naming object*. Each named root has a *root name* that is

unique within its naming object. You can look up the object by name in its naming object.

Finding Objects

You typically find a database by looking up its system name. If a container has a system name, you can find it by looking up its system name. You can also find databases and containers by iterating over the storage object that contains them. See “Finding a Database” on page 165 and “Finding a Container” on page 176.

You can find existing persistent objects in the federated database using several techniques. Typically, you first find an object of interest either by individual lookup (Chapter 16) or by iterating over the objects in a particular group (Chapter 17), for example, the basic objects in a particular container or the elements of a particular persistent collection. If a found object has links to other objects, you follow those links to find the destination objects (Chapter 15).

You can use the containment relationships between objects in the storage hierarchy to help you find objects. You can search down the hierarchy from a given storage object, looking for objects either at the next level or at all lower levels. A search for objects at all lower levels is called a *scan* operation. You can also search up the hierarchy for the containing object:

Content-based filtering (Chapter 18) allows you to modify a search operation to find only those persistent objects that meet a condition. This kind of filtering allows you to search for objects by the values of one or more of their attributes. You can perform content-based filtering by specifying a condition when you scan a storage object and when you search for destination objects linked by a to-many association. The condition is expressed as a *predicate string*—that is, a string in the Objectivity / DB predicate query language.

A *predicate scan* is a scan operation that searches a storage object for objects of a given class that meet a condition. Predicate scans are expensive operations when the number of objects being searched is very large. To optimize such searches, Objectivity / DB supports the definition of *indexes*, which order the persistent objects in a particular storage object according to the values in one or more of their attributes.

Modifying Persistent Objects

After finding a persistent object of an application-defined class, an application can work with its memory representation, setting its attributes and associations as appropriate. All such modification must occur within a transaction. Furthermore, the application must inform Objectivity / DB of its intention to modify the object and must obtain an update lock to protect the object from simultaneous and inconsistent modification by another process.

All changes to the object's attributes and associations are local until the application commits or checkpoints the transaction. At that time, the changes are written to the federated database and become visible to other processes. If the application instead aborts the transaction, the object remains unchanged in the federated database.

Each programming interface to Objectivity / DB has its own mechanism for indicating that it plans to modify an object. Typically, the same operation performs this notification and obtains an update lock.

Deleting Objects

Objectivity / DB objects can be deleted by administrative tools or by an application using one of the programming interfaces. The following table summarizes the mechanisms for deleting each kind of object.

Kind of Object	Delete by Tool	Delete by Application
Federated Database	Yes; `oodeletefd`	No
Database	Yes; `oodeletedb`	Yes
Container	No	Yes
Basic Object	No	Yes

The administrative tools are described in the Objectivity / DB administration book.

An application deletes Objectivity / DB objects within a transaction. A deleted database is removed from the federation immediately; aborting the transaction cannot undo the delete operation. In contrast, a deleted container or basic object is not actually removed from the federated database until the application commits or checkpoints the transaction; if the transaction is instead aborted, the object is not deleted.

When basic objects are linked together, the type of container in which they are stored determines whether you need to delete all unused basic objects explicitly:

- In a *non-garbage-collectible* container, basic objects that are no longer required must be explicitly tracked and deleted by an application. Such containers are used by applications written in a non-garbage-collected language, such as C++. These containers may also be used by Objectivity for Java and Objectivity / Smalltalk applications that need to interoperate with Objectivity / C++ applications.
- In a *garbage-collectible container*, an object automatically becomes available for deletion when no other object contains a link to it. Such containers are typically used by applications written in garbage-collected languages, such

as Java and Smalltalk. Just as memory can contain objects that are no longer referenced, a garbage-collectible container can include invalid objects that were left over after links to them were deleted. Garbage-collectible containers may also be used by Objectivity / C++ applications that need to interoperate with Objectivity for Java or Objectivity / Smalltalk applications.

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.

Developing an Objectivity/DB Application

Like any software development project, development of an Objectivity / DB application consists of designing, implementing, and deploying the application. Over the lifetime of the project, the various aspects of the application may need to be modified as requirements change and enhancements are added.

Designing the Application

When you design an Objectivity / DB application, you need to answer the following questions:

- What objects are you going to save persistently?
- How do you expect applications to use those objects?

Identifying Classes

The first step in designing any object-oriented application is to identify the classes that capture the structure and behavior of the fundamental entities in the application. The logical modeling phase of application development naturally identifies the classes, the purpose of each, the data that each will contain, and the interrelations among objects of various classes. Your one additional task is to decide which class correspond to objects that will be saved persistently. The modeling phase produces the information you need to define your application's persistence-capable classes and their attributes and associations.

Organizing Persistent Objects

Once you have identified the classes of persistent objects, you need to consider how your application will use the objects. You can then organize them in the federated database. Your organization can make it easy for an application to find the objects it needs when it needs them. It can increase concurrent access to objects that may be needed by different applications that run simultaneously.

For additional information, see Chapter 7, “Organization”.

Implementing and Deploying the Application

As in any software-development project, implementation of an Objectivity / DB application consists of writing code that implements the design; in addition, it requires the following tasks:

- Use an administrative tool to create the federated database that will store the application’s persistent objects.
- Add descriptions of your persistence-capable classes to the federated database schema using the mechanism provided by your programming interface.

Deployment requires not only deploying the application, but also deploying the federated database it will use, or creating the federated database at the deployment site. You may choose to develop a special deployment procedure that initializes the federated database schema and creates any storage objects and persistent collections that your application uses to organize basic objects. For additional information, see Chapter 12, “Deploying to End Users,” in the Objectivity / DB administration book.

Evolving Classes of Persistent Objects

At some point during the lifetime of your application, you may need to modify your class definitions to accommodate new requirements or enhancements. Any change to the data to be saved for a persistence-capable class requires modifying its description in the federated database schema.

If you make such changes during the test phase, you can simply delete the test federated database and create a new one with the new schema descriptions and continue testing. Once you deploy a federated database and database applications to your end users, however, it will not be practical for your end users to delete their federated databases and re-create them if the schema changes. Objectivity / DB therefore provides mechanisms to:

- Evolve a schema based on changes to the class definitions.
- Convert existing persistent objects 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 changes to application-specific classes contained in the schema of a federated database. The changes can include:

- Deleting, adding, or changing attributes or associations defined in, or inherited by, a class
- Renaming a class
- Modifying the inheritance hierarchy

For additional information, see Chapter 5, “Schema Evolution,” in the Objectivity / C++ Data Definition Language book.

Object Conversion

When you change a class description in the 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 / DB implicitly performs object conversion on affected objects when they are accessed. You can also explicitly convert all affected objects in a particular storage object. Certain kinds of changes require explicit conversion.

For additional information, see Chapter 19, “Object Conversion”.

2

Getting Started With Objectivity/C++

This chapter provides an introduction to Objectivity / C++, the C++ programming interface to the Objectivity / DB object-oriented database management system. The chapter introduces:

- Objectivity / C++ programming interface
- Steps involved using Objectivity / C++ to develop an application
- Structure of an Objectivity / C++ application

This chapter assumes you are familiar with the Objectivity / DB concepts and terms introduced in Chapter 1, “Objectivity / DB Basics”.

Objectivity/C++ Programming Interface

The Objectivity / C++ programming interface enables an application to represent Objectivity / DB objects and to manage the interaction between an application and Objectivity / DB. It consists of classes and global types, global constants, and global functions. As usual, definitions of classes, types, constants, and functions are provided in header files. In addition, when the definition of an application-defined persistence-capable class is added to the schema of a federated database, various other classes are generated to support Objectivity / DB operations on that class.

This section gives an overview of the main classes and common data types in the Objectivity / C++ programming interface. See the Objectivity / C++ programmer’s reference for a complete description of all the classes and global names in Objectivity / C++.

Application Objects

Two kinds of application objects control the interaction between an Objectivity / C++ application and an Objectivity / DB federated database: transaction objects and Objectivity contexts.

Transaction Objects

A *transaction object* is an instance of the class `ooTrans`; it controls interaction between an application and the federated database through transactions. A transaction object is transient—that is, it exists in while the application program runs and is not stored in the federated database.

An application calls member functions of a transaction object to start, checkpoint, and commit or abort a transaction. The same transaction object can be used to start and stop any number of transactions, but only one at a time. Chapter 4, "Transactions," provides detailed information about working with transaction objects.

Objectivity Contexts

An *Objectivity context* is an instance of the class `ooContext`; it defines a distinct Objectivity / DB operating environment in which to execute a series of transactions. A single-threaded application has a single Objectivity context; a multithreaded application has one for each thread that interacts with the federated database. Each Objectivity context must have its own transaction object.

Although any given thread can perform only one transaction at a time, a multithreaded application can perform concurrent transactions, each in the Objectivity context of a different thread. Because of Objectivity contexts, transactions executed in multiple concurrent threads are similar to transactions executed in multiple concurrent processes. Chapter 5, "Multithreaded Objectivity / C++ Applications," contains additional information about Objectivity contexts.

Objectivity/DB Objects and Operations

All Objectivity / DB objects are instances of classes that are derived from `ooObj`.

Storage Objects

Objectivity / C++ includes classes for the different kinds of storage object.

Class	Represents
`ooFDObj`	Federated database
`ooDBObj`	Database
`ooContObj`	Container (non-garbage-collectible)
`ooGCContObj`	Container (garbage-collectible)

An application may define additional container classes that derive from either `ooContObj` or `ooGCContObj`.

Chapter 8, “Storage Objects,” provides details about working with storage objects.

Autonomous Partitions

Objectivity / FTO adds the `ooAPObj` class for autonomous partitions. The remainder of this chapter addresses only storage objects and persistent objects. Chapter 27, “Autonomous Partitions,” describes autonomous partitions.

Basic Objects

Objectivity / C++ provides predefined persistence-capable classes for a few kinds of basic objects, such as persistent collections. An Objectivity / C++ application can define additional basic-object classes.

An application defines a persistence-capable class for basic objects by deriving the class from `ooObj`. Like any persistence-capable class, it inherits persistence behavior; see Chapter 9, “Persistent Objects”. A basic-object class also inherits *versioning* behavior, which enables you to create a genealogy of different versions of the same object; see Chapter 20, “Versioning Basic Objects”. You can instantiate a basic-object class either as a persistent object (which is stored in the federated database) or as a transient object (which is not stored).

For additional information about persistence-capable classes, see “Persistence-Capable Classes” on page 140. For complete details about defining persistence-capable classes, see the Objectivity / C++ Data Definition Language book.

Handles

Following the Object Database Management Group (ODMG) standard, an Objectivity / C++ application does not act directly on Objectivity / DB objects.

Instead, the application references an object through a *handle* to the object. Handles are instances of the parameterized classes `ooHandle(`*`className`*`)`. For example, an application works with a database through a handle of class `ooHandle(ooDBObj)`, which references an instance of the Objectivity / C++ database class `ooDBObj`. Similarly, if `Library` is an application-defined class derived from `ooObj`, an application works with a persistent instance of `Library` with a handle of class `ooHandle(Library)`. A handle class is generated automatically for every application-defined persistence-capable class.

A handle is typically initialized by an operation that creates a new object or finds an existing object. The initialized handle contains a unique identifier of the

referenced object. A *null handle* does not reference any object; therefore, it does not have an identifier for any object. You can use a handle in a conditional expression to test whether it references a persistent object or is null. Testing for a null handle is analogous to testing for a null pointer.

Smart Pointers

The handle to a persistent object (container or basic object) is a type-safe "smart pointer" to the referenced object. That is, you can access member functions and data members of the referenced object using the indirect member-access operator (`->`).

As with any C++ object, you can call member functions of the handle itself (as opposed to the referenced object) using the direct member-access operator (`.`) The handle classes define member functions that perform various Objectivity / DB operations on the referenced object; for example, a handle's `update` function opens its referenced object for update.

EXAMPLE This example initializes a handle to reference a persistent object of the `Vehicle` class, calls the `update` member function of the handle, then calls the `rentVehicle` member function of the referenced `Vehicle` object.

```
// Set Handle to reference a Vehicle
ooHandle(Vehicle) vH= ...;

// Use . to call member functions of the handle
if (vH.update()) {
   // Use -> to call member functions of the referenced object
   vH->rentVehicle();
}
```

Memory Management

The handle to a persistent object (container or basic object) contains both the object identifier of the referenced object and state information about the memory representation of that object:

- An *open persistent object* is guaranteed to be represented in memory.
- A *closed persistent object* may be swapped out of memory.

A persistent-object handle transparently manages the means by which it references the object. An *open handle* contains a valid pointer to the memory representation of the referenced object; a *closed handle* does not. A closed handle references the object through the object's unique object identifier, which corresponds to the object's location in the federated database.

For example, if an application finds an object without opening it, the referencing handle is initialized with the object's unique object identifier, but remains closed. The first time the object is accessed through the handle, both the object and the handle are automatically opened, and the handle obtains a pointer to the object's representation in memory. The memory pointer provides fast access to the referenced object for all subsequent operations through the handle until the handle is closed—typically, when the transaction commits. The closed handle no longer has a memory pointer, but it continues to reference the persistent object with the object identifier.

Handles are used in managing memory for persistent objects. As long as an application has an open handle to a particular persistent object, that object's persistent data is kept in memory. When all handles to the object are closed, the referenced object is also *closed*.

Object References

An Objectivity / C++ application can identify an Objectivity / DB object with an object called an *object reference*. An object reference is a wrapper for the object identifier of an Objectivity / DB object; thus, Objectivity / DB can use an object reference to locate the object in the federated database.

Object references are instances of parameterized classes `ooRef(`*`className`*`)`. For example, an object reference of the class `ooRef(ooContObj)` references an instance of the Objectivity / C++ container class `ooContObj`. Similarly, an object reference of the class `ooRef(Library)` references a basic object of the application-defined class `Library`. An object-reference class is generated automatically for every application-defined persistence-capable class.

Object references are used primarily for linking persistent objects through reference attributes, associations, or as elements of a collection. For example, a reference attribute in a persistence-capable class is declared as a data member of type `ooRef(`*`className`*`)`; each fully initialized instance of the defining class will then store an object reference to some instance of *`className`* or any of its derived classes.

EXAMPLE In a car rental application, each vehicle belongs to a particular fleet. The persistence-capable class `Vehicle` has an object-reference data member `fleet`. This member links a vehicle to its containing fleet, allowing an application to find that fleet from the vehicle. When created and initialized, each `Vehicle` object will store an object reference to the appropriate instance of class `Fleet`.

```
class Fleet;             // Forward reference to class Fleet
class Vehicle : public ooObj {
  public:
    ...
    ooRef(Fleet) fleet;   // Link to the containing Fleet
};
```

Object References and Handles

Although their main purpose is to provide persistent references between persistent objects in a federated database, object references to persistent objects can also be used as smart pointers to the referenced objects. The object-reference classes for containers and for basic objects define the same member functions as the corresponding handle classes; they overload the indirect member-access operator (`->`) to access members of the referenced object. Furthermore, because handles can be constructed from object references, you can pass an object reference to any Objectivity/C++ function that takes a handle as a parameter. This means you can use a returned object reference as if it were a handle, with no extra steps.

Object references and handles are not completely interchangeable, however, because they are optimized for different purposes:

- Object references are optimized for storage in persistent objects; they are very inefficient for repeated access to in-memory objects.
- Handles are optimized for accessing objects in memory and cannot be used for storing references in persistent objects—that is, they cannot be used as data members in persistence-capable classes.

For detailed information about handles and references, see Chapter 10, “Handles and Object References”.

Standard and Short Object References

Instances of `ooRef(`*`className`*`)` are sometimes called *standard object references* because they identify objects using whole object identifiers. As an alternative, basic objects can be referenced with *short object references*, which store object identifiers in a truncated format. An application can use a short object reference to link one basic object to another basic objects in the same container. Short object references occupy about half the space of standard object references, so they can be useful when disk space is a concern. See “Saving Storage Space When Linking” on page 235.

A short-object-reference class `ooShortRef(`*`className`*`)` is automatically generated for each basic-object class *`className`*.

Object Iterators

An *object iterator* is an transient object that provides a mechanism for iterating through a group of Objectivity/DB objects. (The name "object iterator" distinguishes it from the other kinds of Objectivity/C++ iterators, for example, iterators for persistent collections.) Object iterators are instances of parameterized classes `ooItr(`*`className`*`)`. Each object-iterator class is derived from the corresponding handle class; for example, `ooItr(ooObj)` is a subclass of `ooHandle(ooObj)`. As a consequence, an object iterator is a special kind of handle. An object-iterator class is generated automatically for every application-defined persistence-capable class.

Many lookup operations initialize an object iterator to find a group of Objectivity/DB objects in the federated database. That group of objects is the object iterator's *iteration set*. As you step through the objects in the iteration set, the object iterator is set to reference each of those objects in turn. You can access the object through the iterator; alternatively, you could set a handle from the iterator.

For detailed information about object iterators and other Objectivity/C++ iterators, see Chapter 14, "Iterators".

Utility Classes

Objectivity/C++ defines non-persistence-capable classes for variable-length arrays (Chapter 12) and strings (Chapter 13). It also defines non-persistence-capable classes that represent information about date and time, as described in the ODMG standard. The string, array, and date/time classes can be used as the types for attributes of persistence-capable classes or for transient data in your application.

The Objectivity/C++ Standard Template Library book describes additional classes that are included with that separately purchased product.

Common Types and Constants

Objectivity/C++ defines global types for primitive numeric values, Boolean values, status codes, and access levels. Many functions in the Objectivity/C++ interface use these types for parameters and return values.

Primitive Numeric Values

Objectivity/C++ provides primitive numeric data types that are stored in the same number of bits on all platforms. These types should be used for attributes of persistence-capable classes instead of C++ types like `int` or `short` whose size may be different on different platforms.

Objectivity / C++ numeric types have mnemonic names that indicate both the data type and the number of bits. For example, the type `int32` is a 32-bit signed integer type; `uint16` is a 16-bit unsigned integer type; `float64` is a 64-bit floating-point type.

Boolean Values

The global type `ooBoolean` represents a Boolean or true / false condition; it can be used for attributes of persistence-capable classes. The global constants of this type are `oocTrue` and `oocFalse`. You can use an `ooBoolean` expression in a C++ condition (for example, the condition of an `if` statement); `oocTrue` is nonzero and `oocFalse` is zero.

Status Codes

Functions that perform Objectivity / DB operations return a status code of type `ooStatus` to indicate whether the operation succeeded. The global constants of this type are the following:

- `oocSuccess` indicates success.
- `oocError` indicates that an error occurred, causing the operation to fail.

  When a function returns `oocError`, one or more Objectivity / DB error messages may be printed.

After any call to an Objectivity / C++ function that returns a status code, you should check the returned code and proceed only if the code is `oocSuccess`. You can use an `ooStatus` expression in a C++ condition (for example, the condition of an `if` statement); `oocSuccess` is nonzero and `oocError` is zero.

Access Levels

Functions that access Objectivity / DB objects generally include an *open mode* of type `ooMode` to indicate the intended level of access to the object. `oocRead` indicates that the application intends to read but not modify the object; `oocUpdate` indicates that the application intends to update the object; `oocNoOpen` indicates that the application has no immediate intention to open the object.

Global Functions

Objectivity / C++ provides a variety of global functions for the following purposes:

- Initialization and cleanup of threads and processes
- Establishing settings for the application as a whole or for the current Objectivity context
- Error handling

- Performance tuning
- Administrative operations on the federated database

ODMG Applications

If you need to build an application that conforms to the ODMG standard, you can use the portions of the ODMG interface that are supported by Objectivity/C++. In many cases, you can substitute ODMG names for equivalent Objectivity/C++ classes and primitive types; for example, you can substitute the ODMG class name `d_Transaction` for the Objectivity/C++ class `ooTrans`. In a few cases, you can use Objectivity/C++ implementations of ODMG-standard classes; for example, you can use the ODMG class `d_Database` instead of `ooHandle(ooFDObj)` to operate on the federated database. For a complete list of class and type equivalences, see Chapter 25, "Conforming to the ODMG Interface".

Objectivity/C++ Application Development

Development of an Objectivity/C++ application consists of the following general steps:

1. Create a federated database.
2. Define persistence-capable classes.
3. Add descriptions of persistence-capable classes to the schema of the federated database.
4. Develop source code for the application.
5. Compile and link source files to produce an executable application.

You can create the federated database (step 1) and define the persistence-capable classes (step 2) in either order; both steps must be performed before you can add descriptions of the classes to the federated database schema (step 3). In practice, you may define the persistence-capable classes (step 2) and develop the application source code (step 4) iteratively and in parallel. You may add classes to the schema (step 3) incrementally.

If you ever need to modify the definitions of existing persistence-capable classes, you must perform schema evolution and possibly object conversion.

The following sections contain brief descriptions of each step in the application-development process.

Creating the Federated Database

You use the oonewfd tool to create a federated database. This tool creates the boot file, which identifies the federated database, and the system database file, which contains the schema and the catalog of all databases and autonomous partitions in the federated database.

Figure 2-1 illustrates the creation of a federated database whose boot file is myFD.

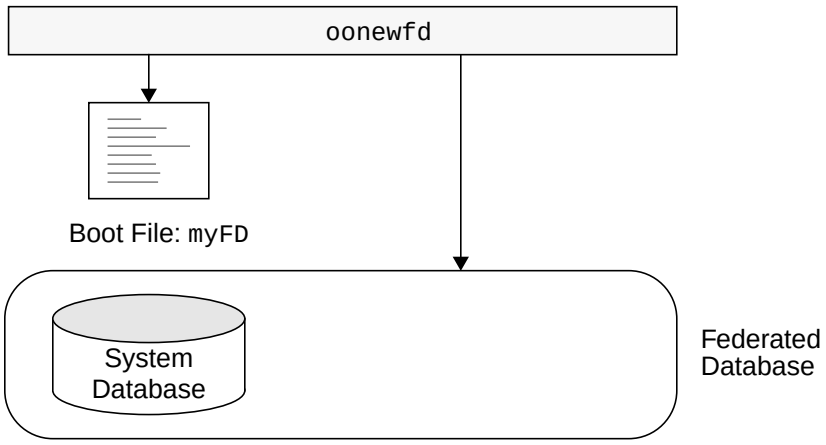


Figure 2-1 Creating a Federated Database

The oonewfd tool is described in the Objectivity/DB administration book.

Defining Persistence-Capable Classes

As discussed in “Designing the Application” on page 42, the design of an Objectivity/DB application includes a logical modeling phase in which you identify the classes your application will use, decide which should be persistence capable, and determine what attributes and associations each one needs.

Logical modeling or analysis may be restricted to a specialized team of developers. To support this specialization, schema development is performed with a separately purchased option to Objectivity/C++, namely, Objectivity/C++ Data Definition Language (Objectivity/DDL). However, even where schema development is restricted to a few people, all application developers must understand the resulting class definitions and how to use them.

You declare persistence-capable classes in one or more *DDL files*. These text files resemble C++ header (.h) files. Each DDL file can have any base name, but the extension *must* be .ddl.

Within a DDL file, declarations for persistence-capable classes are written in the *Data Definition Language* (DDL), which consists of standard C++ syntax with extensions for declaring associations and other Objectivity / DB-specific features. As with C++ header files, you can combine multiple class definitions in a single DDL file, or you can place each persistence-capable class definition in a separate DDL file.

For additional information about persistence-capable classes, see "Persistence-Capable Classes" on page 140. For a detailed description of DDL and the process of defining a persistence-capable class, see the Objectivity / C++ Data Definition Language book.

Adding Class Descriptions to the Schema

After you have created DDL files containing persistence-capable class declarations, you process them using the *DDL processor*.

NOTE Before you run the DDL processor, you must ensure that a lock server is running on the lock server host for the federated database; see the Objectivity / DB administration book.

The DDL processor extracts type information from each class declaration in the DDL files and creates the corresponding class description in the schema of the specified federated database. As you refine your application's logical model, you can incrementally add new persistence-capable class declarations and (re)process the DDL files containing the new classes.

For a detailed description of the DDL processor, see the Objectivity / C++ Data Definition Language book.

For each DDL file *classDefFile*.ddl, the DDL processor generates:

- A primary header file, *classDefFile*.h, which contains C++ definitions of your persistence-capable class, augmented with constructors, operators, and member functions that provide persistence behavior.
- A reference header file *classDefFile*_ref.h. This file contains C++ definitions of the following classes for each persistence-capable class *className* in the DDL file:
  - The handle class ooHandle(*className*)
  - The standard object-reference class ooRef(*className*)
  - The short object-reference class ooShortRef(*className*) if *className* is a class for basic objects (not containers)
  - The object-iterator class ooItr(*className*)

You can use the object-reference classes in your DDL files as the type for object-reference data members. You can use all the generated classes in function definitions in your DDL files or C++ implementation files.

- A method implementation file, *classDefFile*_ddl.cxx, which contains definitions for non-inline functions declared in the primary and reference header files.

NOTE This book uses the filename extension .cxx to indicate C++ implementation files (also called *application code files*). The default extension for implementation files produced by the DDL processor is .cpp on Windows and .C on UNIX.

Figure 2-2 illustrates the results of running the DDL processor on the DDL file myApp.ddl, specifying the boot file myFD to identify the federated database whose schema should be updated.

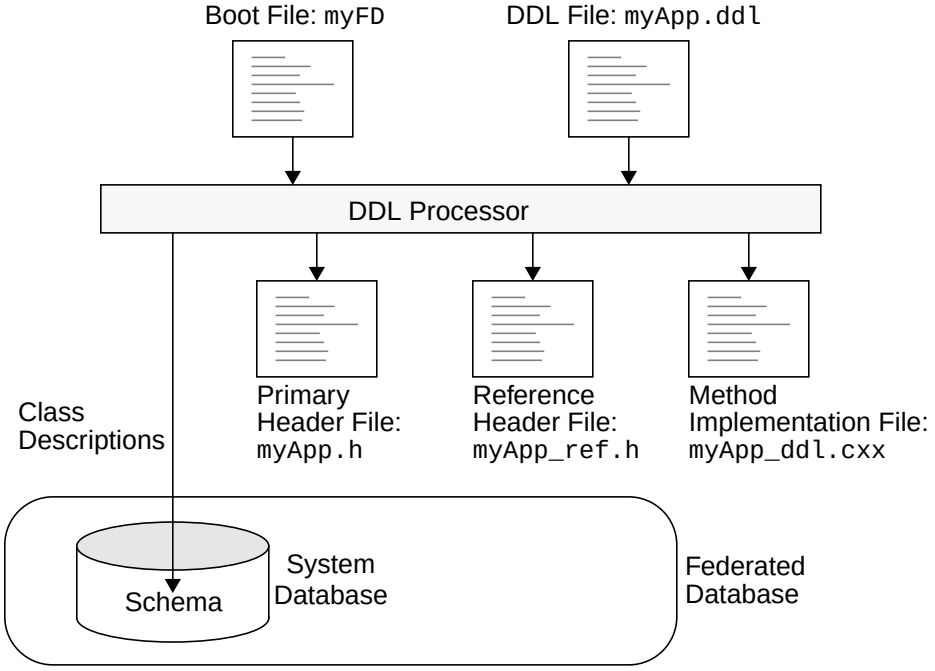


Figure 2-2 Processing DDL Files

Developing Application Source Code

Your application source code can use the Objectivity / C++ programming interface to access the federated database, build its storage hierarchy, and create and work with persistent instances of the classes in its schema. You develop your application source code much as you would do for any C++ application, using standard C++ compilers and tools.

To use classes declared in a DDL file, a source file includes the corresponding primary header file. Each primary header file includes its corresponding reference header file.

All generated header files include the general Objectivity / C++ header file `oo.h`, which defines most Objectivity / C++ global types, constants, functions, and classes. A source file that uses Objectivity / C++ classes and global names and that does not include a generated primary header file must include `oo.h` explicitly.

Figure 2-3 illustrates files for an application that consists of two source files:

- Functions in `main.cxx` use Objectivity / C++ classes to start a transaction and access a federated database; they call functions defined in `myApp.cxx` to access persistent objects. Because `main.cxx` does not use any application-defined classes, it includes `oo.h` explicitly.
- Functions in `myApp.cxx` use the classes declared in the DDL file `myApp.ddl`, so this file includes the primary header file `myApp.h`. It does not need to include either `oo.h` or `myApp_ref.h` because `myApp.h` includes both those files.

Figure 2-3 illustrates how the application's source and header files include the various header files.

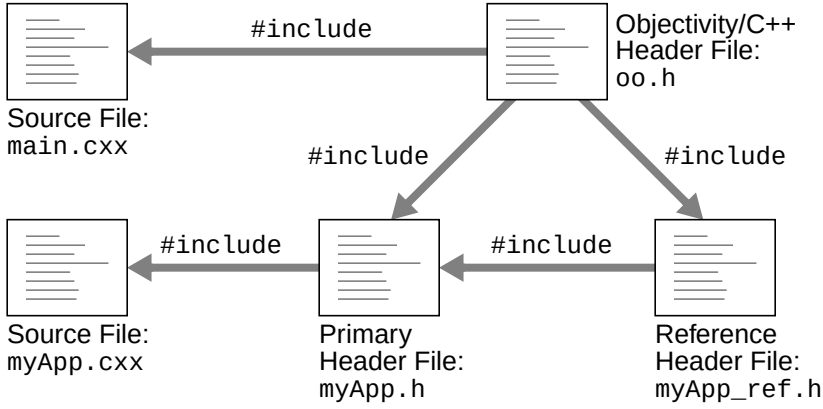


Figure 2-3 Source Files and Included Header Files

If a source file uses certain special-purpose functions or classes, it must include (directly or indirectly) the corresponding Objectivity / C++ header file; see Appendix A, “Objectivity / C++ Include Files”.

Compiling and Linking

When you are ready to build your application, you compile your application source files and the method implementation files that were generated by the DDL processor.

NOTE You can adapt the sample makefiles that are available in the installation directory to facilitate the process of compiling and linking your application; see the *Installation and Platform Notes* for your platform.

Figure 2-4 illustrates the result of compiling source files for the application illustrated in Figure 2-3 on page 57.

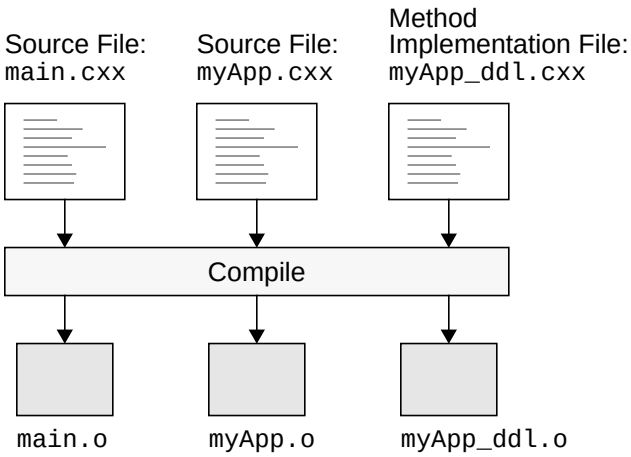


Figure 2-4 Compiling Objectivity/C++ Application Files

You then link the resulting object files with the appropriate Objectivity / DB library file(s) as described in the *Installation and Platform Notes* for your platform.

Figure 2-5 continues the example, illustrating how to build an executable file for the application.

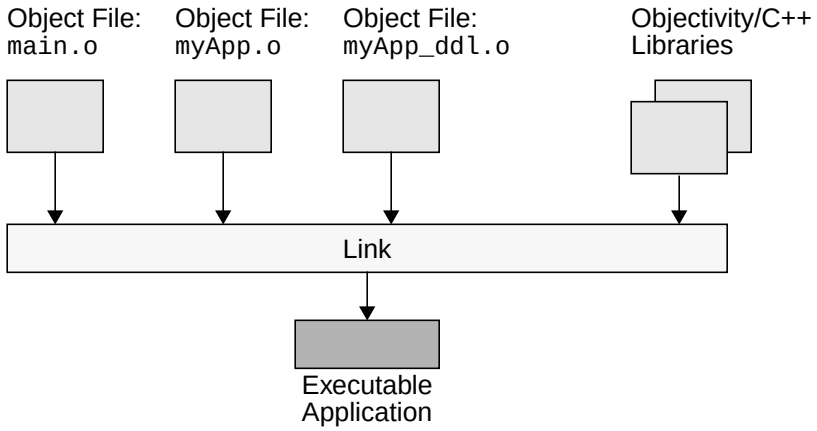


Figure 2-5 Building an Objectivity/C++ Application

Schema Evolution and Object Conversion

During the course of your project, you may find that you need to modify the definition of an existing persistence-capable class declaration after you have added its description to the federated database schema. In that case, you must use the DDL processor either to *evolve* the changed class or to create a new *version* of it. For a detailed description of the DDL processor, class evolution, and class versioning, see the Objectivity / C++ Data Definition Language book.

If your federated database already contains objects of an evolved class, you must *convert* those objects to be consistent with the new class definition. See Chapter 19, “Object Conversion”.

Structure of an Objectivity/C++ Application

An Objectivity / C++ application initializes Objectivity / DB, starts a transaction, opens a federated database, accesses storage objects and persistent objects in the federated database, and terminates the transaction. It may later perform operations on Objectivity / DB objects in other transactions; if it does, it must reopen the federated database at the beginning of each transaction.

The following sections describe the structure of a single threaded application. For information about multithreaded applications, see Chapter 5, “Multithreaded Objectivity / C++ Applications”.

Initialization

Before performing any Objectivity/DB operations, an Objectivity/C++ application must first initialize Objectivity/DB by calling the `ooInit` global function. For most purposes, calling `ooInit` with default parameter values is sufficient; you may choose to try nondefault values when tuning your application. See Chapter 3, "Objectivity/DB Initialization," for additional information.

EXAMPLE This example shows the structure of a typical single-threaded Objectivity/C++ application. Its `main` function calls `ooInit`, then a function that performs some Objectivity/DB operations.

```
// Application code file
#include <oo.h>
...
int main(const int argc, const char *const argv[]) {
   int retval = 0;
   ... // Non-Objectivity/DB operations

   // Initialize Objectivity/DB
   ooInit();

   // Call function that performs Objectivity/DB operations
   retval = dbOperations(argc, argv);

   ... // Non-Objectivity/DB operations
   return retval;
}
```

Beginning and Ending Transactions

An Objectivity/C++ application uses a transaction object to provide the transaction services that guarantee the consistency of the federated database. All Objectivity/DB database operations occur within transactions. You use member functions of the transaction object to control transactions:

- Call the `start` member function to start a transaction.
- Call the `commitAndHold` member function to checkpoint the current transaction, which saves modifications to the federated database without ending the transaction.
- Call the `commit` member function to commit the current transaction.
- Call the `abort` member function to abort the current transaction.

Transactions may not be nested. That is, one transaction must be committed or aborted before another transaction can be started.

Chapter 4, “Transactions,” contains a detailed discussion of transactions.

Opening the Federated Database

An Objectivity/C++ application opens a federated database by calling the `open` member function on a federated-database handle, that is, an instance of `ooHandle(ooFDObj)`. Parameters identify the federated database by its boot file and indicate the intended level of access. If you prefer, you can set the `OO_FD_BOOT` environment variable to specify which federated database to use as a default when you run applications.

Your application must open the federated database at the beginning of each transaction; it must open the same federated database in each transaction.

For additional information, see “Opening a Federated Database” on page 158.

EXAMPLE This example shows a function, `dbOperations`, that starts a transaction, opens a federated database for read-only access, performs some Objectivity/DB operations, and ends the transaction. Note how it tests the status code returned by the `open` member function to see whether the federated database was opened successfully.

```
// Application code file
#include <oo.h>
...
int dbOperations(const int argc, const char *const argv[]) {
   ooTrans trans;           // Transaction object
   ooHandle(ooFDObj) fdH;   // Federated-database handle

   // Start a transaction
   trans.start();
   // Open the federated database
   if (fdH.open("myFD", oocRead) != oocSuccess) {
      cerr << "Failed to open federated database" << endl;
      trans.abort();
      return 1;
   }
   ... // Perform Objectivity/DB operations
   // Terminate the transaction successfully
   trans.commit();

   return 0;
}
```

Objectivity/DB Operations

After opening a federated database, an Objectivity / C++ application can work with the objects in that federated database. The application can create new basic objects, containers, and databases and find existing ones. Having created or found an object, the application can examine or modify its data or delete it. All operations on Objectivity / DB objects must be performed within a transaction.

Most Objectivity / DB operations are performed by calling member functions on handles to Objectivity / DB objects; a few are performed by calling member functions on the Objectivity / DB objects themselves.

- Operations that are particular to one kind of Objectivity / DB object are implemented by member functions of the corresponding class. For example, operations specific to databases are implemented by member functions of the database handle class `ooHandle(ooDBObj)`.
- Operations that are applicable to different kinds of Objectivity / DB objects are generally implemented by member functions of the general-purpose handle class `ooHandle(ooObj)`. Some of these operations are available for all Objectivity / DB objects: the federated database, databases, containers, basic objects, and autonomous partitions. Other operations are available only for persistent objects, that is, containers and basic objects. Still other operations are available only for basic objects. As a consequence, the kind of object referenced by a particular handle of type `ooHandle(ooObj)` determines which member functions you can call on that handle.

Creating and Finding Objects

You use the overloaded `new` operator for the appropriate class to create a basic object, container, or database. In all cases, you assign the pointer returned by `new` to a handle and work with the object through that handle. Parameters to the `new` operators typically specify where in the federated database the new object is to be stored.

As “Finding Objects” on page 40 describes, Objectivity / DB provides several mechanisms for finding objects of various kinds. Because every database has a system name, finding a database by its system name does not require any special setup. In contrast, many of the mechanisms for finding persistent objects require some prior setup; for example, if you want to find a basic object by name, you must give it a name in the scope of some other object.

Typically, the mechanism that finds a database, container, or basic object either sets or returns a handle through which you work with the found object.

EXAMPLE This example opens a federated database, then looks for a database named `InventoryDB` and creates that database if it doesn't already exist.

When a database is created, a parameter to the `ooDBObj` constructor specifies its system name. The `exist` member function of a database handle tests whether a database with the specified system name exists.

```
// Application code file
#include <oo.h>
...
ooTrans trans;           // Transaction object
ooHandle(ooFDObj) fdH;   // Federated-database handle
ooHandle(ooDBObj) dbH;   // Database handle

trans.start();           // Start a transaction

// Open the federated database
if (fdH.open("myFD", oocUpdate) == oocSuccess) {
   // If a database named InventoryDB exists, open
   // it for update access; if not, create it
   if (!dbH.exist(fdH, "InventoryDB")) {
      // Create a database named InventoryDB in
      // the federated database referenced by fdH
      dbH = new(fdH) ooDBObj("InventoryDB");
      if (!dbH) {
         cerr << "Couldn't find or create database";
         cerr << endl;
         trans.abort();
      } // End if database was not created
   } // End if database did not exist
   // Terminate the transaction successfully
   trans.commit();
} // End if federated database was opened successfully
else {
   cerr << "Failed to open federated database" << endl;
   trans.abort();
} // End else federated database couldn't be opened
```

For additional information and examples of creating objects of various kinds, see "Creating a Database" on page 163, "Creating a Container" on page 172, and "Creating a Basic Object" on page 184. For detailed descriptions of the various mechanisms for organizing and finding persistent objects, see Chapter 15, "Creating and Following Links," Chapter 16, "Individual Lookup of Persistent Objects," and Chapter 17, "Group Lookup of Persistent Objects".

Accessing Persistent Objects

Once you have a handle for a persistent object, you can use the handle as a smart pointer to access the data members. You get and set attributes of the object by getting and setting values in its data members, just as you would with any C++ object.

The DDL processor generates special accessor member functions for each association of a persistence-capable class; see "Generated Member Functions" on page 319. You call those member functions (through the handle) to get and set the destination object(s) for the association.

If you plan to modify a persistent object, you should call the handle's `update` member function to obtain an update lock and to notify Objectivity/DB that you intend to modify the object. The generated member functions that set associations call update automatically; if you modify an attribute, you must call `update` explicitly. For additional information, see "Modifying a Persistent Object" on page 193.

WARNING If `update` is not called, your modifications may not be written to the federated database when you commit or checkpoint the transaction.

EXAMPLE The class `Employee` is declared in the file `company.ddl` (not shown); it has two attributes, `name` and `numSupervised`, and a to-one association `manager` that links one employee to another. The DDL processor generates the member function `set_manager` to set the destination object for the `manager` association.

This application code gets handles for two employees: a boss and an assistant. It prints the assistant's name, makes the boss the manager of the assistant, and increments the number of employees that the boss supervises.

```
// Application code file
#include "company.h"
...
ooTrans trans;           // Transaction object
trans.start();
...
// Obtain handles for boss and assistant
ooHandle(Employee) bossH = ...;
ooHandle(Employee) assistantH = ...;

// Access assistant's name data member using the handle as a
// smart pointer.
cout << "Replacing manager for " << assistantH->name << endl;
```

```
// Call the set_manager method to set the manager association;
// this method automatically calls update on the
// assistant handle.
assistantH->set_manager(bossH);

// Call the boss handle's update method to indicate that the
// boss is being modified
bossH.update();

// Increment the number of employees that the boss supervises.
(bossH->numSupervised)++;

// Commit the transaction to update boss and assistant
// objects in the federated database.
trans.commit();
```

Deleting Objects

An application typically deletes an Objectivity / DB object by calling the global function `ooDelete`; the parameter is a handle to the object to be deleted. For additional information, see "Deleting a Database" on page 169, "Deleting a Container" on page 178, and "Deleting a Persistent Object" on page 196.

EXAMPLE This example deletes an employee who has retired, removing that employee object from the federated database.

```
// Application code file
#include "company.h"
…
ooTrans trans;           // Transaction object
trans.start();           // Start a transaction
…
// Get a handle to the employee to be deleted.
ooHandle(Employee) retiredH = …;

// Delete the employee and check that deletion operation
// succeeded.
if (ooDelete(retiredH) != oocSuccess) {
   cerr << "Couldn't delete the employee" << endl;
   trans.abort();
}
…
trans.commit();
```

Part 2 OBJECTIVITY/C++ PROCESSES

This part describes the classes and mechanisms that control Objectivity/C++ application processes.

3

Objectivity/DB Initialization

An Objectivity / C++ application is linked with one or more Objectivity / DB libraries that provide the database services an application uses to create, store, find, and update persistent objects in a federated database. Objectivity / DB must be initialized before an application can invoke any of these services.

This chapter describes:

- General information about the Objectivity / DB initialization process
- How to initialize Objectivity / DB
- Arranging for automatic recovery
- Optional application setup

Understanding the Initialization Process

Objectivity / DB starts to initialize itself automatically when you start an application. To continue with initialization, Objectivity / DB must receive certain information from the application. You supply this information by calling the `ooInit` global function. When the federated database is opened in the first transaction, Objectivity / DB obtains schema and other information from the federated database, which completes the initialization process. See “Opening a Federated Database” on page 158 for information about opening a federated database.

During initialization, Objectivity / DB creates an *Objectivity context* for the main thread in the application. An Objectivity context is the set of data and memory resources required to execute a series of transactions. As part of an Objectivity context, Objectivity / DB earmarks certain process resources for its own use. In particular, Objectivity / DB reserves:

- A portion of the process’s virtual memory for transactions to use when operating on persistent objects; this memory is called the *Objectivity/DB cache*
- A subset of the process’s file descriptors for accessing database files

An Objectivity context also includes the values of error context variables, the currently registered error and message handlers, and so on. In a single-threaded application, the Objectivity context is transparent to you—only one Objectivity context is required, and it is created and destroyed automatically. In contrast, a multithreaded application normally provides an additional Objectivity context for each additional thread that executes transactions; see Chapter 5, “Multithreaded Objectivity / C++ Applications”.

Initializing Objectivity/DB

To initialize Objectivity / DB, you call the `ooInit` global function. You can check the returned status code to see whether initialization succeeded. You can call `ooInit` any time before the first transaction. If your application has multiple threads, you must call `ooInit` in the main thread (the thread that starts implicitly when you start the application) before you start any other threads. You call `ooInit` only one time per application. Subsequent calls to this function are ignored.

Most developers find that *basic initialization*—calling `ooInit` with default parameter values—is sufficient for their applications. During basic initialization, Objectivity / DB registers the predefined Objectivity / DB signal handler, which is used in all Objectivity contexts. In addition, basic initialization provides default values for:

- The initial and maximum sizes of the Objectivity / DB cache in the newly created Objectivity context
- The maximum number of active file descriptors allowed in any Objectivity context

If your application has special requirements for signal handling or process resources, you can perform *customized initialization* by calling `ooInit` with nondefault values. For example:

- If you created your federated database with a nondefault storage page size, you should adjust the minimum and maximum number of buffer pages in the initial Objectivity / DB cache accordingly; see “Optimizing the Page Size” on page 511.
- If you plan to register your own signal handlers (Chapter 22), you may want to suppress the registration of the predefined Objectivity / DB signal handler. If your application runs on a Windows platform and is multithreaded, all application-defined signal handlers that exit the program must call the `ooExitCleanup` function before exiting; see “Preparing Objectivity / DB for Shutdown” on page 103.

Typically, you use basic initialization in the early versions of your application and then adjust the initialization parameters later during performance tuning. For example, you may want to try varying the initial and maximum number of buffer pages to determine whether a different page limit reduces swapping significantly; see "Optimizing the Cache Size" on page 510.

EXAMPLE This example shows an Objectivity / C++ application that performs basic initialization. Its `main` function calls `ooInit` before performing any Objectivity / DB operations.

```
// Application code file
#include <oo.h>
...
int main(const int argc, const char *const argv[]) {
   int retval = 0;
   ... // Non-Objectivity/DB operations

   // Initialize Objectivity/DB
   if (ooInit()) {
      // Call function that performs Objectivity/DB operations
      retval = dbOperations(argc, argv);
      ...
   }
   else {
      cerr << "Unable to initialize Objectivity/DB" << endl;
      retval = 1;
   }
   ...
   return retval;
}
```

The following subsections describe the Objectivity / DB cache and the predefined signal handler in more detail.

Objectivity/DB Cache

The Objectivity / DB cache is a portion of the process's virtual memory that is managed by Objectivity / DB for the purpose of providing fast access to persistent objects. A single-threaded application has one Objectivity / DB cache; a multithreaded application has multiple Objectivity / DB caches (one for each Objectivity context).

Storage Pages and Buffer Pages

Every Objectivity / DB cache consists of *buffer pages*, which are the same size as the *storage pages* in the federated database. You specify the size of a storage page when you create a federated database; your application obtains this size when it first opens the federated database. A page is the minimum unit of transfer to and from disk and across networks. That is, when you access an object in a database, Objectivity / DB reads the storage page or pages containing the object into buffer pages in the cache. Conversely, when you commit a transaction after creating or updating a persistent object, Objectivity / DB writes the buffer pages that contain the object as storage pages on disk.

Cache Components

Objectivity / DB classifies persistent objects according to their storage size:

- A *small object* is one whose persistent data is smaller than a storage page. One or more small objects are stored in a buffer page in the cache.
- A *large object* is one whose persistent data spans multiple storage pages. A large object is stored in the cache in more than one buffer page. The first page, called the *header page*, contains overhead information about the object and a pointer to a single dynamically allocated block of contiguous buffer pages containing the large object.

In general, most persistent data is small, with the exception of very large arrays (such as strings and the data structures that support associations).

An Objectivity / DB cache consists of three components:

- The *small-object buffer pool* of pages containing small objects
- The *large-object buffer pool* of header pages for large objects
- The *large-object memory pool* of dynamically allocated memory blocks for large objects

Cache Size

The initial and maximum sizes of an Objectivity / DB cache is controlled by three attributes of its Objectivity context:

- `nPages` is the initial number of buffer pages in each of the two buffer pools.
- `nMaxPages` is the maximum number of buffer pages in each of the two buffer pools.
- `lgObjMemoryLimit` is the suggested limit for the number of bytes in the large-object memory pool.

By default, the cache initially contains 200 pages in each buffer pool and allows each buffer pool to grow to 500 pages maximum. Different values can be specified for the *nPages* and *nMaxPages* attributes when the Objectivity context is created.

- Parameters to the ooInit global function specify these attributes for the (only) Objectivity context of a single-threaded application, or for the first Objectivity context in a multithreaded application.
- Parameters to the ooContext constructor specify these attributes for each additional Objectivity context in a multithreaded application.

The Objectivity / DB cache for a newly created Objectivity context consists of the two buffer pools; its large-object memory pool is empty. If *pageSize* is the number of bytes per storage page in the federated database, the initial size (in bytes) of the cache is:

```
2 * nPages * pageSize
```

When the Objectivity context is created, the *lgObjMemoryLimit* attribute is initialized to:

```
nMaxPages * pageSize
```

You can change this attribute at any time by calling the ooSetLargeObjectMemoryLimit global function.

The *nMaxPages* attribute is a hard limit—the cache can contain no more than *nMaxPages* small objects and no more than *nMaxPages* large objects. In contrast, the *lgObjMemoryLimit* attribute is a soft limit. Objectivity / DB will try to restrict the large-object memory pool to that size. If, however, the large objects whose headers are in the large-buffer pool require more space than *lgObjMemoryLimit* bytes, the large-object memory pool is allowed to grow to accommodate those large objects.

Given these limits, the maximum expected size of the cache is:

```
2 * nMaxPages * pageSize + lgObjMemoryLimit
```

Caching Small Objects

When a new persistent small object is created or an existing persistent small object is opened, Objectivity / DB reads the small object's storage page to a buffer page in the small-object buffer pool. Objectivity / DB may add buffer pages to this pool, up to the specified maximum number of pages (*nMaxPages*). When this limit is reached, Objectivity / DB swaps out unneeded buffer pages before adding new ones. A page is unneeded if all the persistent objects it contains are closed; see "Reference Counting With Handles" on page 212.

Caching Large Objects

When a new persistent large object is created or an existing persistent large object is opened, Objectivity / DB:

- Reads the large object's header page into the large-object buffer pool.
- Dynamically allocates a block of buffer pages for the large object's data and adds the block to the large-object memory pool.
- Reads the large object's storage pages into the new block.

When the number of header pages in the large-object buffer pool reaches the specified maximum number of pages (*nMaxPages*), Objectivity / DB swaps out unneeded large objects before adding new ones. As with small objects, Objectivity / DB swaps only the large objects that are closed.

If the size of the large-object memory pool reaches the specified limit (*lgObjMemoryLimit*), Objectivity / DB also tries to swap out unneeded large objects before adding new ones. If, however, Objectivity / DB cannot find enough closed large objects to swap out, it will ignore the specified limit and allocate additional pages as needed.

Timing of Cache Operations

When an existing persistent object is opened, Objectivity / DB performs certain operations to prepare the object for representation in the Objectivity / DB cache; when the object is closed, Objectivity / DB reverses those operations. Among these operations are format conversions that enable an application running on one architecture (for example, Windows) to open an object that was created by an application running on a different architecture (for example, one of the UNIX architectures).

The Objectivity / DB cache has two modes for performing format conversions:

- In the default mode, Objectivity / DB converts a persistent object's format immediately upon opening or closing the object. This mode uses less memory and is appropriate for most applications.
- In *hot mode*, Objectivity / DB delays the conversion of a closed object, so that the object can be reopened without being reconverted. This mode uses more memory, but may improve the performance of an application that repeatedly opens objects created by applications on other architectures.

An application can enable and disable hot mode by calling the global function `ooSetHotMode` in a particular Objectivity context. For details, see "Using Hot Mode" on page 514.

Objectivity-Defined Signal Handler

Objectivity / C++ provides a predefined signal handler to respond to various signals that may be raised by the operating environment. By default, the `ooInit` function causes this signal handler to be registered. When an application receives a signal that can cause process termination, this signal handler:

1. Invokes the `ooExitCleanup` function to prepare Objectivity / DB for shutdown. The call to `ooExitCleanup` enables the predefined signal handler to be used by any Objectivity / C++ application. This function may be called by any application, but it is required only by multithreaded applications on Windows platforms. See “Preparing Objectivity / DB for Shutdown” on page 103.
2. Resignals the signal.

See the *Installation and Platform Notes* for your platform for a list of signals that are caught by the predefined signal handler.

Initializing Child Processes

An Objectivity / DB application running on a UNIX platform may create a child process, which may also perform Objectivity / DB operations. The parent process calls the `fork` function to create a child process. The child process must call the `exec` function immediately after being started; this step is necessary even if the child process does not perform any Objectivity / DB operations. If `exec` fails, the child process must call `_exit` (as is standard programming practice).

If the child process performs any Objectivity / DB operations, it must first call the `ooInit` function, just like any normal process. Failure to do so could result in database corruption because both parent and child processes would share the same Objectivity data structures.

Arranging for Automatic Recovery

When an application first interacts with a federated database, it can arrange to perform automatic recovery. When a recovery-enabled application opens a federated database, Objectivity / DB rolls back any incomplete transactions started by applications running on the same client host as your recovery-enabled application. For more information on automatic recovery, see the Objectivity / DB administration book.

You enable automatic recovery by setting the *`recover`* parameter to `oocTrue` when you call the `open` member function on a federated database handle. For performance reasons, you arrange for this parameter to be set to true only once in an application (during the first transaction).

Optional Application Setup

In addition to initializing your application, you may wish to consider the `ooSetAMSUsage` global function for one-time application setup. This function sets the application's policy for using the Advanced Multithreaded Server (AMS).

4

Transactions

A *transaction* is the unit of work that an Objectivity/DB application can apply to a database. A transaction contains one or more logically related operations that create, access, or modify persistent objects. Every interaction with persistent objects must occur within a transaction.

This chapter describes:

- General information about transactions
- Creating a transaction object
- Starting a transaction
- Committing or checkpointing a transaction
- Aborting a transaction
- Guidelines for grouping operations into transactions

Understanding Transactions

Transactions guarantee consistency among 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.

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

Controlling Transactions

A transaction is, in effect, a subsection of an application, the extent of which is determined by four operations: *start*, *commit*, *checkpoint*, and *abort*. The application is said to be *within a transaction* after it starts and until it commits or aborts the 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. During the transaction, any change made to a persistent object is visible only to other operations within the same transaction.

When the application commits the transaction, all modifications to the objects are saved to the federated database, where the modifications become visible to other transactions. A committed transaction cannot be undone.

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 committing and aborting signify the end of the transaction. The local representations of any Objectivity/DB objects are invalidated and any locks on the objects are released. The application can continue to perform other processing, such as operating on transient objects, but it may not operate on persistent objects until it starts another transaction.

Before ending a transaction completely, the application can checkpoint it one or more times. Checkpointing saves modifications to the federated database, but retains the local representations of objects and their locks.

Multiple Transactions

An application may execute any number of transactions. An important part of application design is deciding whether to group database operations into a few large transactions or into many small transactions. Regardless of transaction size, however, note that multiple transactions in the same thread must execute serially. That is, once a transaction has started, it must be committed or aborted before a new transaction can start. Nested or overlapping transactions are *not* allowed.

In multithreaded applications, transactions in different threads can execute concurrently. That is, a separate series of transactions can run in each Objectivity context; see Chapter 5, “Multithreaded Objectivity/C++ Applications”.

Creating a Transaction Object

An application uses a *transaction object* (an instance of the class `ooTrans`) to start and stop its transactions. The following definition creates a transaction object called `transaction`:

```
ooTrans transaction;         // Define a transaction object
```

You can use a single transaction object to start and stop any number of transactions. In a single-threaded application, you normally create one transaction object; in a multithreaded application, you normally create one transaction object in each Objectivity context that is to execute transactions.

You may create additional transaction objects for programming convenience—for example, in each of several local scopes. However, in a given Objectivity context, only one transaction object may be *active* (used to start a transaction) at a time. If you have defined several transaction objects in the same Objectivity context, and you have started a transaction from one of them, you must commit or abort that transaction before starting another transaction, whether from the same or a different transaction object.

Starting a Transaction

You start a new transaction by calling the `start` member function on a transaction object. This member function allows you to specify:

- Whether the transaction uses the MROW concurrent access policy. The MROW policy can improve concurrent access to objects.
- Whether and how long the transaction is to wait for locks on objects that are locked by other transactions.
- How the transaction is to update application-defined indexes to reflect new or changed objects.

By default, `start` starts a standard (non-MROW) transaction that:

- Uses the default lock-waiting option set by the `ooSetLockWait` function.
- Updates indexes automatically at commit time.

Read and Update Transactions

Every transaction must be started either as a *read transaction* or as an *update transaction*, depending on the required level of access to the federated database:

- A read transaction allows operations to obtain read locks only.
- An update transaction permits operations to obtain either read or update locks.

To indicate the required level of access, the first operation of every transaction must be to open the federated database in the appropriate open mode; see “Opening a Federated Database” on page 158. To do this, you call the `open` member function on a federated-database handle, specifying either `oocRead` (the default) or `oocUpdate`. You can promote a transaction from read to update by reopening the federated database with the `oocUpdate` open mode during the transaction.

NOTE Every transaction in an application must open the same federated database; *only one* federated database can be open in a process.

EXAMPLE This example outlines a simple application with a read transaction and an update transaction. The application uses the return values of the `start` and `open` member functions to determine whether to continue.

```
// Application code file
#include "myClasses.h"
…
ooTrans trans;               // Define a transaction object
ooHandle(ooFDObj) fdH;       // Define a federated-db handle
status ooStatus;

status = trans.start();      // Start a transaction
if (status != oocSuccess) {
   cout << "Failed to start transaction." << endl;
   exit(1);
}
status = fdH.open("myFDB");  // Open myFDB for read
if (status != oocSuccess) {  // Test whether myFDB is open
   cout << "Failed to open myFDB." << endl;
   trans.abort();
}
else {                       // If myFDB is open
…                            // Read persistent objects
trans.commit();              // Commit the transaction
}

…                            // Non-Objectivity/DB operations
                             // between transactions

status = trans.start();      // Start a second transaction
if (status != oocSuccess) {
   cout << "Failed to start transaction." << endl;
   exit(1);
}
status = fdH.open("myFDB",
                  oocUpdate);  // Open myFDB for update
if (status != oocSuccess) {    // Test whether myFDB is open
   cout << "Failed to open myFDB." << endl;
   trans.abort();
}
else {                   // If myFDB is open
   …                     // Read or modify persistent objects
   trans.commit();       // Commit the transaction
}
```

Update Transactions and Journal Files

After you start an update transaction (or promote a read transaction to an update transaction), Objectivity / DB automatically records all modifications in a *journal file*. This file is used for restoring the federated database to its previous state if the transaction is aborted or terminated abnormally. The journal file is created in the journal directory specified by the federated database. The same journal file is used by every update transaction in the same Objectivity context; each successive update transaction overwrites the information left by the previous one. A single journal file is created for each Objectivity context that executes an update transaction. Journal files are deleted automatically when the application terminates.

To reduce the performance overhead associated with journal files, you should start a transaction as an update transaction only if update locks will be needed.

Starting the First Transaction

When you open a federated database in the first transaction of the application, you should do so with automatic recovery enabled. See “Enabling Automatic Recovery” on page 159.

Opening the federated database in the first transaction incurs some extra performance overhead because various initialization operations are performed. In subsequent transactions, opening the federated database is faster because the operation simply determines the transaction’s level of access.

Checking Whether a Transaction Object is Active

To ensure that transactions in an Objectivity context execute serially, you can check whether any transaction object is currently active before you start a new transaction. To do this, you call the `isActive` member function on each transaction object to be checked.

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 unless explicitly requested to do so. Consequently, 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 either:

- Commit the transaction.
- Checkpoint the transaction; see “Checkpointing a Transaction” on page 83.

You commit a transaction by calling the transaction object's commit member function. Committing a transaction:

- Saves all newly created or modified Objectivity / DB objects to the federated database.
- Ends the transaction and changes the state of the transaction object to inactive.
- Closes all open persistent objects.
- Closes all handles. The closed handles retain the object identifiers (OIDs) of the objects to which they referred.
- Updates all applicable indexes according to the transaction's index mode; see "Updating Indexes" on page 402.
- Releases any locks acquired in the course of the transaction.

EXAMPLE This example opens a federated database with system name myFDB for read. After some processing, it commits the transaction.

```
// Application code file
#include "myClasses.h"
...
ooHandle(ooFDObj) fdH;
ooTrans trans;
status ooStatus;

status = trans.start();      // Start a transaction
if (status != oocSuccess) {
   cout << "Failed to start transaction." << endl;
   exit(1);
}
status = fdH.open("myFDB");  // Open myFDB for read
if (status != oocSuccess) {  // Test whether myFDB is open
   cout << "Failed to open myFDB." << endl;
   trans.abort();
}
else {                       // If myFDB is open
   ...                       // Read persistent objects
   trans.commit();           // Commit the transaction
}
```

A committed transaction cannot be undone.

To continue working with the database, your application must start a new transaction and reopen the federated database. Because object references and handles retain the object identifiers of the persistent objects to which they refer,

you can reuse them in the new transaction without reinitializing them. Note, however, that a retained object identifier can become invalid between transactions (for example, because another process deleted the corresponding persistent object); in this case, the open operation signals an error.

If a commit operation fails (for example, because the database file is inaccessible), the transaction is aborted and the `commit` member function returns the constant `oocError`.

Checkpointing a Transaction

You *checkpoint* a transaction by calling the transaction object's `commitAndHold` member function. Checkpointing a transaction saves the changes made up to that point, making those changes visible to other transactions. This allows an application to continue as if no interruption of the transaction has occurred. Checkpointing is useful when you want to save the results of a computation during a long transaction, but still retain access to open objects.

Checkpointing a transaction:

- Saves all newly created or modified Objectivity / DB objects to the federated database.
- *Does not close* persistent objects.
- *Does not close* handles.
- Updates all applicable indexes according to the transaction's index mode; see "Updating Indexes" on page 402.
- Implicitly ends the current transaction and immediately starts a new one.
- Retains the same locks, thus preventing other transactions (for example, in a concurrent process) from modifying the open objects.

When you checkpoint an update transaction, the modifications recorded in the journal file prior to checkpointing are overwritten by modifications made after checkpointing.

You can checkpoint multiple times before committing or aborting. If you abort a transaction after checkpointing it, only the changes made after the (last) checkpoint are discarded.

To continue the transaction after invoking `commitAndHold`, your application simply accesses the already-opened persistent objects through their handles.

EXAMPLE This example opens a federated database with system name myFDB for update. After some processing, it checkpoints the transaction; after more processing, it commits the changes. (For simplicity, error-checking code is omitted.)

```
// Application code file
#include "myClasses.h"
...
ooHandle(ooFDObj) fdH;
ooTrans trans;

trans.start();                  // Start a transaction
fdH.open("myFDB", oocUpdate);   // Open myFDB for update
...                             // Work with persistent objects
trans.commitAndHold();          // Checkpoint the transaction
...                             // Work with persistent objects
trans.commit();                 // Commit the transaction
```

Improving Concurrency

By default, the commitAndHold member function causes the application to retain all locks as is. If you are finished updating the objects, you can request that all locks be downgraded to read locks. Doing so permits other processes to gain read access to those objects. You downgrade locks by specifying oocDowngradeAll as the parameter to commitAndHold.

Aborting a Transaction

You abort a transaction by calling the transaction object's abort member function. When you abort a transaction, any changes are discarded (or *rolled back),* leaving the federated database in the logical state it was in before the transaction started. However, certain operations, such as deleting a database or federated database, cannot be rolled back by aborting a transaction; this limitation is indicated in the reference documentation for each such operation.

Aborting a transaction:

- Ends the current transaction and changes the state of the transaction object to inactive.
- Closes all open persistent objects.
- Closes all handles. By default, the object identifiers in the handles are replaced with null. See "Closing Handles" on page 86.
- Releases any locks acquired in the course of the transaction.

EXAMPLE This example opens a federated database with system name `myFDB` for read and performs some processing. The transaction is aborted if `myFDB` cannot be opened; if the transaction continues, it is aborted if *`some condition`* is met.

```
// Application code file
#include "myClasses.h"
...
ooHandle(ooFDObj) fdH;
ooTrans trans;
status ooStatus;

status = trans.start();      // Start a transaction
if (status != oocSuccess) {
   cout << "Failed to start transaction." << endl;
   exit(1);
}
status = fdH.open("myFDB");     // Open the myFDB for read
if (status != oocSuccess) {     // If myFDB is not open
   cout << "Failed to open myFDB." << endl;
   trans.abort();               // Abort the transaction
}
else {                          // If myFDB is open
   ...                          // Read persistent objects
   if (some condition) {        // Test for some condition
      trans.abort();            // If met, abort the transaction
   }
   else {
      trans.commit();           // Otherwise, commit
   }
}
```

If a process ends abnormally during a transaction (for example, because of a hardware or network failure), the locks acquired during the transaction are not released. To clear these locks, you can enable automatic recovery in the application or you can use the `oocleanup` tool to perform manual recovery; see the Objectivity / DB administration book.

Closing Handles

By default, when the `abort` member function closes handles, it replaces their object identifiers with null. In subsequent transactions, these handles must be reinitialized to reference the desired persistent objects before they can be used. In cases where finding the desired objects is time-consuming, you can improve performance by causing `abort` to preserve object identifiers instead of replacing them with null when closing handles. To do this, you specify `oocHandleToOID` as the parameter to the `abort` member function.

NOTE A retained object identifier can become invalid after an aborted transaction (for example, another concurrent process might delete the corresponding persistent object). Opening a handle whose object identifier is invalid signals an error.

Aborting Transactions Automatically

An active transaction is aborted automatically if the application terminates (for example, by calling the `exit` function) or if a call to `commit` or `commitAndHold` fails.

If an external source terminates the process, the locks acquired during the active transaction are *not* released. For example, a programmer debugging an application might terminate the process from the debugger. On the UNIX platform, a user might issue an operating-system command to kill the process by sending a `SIGKILL` signal; on Windows NT, the user might kill the process from the Task Manager window.

To clear any locks that are left by a process that was terminated, follow the procedures described in the chapter on automatic and manual recovery in the Objectivity / DB administration book.

Transaction Usage Guidelines

A transaction can contain any number of Objectivity / DB operations and other application actions. Deciding how to break up a given task into separate transactions involves trade-offs among performance, concurrency, and usability. In making such decisions, you should consider what concurrent-access policies are used by transactions that may be performed concurrently, and how each transaction will lock persistent objects. Chapter 6, “Locking and Concurrency,” discusses both these topics.

Here are some guidelines for deciding how to group operations into transactions:

- **Keep important transactions short.**

  A short transaction is less likely to be aborted for lack of a lock, or for any other reason, because it has fewer operations that might fail. 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 online 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 update locks to read locks at such junctures.

- **Use short transactions for update-intensive objects.**

  When a transaction involves a persistent object that is frequently updated, you can minimize the waiting time for competing applications by keeping the transaction short.

- **Use long transactions when slow network access is involved.**

  Each time a transaction ends, locks are released on the persistent objects that were accessed in that transaction. If some or all of those objects are accessed in the next transaction:

  - Locks must be reacquired, which involves network traffic to and from the lock server.
  - The memory representation of the accessed objects must be refreshed if those objects were updated by a competing application between transactions. This refresh 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 cache activity. Again, checkpointing and lock downgrading can be used to make changes visible incrementally.

5

Multithreaded Objectivity/C++ Applications

An Objectivity / C++ application can use multiple threads to execute multiple concurrent transactions within a single process. For example, consider a server application that responds to requests from local or remote client applications. Such a server may respond to each client request by creating a thread that executes an independent series of transactions with a federated database.

Objectivity / C++ supports POSIX threads for UNIX platforms and MS Win32 threads for Windows platforms.

This chapter describes:

- General information about threads in an Objectivity / C++ application
- Initializing Objectivity / DB in a multithreaded application
- Initializing, programming, and terminating a thread that performs Objectivity / DB operations
- Reusing an Objectivity context in multiple threads
- Changing a thread's Objectivity context
- Preparing Objectivity / DB for shutdown

Objectivity/C++ and Threads

Every Objectivity / C++ application has a main thread, which is the thread that executes when you start the application (that is, when the application's `main` function is called). In a multithreaded application, the main thread typically creates one or more additional threads and waits for them to complete. Your application can set up the main thread and / or any number of other threads to execute transactions with a federated database.

Executing transactions in multiple concurrent threads is similar to executing them in multiple concurrent processes. The same transaction and locking semantics apply in both cases. Objectivity / DB prevents two transactions from

updating the same object simultaneously, whether from different threads or different processes.

Objectivity Contexts

As in any multithreaded application, the threads in an Objectivity / C++ application share general process resources. However, within the shared process resources, each thread that performs Objectivity / DB operations must have a separate *Objectivity context* in which to execute its transactions. Each Objectivity context is a complete set of data and memory resources that Objectivity / DB manages. These resources include:

- An Objectivity / DB cache, its initial and maximum sizes, and the policies governing its behavior; see "Cache Size" on page 72 and "Timing of Cache Operations" on page 74
- Dynamically allocated memory for caching large objects, and the limit on this memory; see "Cache Size" on page 72
- The current values of Objectivity / C++ error context variables, which keep information about the last error that occurred; see Chapter 23, "Error Handling"
- The currently registered error and message handlers; see Chapter 23, "Error Handling"
- Policies that govern transaction behavior (such as the lock-wait policy; see "Lock Waiting" on page 120)

At process initialization, an Objectivity context is created automatically for an application's main thread. The application must provide an Objectivity context for each additional thread that is to execute Objectivity / DB transactions. Each thread typically has its own unique Objectivity context, although it is possible for the same Objectivity context to be reused by multiple threads in turn. Objectivity contexts are implemented as instances of the `ooContext` class.

In an application with multiple Objectivity contexts, each context has its own Objectivity / DB cache, its own values for error context variables, and so on. When a particular thread executes database operations, however, these operations interact with only one Objectivity context—namely, the *current Objectivity context* that has been set by the application for the executing thread. The operation uses the current Objectivity context's resources, such as its Objectivity / DB cache.

NOTE In Objectivity documentation, phrases such as "the Objectivity / DB cache" or "the value of an error context variable" always refer to the content of the current Objectivity context for the thread under discussion.

Transactions, Threads, and Objectivity Contexts

Each Objectivity context defines a distinct Objectivity / DB operating environment that is shared by only the transactions that are executed in it. In effect, each Objectivity context defines a single, independent series of transactions, analogous to the series of transactions executed in a single-threaded process. When an application creates multiple Objectivity contexts, it can execute multiple independent series of transactions (one series per context), all within the same process.

Objectivity / C++ supports several models for executing a series of transactions within a multithreaded application. In the simplest model, you execute a single series of transactions entirely within a single thread. In this case, your application provides the thread with a single Objectivity context for the life of the thread, and destroys the context when the thread terminates.

In some applications, it may make sense to extend a series of transactions across several threads. For example, a transaction in one thread may populate the Objectivity / DB cache of its current Objectivity context with data that can be used by transactions in a subsequent thread. In this case, your application can preserve the Objectivity context from the first thread and pass it to the second thread. The application must, however, ensure that the two threads are not using the same Objectivity context at the same time.

In some cases, you may want a single thread to execute several separate series of transactions. For example, you may want one thread to run three sequential tests, where each test contains multiple transactions. You can ensure the independence of these tests by running them in separate Objectivity contexts. This also allows you to tune the Objectivity / DB cache sizes for each test individually; see “Optimizing the Cache Size” on page 510.

Preemptive Multithreading

Objectivity / C++ multithreading libraries implement preemptively scheduled threads, which means that switching among multiple concurrent threads is controlled outside the application. Whenever threads are switched, Objectivity / DB automatically switches Objectivity contexts as appropriate.

The following Objectivity / DB operations block the executing thread without blocking the entire process:

- Acquiring locks
- Releasing locks
- Reading objects from disk
- Writing objects to disk
- Writing recovery information to a journal file

Initializing Objectivity/DB

You initialize Objectivity / DB in a multithreaded application the same way you initialize a single-threaded application—by calling the `ooInit` function before the first transaction (see Chapter 3, "Objectivity / DB Initialization").

In a multithreaded application, you must call `ooInit` in the main thread before you create any other threads that will perform Objectivity / DB operations.

EXAMPLE This example shows a simple outline for a multithreaded Objectivity / C++ application. In this outline, the `main()` function invokes `ooInit` before the first thread is created.

Because multithreading libraries differ across platforms, the examples in this chapter use pseudocode (*`createThread`*) for the calls that create threads. Such calls normally include a reference to the function to be executed by the thread, along with any parameters to that function.

```
// Application code file
#include <oo.h>
…
int main(const int argc, const char *const argv[]) {
   int retval = 0;
   … // Non-Objectivity/DB operations

   // Initialize Objectivity/DB
   if (ooInit()) {
      …
      createThread (…,&Func1,…,parameters,…); // Pseudocode
      …
      createThread (…,&Func2,…,parameters,…); // Pseudocode
      …
   }
   else {
      cerr << "Unable to initialize Objectivity/DB" << endl;
      retval = 1;
   }
   …
   return retval;
}
```

In a multithreaded application the `ooInit` function performs process-wide initialization, specifying:

- Whether or not to register the predefined Objectivity / C++ signal handler.
- The maximum number of active file descriptors allowed in any Objectivity context.

In addition, the `ooInit` function initializes the application's main thread for Objectivity / DB. Specifically, `ooInit`:

- Calls the `ooInitThread` function for the main thread. This initializes the thread and creates an Objectivity context for it.
- Sets the initial and maximum Objectivity / DB cache sizes in the main thread's Objectivity context. You should probably reduce these cache sizes if you do not plan to execute transactions in the main thread.

Initializing Threads

You must initialize every thread that is to execute Objectivity / DB operations. The main thread is initialized automatically when you call the `ooInit` function. You use the `ooInitThread` function to initialize each additional thread. The call to `ooInitThread` must precede all other Objectivity / DB operations in a thread.

Initializing With a New Objectivity Context

By default, the `ooInitThread` function creates a new Objectivity context (a new, dynamically allocated instance of `ooContext`) for the thread in which it is executed. This supports the common design in which each thread has its own Objectivity context for the life of the thread.

The new Objectivity context is created by the `ooContext` constructor with default parameter values, which specify the default initial and maximum sizes of the Objectivity / DB cache in the context. If you want a customized cache size, you must create the Objectivity context explicitly, as described in "Initializing With an Existing Objectivity Context" on page 94.

EXAMPLE This application creates a thread to execute the myFunc function, which calls ooInitThread before invoking constructors or starting a transaction.

```
// Application code file
#include <oo.h>
…
void myFunc(parameters…) {
   ooInitThread();          // Initialize thread with new
                            // context
   ooTrans transaction;
   ooHandle(ooFDObj) fdH;
   transaction.start();
   fdH.open("bootFilePath");
   …
   transaction.commit();
   ooTermThread();       // Terminate use of Objectivity/DB
                         // (see page 100)
} // End myFunc

int main(const int argc, const char *const argv[]) {
   int retval = 0;
   … // Non-Objectivity/DB operations
   if (ooInit())
      …
      createThread (…,&myFunc,…,parameters,…); // Pseudocode
      …
   }
   else {
      cerr << "Unable to initialize Objectivity/DB" << endl;
      retval = 1;
   }
   …
   return retval;
} // End main
```

Initializing With an Existing Objectivity Context

You can initialize a thread with an existing context. To do so, you pass a pointer to the desired Objectivity context as a parameter to ooInitThread.

For example, you may want a thread to use an Objectivity context that has nondefault Objectivity / DB cache size. This is usually appropriate if you created your federated database with a nondefault storage page size or while you are tuning the application's performance; see "Optimizing the Page Size" on page 511 and "Optimizing the Cache Size" on page 510. To customize the cache

size for an Objectivity context, use parameters to the ooContext constructor to specify the desired initial and maximum sizes for the Objectivity / DB cache, expressed as numbers of pages (see "Objectivity / DB Cache" on page 71).

EXAMPLE This application creates a thread to execute the myFunc function, which calls ooInitThread to initialize the thread with a newly created Objectivity context with nondefault cache size.

```
// Application code file
#include <oo.h>
...
void myFunc(parameters...) {
   // Create an Objectivity context with nondefault cache size
   ooContext* context(300, 600);
   // Initialize thread to use that Objectivity context
   ooInitThread(context);
   ooTrans transaction;
   ooHandle(ooFDObj) fdH;
   transaction.start();
   fdH.open("bootFilePath");
   ...
   transaction.commit();
   ...
   ooTermThread();       // Terminate use of Objectivity/DB
                         // (see page 100)
}
```

If you want to initialize a thread to reuse an existing Objectivity context that was previously used by a different thread, you must preserve that context as described in "Preserving the Current Objectivity Context" on page 101.

Initializing With a Null Context

You can initialize the thread with a null Objectivity context and then set the current Objectivity context for the thread as described in "Changing the Current Objectivity Context" on page 97.

To initialize a thread with a null Objectivity context, pass 0 (a null pointer) as the parameter to ooInitThread:

```
ooInitThread(0);          // Initialize with null context
```

NOTE If you initialize a thread with a null context, you must set its context before you perform any Objectivity / DB operations.

Using Objectivity/C++ in Threads

For the most part, you use Objectivity / C++ in a thread just as if you were programming a single-threaded application—you invoke Objectivity / DB operations to create transactions, set modes and limits, provide for error handling, and so on. However, because a multithreaded application operates in multiple Objectivity contexts, you must observe the behavior and restrictions described in the following subsections.

Operations That Set Context-Specific State

Certain Objectivity / DB operations affect only the current Objectivity context of the thread in which they are executed. Such operations must be invoked once per context rather than once per process.

For example, Objectivity / C++ error handlers are specific to the Objectivity context in which they are registered. Therefore, to use a nondefault error handler in multiple threads, you should register the error handler in the Objectivity context for each of these threads. This means executing the `ooRegErrorHandler` function (and any related functions, such as `ooSetErrorFile`) in each thread. Chapter 23, “Error Handling,” contains more information about error handlers.

Operations that affect resources that are managed by an Objectivity context must be invoked on a per-context basis. These operations include:

- Enabling or disabling hot mode by calling the `ooSetHotMode` function.
- Dynamically allocated memory for caching large objects, and the limit on this memory by calling the `ooSetLargeObjectMemoryLimit` function.
- Setting the lock-wait policy by calling the `ooSetLockWait` function.
- Setting a timeout period for communicating with the lock server or AMS by calling the `ooSetRpcTimeout` function.

Error Context Variables

The Objectivity / C++ error context variables (such as `oovLastError`) keep information about the last error that occurred in the thread’s current context. Like global variables, these expressions are visible to all functions in the application, but they can have different values in each Objectivity context. Chapter 23, “Error Handling,” explains how to examine and set the values of the error context variables.

Restricted Use of Objectivity/C++ Transient Objects

An application uses various Objectivity/C++ transient objects to manage its interaction with persistent objects. These transient objects include transaction objects, handles, and iterators. Such objects store a state that is specific to a single series of transactions and therefore to a single Objectivity context.

To preserve integrity of the federated database, transactions in one Objectivity context may not manipulate Objectivity/C++ transient objects that are defined in another context. This means, for example, that you cannot pass a transaction object or a database handle between threads that have different current contexts. To do so produces undefined results.

This restriction applies to persistent objects. If multiple concurrent threads need to access the same persistent object, they must do so through separate transactions, using separate handles. Each thread will access a separate in-memory representation of the object in the Objectivity/DB cache of its own Objectivity context—just as if each thread were a separate process.

The restriction does *not* apply to:

- Objectivity contexts (instances of `ooContext`). These transient objects can be passed between threads—for example, to extend a series of transactions from one thread to another. Your application must ensure that each Objectivity context is used by only one thread at a time, however.
- Object references (instances of `ooRef(`*`className`*`)`). Because they contain no context-specific state, object references can be passed between Objectivity contexts. Doing so is similar to reusing an object reference in a new transaction without reinitializing it; your application should check the validity of a passed object reference before using it in the new context.

You cannot use global variables, file-scope variables, and static member variables to provide interthread access to Objectivity/C++ transient objects. Objectivity/C++ considers the values of such variables to be part of the main thread's Objectivity context.

Changing the Current Objectivity Context

You can change a thread's current Objectivity context by calling the static member function `ooContext::setCurrent`. The parameter to this function is a pointer to the desired thread. You can specify 0 (a null pointer) as the parameter to change the current context to a null Objectivity context.

For example, you may want one thread to run a number of sequential tests, where each test contains multiple transactions. You can ensure the independence of these tests by running them in separate Objectivity contexts.

EXAMPLE This application creates a thread to execute the `myFunc` function, which in turn calls two test functions (`test1` and `test2`). Each test function creates its own Objectivity context, sets this context as the current context, runs its transactions, and sets the current context to null.

Because each Objectivity context must have its own transient data, this application bundles an `ooContext` object along with the associated transient data (in this case, a transaction object and a federated-database handle) in an object of class `ContextWrapper`.

Each test function has a local `ContextWrapper`, created by the default constructor for that class. That constructor calls the default constructors for `ooContext`, `ooTrans`, and `ooHandle(ooFDObj)` to initialize its data members.

```
// Application code file
#include <oo.h>
...
class ContextWrapper {
   ooContext context;
   ooTrans trans;
   ooHandle(ooFDObj) fdH;
};

void test1(parameters...) {
   ContextWrapper cw;
   // Set the thread's current context to this function's
   // local Objectivity context
   ooContext::setCurrent(&(cw.context));
   // Start transaction using the transaction object
   // bundled with the local Objectivity context
   cw.trans.start();
   cw.fdH.open("bfpath");
   ...
   cw.trans.commit();
   ...
   ooContext::setCurrent(0);    // Set current context to null
}
```

```
void test2(parameters…) {
   ContextWrapper cw;
   // Set the thread’s current context to this function’s
   // local Objectivity context
   ooContext::setCurrent(&(cw.context));
   // Start transaction using the transaction object
   // bundled with the local Objectivity context
   cw.trans.start();
   cw.fdH.open("bfpath");
   …
   cw.trans.commit();
   …
   ooContext::setCurrent(0);    // Set current context to null
}

void myFunc(parameters…) {
   ooInitThread(0);      // Initialize thread with no context
   …
   test1();              // Run test1 in its context
   …                     // Non-Objectivity/DB operations
   test2();              // Run test2 in its context
   …
   ooTermThread();       // Terminate use of Objectivity/DB
                         // (see page 100)
}

int main(const int argc, const char *const argv[]) {
   … // Non-Objectivity/DB operations
   if (ooInit()) {
      …
      createThread (…,&myFunc,…,parameters,…); // Pseudocode
      …
   }
   …
}
```

Terminating a Thread's Use of Objectivity/DB

You must explicitly terminate the use of Objectivity / DB in a thread before the thread terminates. That is, every thread that has been initialized with `ooInitThread` or `ooInit` must call the function `ooTermThread` as its last Objectivity / DB operation, unless the thread is to terminate due to process termination. So, for example, the main thread need not invoke `ooTermThread` if it is to terminate when a user action causes the process to exit.

In addition to terminating a thread's use of Objectivity / DB, the `ooTermThread` function destroys the thread's current Objectivity context. Under some circumstances, however, you should use `ooTermThread` without destroying the current Objectivity context. See the following subsections for more information.

You cannot reinitialize a thread after it has invoked `ooTermThread`. If your purpose is to delete the thread's current Objectivity context and then create a new one for the same thread, see "Changing the Current Objectivity Context" on page 97.

Destroying the Current Objectivity Context

If the thread to be terminated has a nonnull current Objectivity context, the `ooTermThread` function automatically destroys that context. This supports the common design in which a thread has a single Objectivity context for the life of the thread and this Objectivity context is not used in any other thread.

The `ooTermThread` function uses the `delete` operator to delete the current Objectivity context. This is appropriate for deleting a dynamically allocated context such as one created by `ooInitThread`. If, however, you initialized a thread with an Objectivity context that cannot be deleted in this way, you must prevent `ooTermThread` from deleting the context; see "Preserving the Current Objectivity Context" on page 101.

EXAMPLE This application creates a thread to execute the myFunc function, which calls ooTermThread to terminate the thread's use of Objectivity / DB. Because the current Objectivity context is nonnull, ooTermThread deletes it.

```
// Application code file
#include <oo.h>
...
void myFunc(parameters...) {
   ooInitThread();          // Initialize; create context
   ooTrans transaction;     // Transaction object
   ooHandle(ooFDObj) fdH;   // Federated db handle

   transaction.start();     // Start a transaction
   fdH.open("bootFilePath");
   ...
   transaction.commit();
   ...
   ooTermThread();          // Terminate use of Objectivity/DB
}

int main(const int argc, const char *const argv[]) {
   ... // Non-Objectivity/DB operations
   if (ooInit()) {
      ...
      createThread (...,&myFunc,...,parameters,...); // Pseudocode
      ...
   }
   ...
}
```

Preserving the Current Objectivity Context

You can terminate a thread's use of Objectivity / DB without destroying the current Objectivity context. You must do this if:

- You want to preserve the current Objectivity context beyond the life of the thread—for example, to reuse it in another thread.
- The thread was initialized with an Objectivity context that should not be destroyed by the delete operator—for example, because this context is statically allocated, is allocated on the stack, or is a member of some other object.

To terminate a thread's use of Objectivity / DB without destroying the current Objectivity context, you set the current context to null before calling `ooTermThread`—for example:

```
ooContext::setCurrent(0);    // Set current context to null
…                            // Non-Objectivity/DB operations
ooTermThread();              // Terminate use of Objectivity/DB
```

You can then reuse the preserved Objectivity context or perform the appropriate operations to destroy it.

Reusing an Objectivity Context

In some applications, it may make sense to reuse an Objectivity context across a series of threads. For example, a transaction in one thread may populate the current Objectivity / DB cache with data that can be used by transactions in a subsequent thread. Or, the main thread may maintain a pool of available Objectivity contexts that can be passed to newly created threads, which return their contexts to the pool upon completion.

NOTE If your application reuses Objectivity contexts, you must ensure that only one thread at a time uses a given Objectivity context.

EXAMPLE This example shows a simple scheme for providing a thread with an existing Objectivity context; the thread preserves that context before terminating.

```
// Application code file
#include <oo.h>
…
void myFunc(parameters…, ooContext *someContext) {
   ooInitThread(someContext);  // Initialize thread with
                               // specified context
   ooTrans transaction;
   ooHandle(ooFDObj) fdH;
   transaction.start();        // Start a transaction
   fdH.open("bootFilePath");
   …
   transaction.commit();
   …
   ooContext::setCurrent(0);   // Set current context to null
   ooTermThread();          // Terminate use of Objectivity/DB
} // End myFunc
```

```
int main(const int argc, const char *const argv[]) {
   ... // Non-Objectivity/DB operations
   if (ooInit()) {
      ...
      ooContext *myContext = new ooContext;   // Create context
      createThread (...,&myFunc,..., myContext); // Pseudocode
   }
   ...
}
```

Preparing Objectivity/DB for Shutdown

During normal process termination, Objectivity / DB calls various destructors to shut itself down. To prepare Objectivity / DB for shutdown, a multithreaded application running on a Windows platform must call the ooExitCleanup global function before the application exits. This function leaves Objectivity / DB in a safe state for process termination. On a Windows platform, ooExitCleanup ensures that the Objectivity / C++ dynamic load libraries (DLLs) terminate properly.

You should call this function before returning from your main function and before any call to exit (for example, in an application-defined signal handler). It is good programming practice to terminate all threads that perform Objectivity / DB operations before you invoke ooExitCleanup.

The ooExitCleanup function:

- Aborts all active transactions in all threads.
- Leaves the calling thread executing, along with any thread that has no Objectivity context, or whose Objectivity context has been set to null.
- Suspends or terminates any other thread (that is, any thread with a nonnull Objectivity context). Whether threads are suspended or terminated depends on the platform. You must not attempt to restart any thread suspended by ooExitCleanup.

The call to ooExitCleanup must be the *last* Objectivity / DB operation in an application. In particular, an application must call ooExitCleanup after all threads have finished performing Objectivity / DB operations and after all instances of Objectivity / C++ classes have been destructed.

WARNING If ooExitCleanup is not the last Objectivity / DB operation in an application, undefined results (such as an access violation or data corruption) may occur.

To ensure that all instances of Objectivity / C++ classes are destructed before your call to ooExitCleanup:

- You must explicitly delete any dynamically allocated instances of Objectivity / C++ classes before you call ooExitCleanup.
- You must not use global instances of Objectivity / C++ classes.
- You must not declare instances of Objectivity / C++ classes inside the same block that contains a call to ooExitCleanup.

EXAMPLE This example shows a simple outline for a multithreaded Objectivity / C++ application on a Windows platform.

```
// Application code file
#include <oo.h>
...
int main(const int argc, const char *const argv[]) {
   int retval = 0;
   ... // Non-Objectivity/DB operations
   // Initialize Objectivity/DB
   if (ooInit()) {
      ...
      createThread (...,&Func1,...,parameters,...); // Pseudocode
      ...
      createThread (...,&Func2,...,parameters,...); // Pseudocode
      ...
      createThread (...,&FuncN,...,parameters,...); // Pseudocode

      ...                  // Wait until all threads terminate
      ooExitCleanup();     // Prepare Objectivity/DB for shutdown
   }
   else {
      cerr << "Unable to initialize Objectivity/DB" << endl;
      retval = 1;
   }
   ... // Non-Objectivity/DB operations
   return retval;
}
```

The call to ooExitCleanup is *required* in any multithreaded application that will run on a Windows platform and any multithreaded application that must support future portability to Windows. However, an application that is intended to run *only* on UNIX platforms can omit the call to ooExitCleanup, because it is invoked implicitly during the current UNIX shutdown process.

6

Locking and Concurrency

Transactions in multiple applications (or in multiple threads of a single application) can share access to the same federated database at the same time. Objectivity / DB uses a mechanism called *locking* to guarantee that concurrent transactions leave the data in a well-defined, consistent state.

This chapter describes:

- General information about Objectivity / DB locks
- Locking a persistent object, a database, or a federated database
- Managing locks
- Concurrent access policies
- Lock conflicts
- Disabling the locking mechanism

Understanding Locks

The objects in an Objectivity / DB federated database can be shared by multiple concurrent transactions that may, at times, try to perform incompatible operations on those objects. For example, two transactions may read, and then subsequently update, an object. If both transactions perform these actions simultaneously, one of the updates would be overwritten by the other update. Objectivity / DB uses *locks* to prevent multiple transactions from performing incompatible operations on the same federated database.

Kinds of Locks

An application requests different kinds of locks to inform Objectivity / DB how it plans to use an Objectivity / DB object in a transaction. When an application requests a *read lock*, the application indicates that it needs read-only access to an object. When an application requests an *update lock*, the application indicates that it intends to modify the object.

Certain Objectivity / DB operations, such as creating or deleting a container, request *exclusive locks*. An exclusive lock indicates that any other concurrent operation on the locked object is considered incompatible.

Operations that open a database or federated database place a special kind of lock on it, called an *intention lock*. An intention lock simply indicates that the transaction may also hold a read, update, or exclusive lock lower in the storage hierarchy. You normally don't need to be aware of intention locks, although they may be reported by certain administration tools.

Limits on Locks

Each transaction places an overall limit on the kinds of locks that its operations can obtain:

- An update transaction permits operations to obtain read, update, or exclusive locks.
- A read transaction allows operations to obtain read locks only.

The current open mode of the federated database determines whether a transaction is an update transaction or a read transaction. Each transaction sets the open mode when it opens the federated database; you can promote the open mode from read to update by reopening the federated database during the transaction. See "Read and Update Transactions" on page 79.

NOTE Only read locks can be obtained for objects in a read-only database, even if the requesting transaction is an update transaction. See "Making a Database Read-Only" on page 168.

Units of Locking

An application can request a lock on a basic object, container, database, or federated database. However, locks are actually granted only on containers, databases, and the federated database—not on individual basic objects. Thus, *containers* are the smallest unit of locking for persistent objects. In particular:

- When an application requests a lock on a container, the container itself is locked.
- When an application requests a lock on a basic object, the container in which the object resides is locked.

Locking a single basic object effectively locks all the basic objects in the same container. This is a performance advantage for a transaction that needs to access multiple objects in the same container; such a transaction can obtain the necessary permissions through a single lock request.

Lock Requests

An Objectivity / C++ application can request locks implicitly or explicitly. Regardless of how they originate, lock requests are granted or refused by an Objectivity / DB lock server.

Implicit and Explicit Requests

An Objectivity / C++ application normally requests locks *implicitly*. That is, when the application calls a function that accesses one or more objects, Objectivity / DB automatically generates implicit requests for all the necessary locks. A function that reads an object will obtain a read lock; a function that modifies an object will obtain an update lock. If a function affects multiple objects, Objectivity / DB implicitly obtains locks for all the required objects; for example, deleting a container puts an exclusive lock on the container and an intention lock on the database and federated database that contained it.

Implicit locking obtains access rights to resources as they are needed by an application, which is generally 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 transactions from modifying objects critical to the operation.

An application needing to reserve access to all required objects in advance can *explicitly* lock the 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 transaction can modify them as long as they are locked. An application requests a lock *explicitly* by calling the `lock` member function on a handle to the desired object.

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 29, “In-Process Lock Server”.

NOTE An application bypasses 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 168.

Lock Compatibility

When servicing a lock request on an object, the lock server looks at any existing locks held by other transactions, and determines whether the requested lock is compatible with the existing locks. In general:

- A requested read lock is always compatible with one or more existing read locks.
- A requested update lock is always incompatible with any existing update lock.
- A requested exclusive lock is always incompatible with any existing read or update lock.

The lock server applies a *concurrent access policy* to determine whether a requested read lock is considered compatible with an existing update lock; see “Concurrent Access Policies” on page 113.

If a requested lock cannot be granted, a *lock conflict* occurs, and the member function that generated the lock request will signal an error. If you don’t want Objectivity / DB to abort the request immediately, you can configure your application to wait for a period of time in case the locked object becomes available; see “Handling Lock Conflicts” on page 120.

Lock Duration

Locks are held until the application commits or aborts the transaction, at which time all locks obtained during the transaction are released. During the transaction, you can change read locks to update locks, but you cannot explicitly release locks before the end of the transaction; see “Managing Locks” on page 112.

Until locks are released, the potential exists for lock conflicts. Depending on your application’s concurrency requirements, you can consider various techniques for reducing the probability of lock conflicts; see “Strategies for Avoiding Lock Conflicts” on page 119.

NOTE Certain operations result in locks on Objectivity / DB-internal objects. You may notice that locks on such objects are held only as long as necessary, and may be released before the end of a transaction.

Locking a Persistent Object

Containers are the smallest unit of locking for persistent objects; when an application requests a lock on a basic object, the lock is granted on the container in which the basic object resides. In effect, locking a single basic object locks all of the basic objects in the same container.

When an application requests a lock on a persistent object, the lock server applies the transaction's concurrent access policy to determine whether the requested lock is compatible with any existing locks on the relevant container.

Implicitly Locking a Persistent Object

An application normally requests locks implicitly—by performing various Objectivity / DB operations that automatically generate the appropriate lock requests. The following table lists common operations for which Objectivity / DB implicitly obtains locks on persistent objects.

Operation	Implicit Locks Obtained
Opening a container or basic object for read or update	A read or update lock on the opened container or on the container of the opened basic object.
Creating or deleting a basic object	An update lock on the object's container.
Creating or deleting a container	An exclusive lock on the container.
Copying a basic object	A read lock on the object's container and an update lock on the container that contains the copy.
Moving a basic object	An update lock on the object's original container and on the container to which it was moved.
Scanning a federated database or database for persistent objects	A read lock on every container in the federated database or database being scanned.
Scanning a container for basic objects	A read lock on the container being scanned.
Explicitly locking a composite object for read or update	Read or update locks on the containers of any objects that are linked through associations with lock propagation enabled.
Deleting a composite object	Update locks on the containers of any objects that are linked through associations with delete propagation enabled.

Explicitly Locking a Persistent Object

To minimize the possibility of a lock conflict while a transaction is in progress, you can explicitly request all the locks you might need at the beginning of the transaction. For example, if your application requires guaranteed access to the objects in an iteration set, you should explicitly lock all the relevant containers before you initialize and advance the iterator.

Explicit locking reserves access to persistent objects without bringing them into virtual memory. In contrast, explicitly opening a set of persistent objects in advance not only locks the objects, but also brings them all into virtual memory, possibly long before they are required.

To explicitly lock a persistent object, you call the `lock` member function on a handle to the object, passing a constant of type `ooLockMode` to specify the kind of lock to be obtained (`oocLockRead` or `oocLockUpdate`). You cannot explicitly request an exclusive lock on a persistent object.

When you lock a persistent object that has associations for which lock propagation is enabled, the `lock` method also locks the destination objects linked by those associations. If you want to lock a source object without locking its destination objects, you can call the `lockNoProp` member function on a handle to the source object. For additional information about lock propagation along association links, see “Propagating Operations” on page 148.

Locking a Database or Federated Database

Locking a database or federated database limits the level of concurrent access that is permitted to its contents:

- Locking a database for read or update prevents another transaction from concurrently opening the database (or its contents) for update.
- Opening and locking a federated database for read prevents another transaction from concurrently opening it (or its contents) for update.
- Opening and locking a federated database for update places an exclusive lock on it, so that no other transaction can concurrently open it at all.

NOTE Locking a database or federated database is *not* equivalent to locking its containers individually. For example, assume that container `C` resides in database `D`, which is locked for update. A concurrent non-MROW transaction can subsequently lock container `C` for read, which would not be permitted if locking the database also locked the container itself.

Implicitly Locking a Database or Federated Database

An application normally locks databases or federated databases implicitly:

- An operation that reads or modifies a database or a federated database implicitly obtains a read, update, or exclusive lock on it, as appropriate. Such operations include tidying a federated database, deleting a database, and so on.
- An operation that simply *opens* a database or federated database implicitly obtains an *intention lock* on it:
  - An intention lock is granted on a database whenever it is opened, either explicitly (for example, by a call to the `open` member function on a database handle) or implicitly (for example, when one of the database's containers is opened or locked).
  - An intention lock is granted on the federated database each time it is opened at the beginning of each transaction. An additional intention lock is placed on the federated database whenever one of its databases or containers is opened or locked.

An intention lock essentially exists to inform other transactions that an object lower in the storage hierarchy might be locked; the type of intention lock indicates whether the object is potentially locked for read or update.

Intention locks are always compatible with each other. This allows, for example, two transactions to concurrently open the same database for update (provided, of course, that they update different containers in that database).

An intention update lock is incompatible with any (non-intention) read, update, or exclusive lock. Consequently, opening a database for update prevents a concurrent transaction from locking the database for read or update.

Explicitly Locking a Database

You can explicitly place a read lock or an update lock on a database. Either kind of lock prevents other transactions from concurrently opening the database (or its contents) for update.

To explicitly lock a database, you call the `lock` member function on a handle to the database, passing a constant of type `ooLockMode` to specify the kind of lock to be obtained (`oocLockRead` or `oocLockUpdate`).

Explicitly Locking a Federated Database

You can explicitly place a read lock or an exclusive lock on a federated database.

A read lock prevents other transactions from concurrently opening the federated database (or its contents) for update. To explicitly place a read lock on a federated database:

1. Open the federated database for read.
2. Call the `lock` member function on a handle to the federated database, passing the constant `oocLockRead`.

An exclusive lock prevents all concurrent access to the federated database; no other transaction can concurrently open it for read or update. To explicitly place an exclusive lock a federated database:

1. Open the federated database for update.
2. Call the `lock` member function on a handle to the federated database, passing the constant `oocLockUpdate`.

Managing Locks

An application can upgrade and downgrade locks within a transaction; locks are released at the end of a transaction.

Upgrading Locks

You can upgrade the lock of an object from read to update by using the `lock` member function on a handle to the object. Alternatively, you can open the object for update, using either the `open` or `update` member function on a handle to the object.

Downgrading Locks

You can downgrade (from update to read) all locks obtained during a transaction by checkpointing the transaction. Checkpointing saves all the changes made thus far to the federated database, while keeping the transaction active and retaining its locks. The locks can be retained as is, or you can convert them all to read locks, which makes the locked objects available for other transactions to read.

To downgrade locks while checkpointing a transaction, you call the transaction object's `commitAndHold` member function, passing the parameter `oocDowngradeAll`.

NOTE You cannot downgrade an individual update lock to a read lock, nor can you downgrade locks without checkpointing the transaction.

Releasing Locks

Committing or aborting a transaction automatically releases *all* the locks obtained in that transaction, thus permitting other transactions to access the objects. Objectivity / C++ does not provide a way to release locks explicitly during a transaction.

Concurrent Access Policies

Objectivity / DB allows multiple transactions to read a container simultaneously and prevents multiple transactions from updating a container simultaneously. It supports two concurrent access policies for controlling whether one transaction can update a container while one or more transactions are reading the same container. The transactions requesting simultaneous access may be executed by different processes, or by different Objectivity contexts within the same process.

When a transaction requests a lock for a container that has already been locked by one or more other transactions, the concurrent access policies of the transactions determine whether the lock request is compatible with the existing locks. Objectivity / DB supports two concurrent access policies: *standard* and *multiple readers, one writer (MROW)*.

You specify a transaction's access policy through the first parameter to the transaction object's `start` member function. The policy determines whether the transaction can obtain a read lock on a container that is already locked for update. Objectivity / DB's general access rules determine whether the transaction can obtain locks in other situations.

Standard Policy

The *standard* concurrent access policy prevents a transaction from viewing data that may be in the process of being altered by another transaction. A transaction using the standard policy is sometimes called a *standard transaction*.

When a standard transaction requests a read lock on a container, it can obtain the lock only if the container is not locked for update by any other transaction. If the

container is already locked for update, the lock request fails. Furthermore, as long as a standard transaction has a read lock on a container, no other transaction can obtain an update lock on that container. A read lock held by a standard transaction is called a *non-MROW read lock*.

To specify the standard policy, you pass the constant `oocNoMROW` to the transaction object's `start` member function. If you omit parameters to `start`, the default is to start the transaction using the standard policy.

Multiple Readers, One Writer (MROW) Policy

The MROW concurrent access policy relaxes the restriction that a container (and its contents) may not be simultaneously updated and read. A transaction using the MROW policy is sometimes called an *MROW transaction*; it can read the last-committed or checkpointed version of a container while another transaction updates the same container.

The MROW policy is appropriate for applications that 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 federated database. If the application uses the standard access policy to read web pages, it cannot read a web page when the webmaster is updating that page. However, if the application uses MROW transactions, it can read a version of the web page at all times.

When an MROW transaction requests a read lock on a container, it can obtain the lock even if the container is already locked for update. Furthermore, an MROW transaction's read lock does not prevent another transaction from obtaining an update lock on that container. A read lock held by an MROW transaction is called an *MROW read lock*.

The MROW policy is particularly useful for transactions that read from containers that are infrequently updated, such as libraries.

WARNING The MROW policy allows a transaction to read data that is potentially out of date. Therefore, an MROW transaction should not read data from one container and use that data as the basis for updates to objects in a different container. To avoid making updates based on out-of-date information, you should use the MROW policy only for transactions that read objects, not for transactions that update objects.

You start an MROW transaction by passing the constant `oocMROW` as the first parameter to the transaction object's `start` member function.

General Access Rules

Both concurrent access policies apply the following general access rules to any transaction requesting a lock:

- The transaction can obtain either a read or update lock on a container that is not locked by any other transaction.
- The transaction can obtain a read lock on a container that is locked for read by another transaction.
- The transaction can obtain an update lock on a container if both of the following conditions are met:
  - The container is not locked for update by any other transaction.
  - The container is not locked for read by any standard transaction; the container may, however, be locked for read by an MROW transaction.

Neither access policy allows a container to be updated by two transactions simultaneously. Consequently, to achieve maximum concurrency in your application, you should assign your basic objects to containers based on the expected usage profiles of the objects. “Assigning Basic Objects to Containers” on page 131 discusses a variety of strategies for assigning objects to containers.

Summary of Access Rules

The following table lists the kinds of locks that can be requested and the success or failure of each request, given the existing locks on the container. As the table shows, a transaction’s concurrent access policy affects its requests for read locks, but not its requests for update locks.

Request	Existing Locks on Container		
	0 Update **≥ 0 MROW Read**[a] **0 Non-MROW Read**	**0 Update** **≥ 0 MROW Read**[a] **≥ 1 Non-MROW Read**[b]	**1 Update** **≥ 0 MROW Read**[a] **0 Non-MROW Read**
MROW transaction requests read lock	Request succeeds	Request succeeds	Request succeeds
Standard transaction requests read lock	Request succeeds	Request succeeds	Request fails
Any transaction requests update lock	Request succeeds	Request fails	Request fails

a. Zero or more MROW read locks.
b. One or more non-MROW read locks.

Example of Access Rules

In the following example, three applications, A, B, and C, need to access objects in the same container; none of the applications activates lock waiting.

Existing locks: None

1. Application A starts a transaction and attempts to open an object in the container for update, implicitly requesting an update lock on the container:

```
Atrans.start(...);      // Start a standard or MROW transaction
...
AobjH.open(oocUpdate);
```

No other transaction has a lock on the container, so A's transaction is granted an update lock and its call to `open` returns `oocSuccess`.

Resulting locks: A: Update

2. Application B starts a transaction using the standard access policy. It attempts to open an object in the container for read, which implicitly requests a non-MROW read lock on the container:

```
Btrans.start();       // Start a standard transaction
...
if (BobjH.open(oocRead) == oocError)
   Btrans.abort();
```

Because B's transaction uses the standard access policy and A's transaction already holds an update lock on the container, application B's request for a read lock is denied; its call to `open` returns `oocError`. Application B chooses to abort its transaction.

Resulting locks: A: Update

3. Application C starts a transaction using the MROW access policy. It attempts to open an object in the container for read, which implicitly requests an MROW read lock on the container:

```
Ctrans.start(oocMROW);       // Start an MROW transaction
...
CobjH.open(oocRead);
```

Because C's transaction uses the MROW policy, its request for a read lock is granted even though A's transaction already holds an update lock on the container.

Resulting locks: A: Update
C: MROW read

4. Application A commits its transaction, which releases its update lock:

```
Atrans.commit();
```

Resulting locks: C: MROW read

5. Application B starts another transaction using the standard access policy and again attempts to open an object in the container for read:

```
Btrans.start();        // Start a standard transaction
...
BobjH.open(oocRead);
```

Because no other transaction has an update lock on the container, B's transaction is granted a read lock and its call to `open` returns `oocSuccess`.

Resulting locks: C: MROW read
B: Non-MROW read

6. Application A starts another transaction and again attempts to open an object in the container for update, implicitly requesting an update lock on the container:

```
Atrans.start(...);      // Start a standard or MROW transaction
...
if (AobjH.open(oocUpdate) == oocError)
   Atrans.abort();
```

Because the container is locked for non-MROW read by application B, application A's request for an update lock is denied; its call to `open` returns `oocError`. Application A chooses to abort its transaction.

Resulting locks: C: MROW read
B: Non-MROW read

7. Application B commits its transaction, which releases its read lock.

```
Btrans.commit();
```

Resulting locks: C: MROW read

8. Application A starts another transaction and again attempts to open an object in the container for update, implicitly requesting an update lock on the container:

```
Atrans.start(...);      // Start a standard or MROW transaction
...
AobjH.open(oocUpdate);
```

Because C's transaction uses the MROW policy, application A's request for an update lock is granted even though C's transaction already holds a read lock on the container.

Resulting locks: C: MROW read
A: Update

Note that the success or failure of the various lock requests in this example are the same regardless which access policy application A uses. Because A updates objects, however, it would use the standard access policy.

Managing Containers Under MROW

If you read from a container during an MROW transaction, another transaction may concurrently update the container or any objects in it. If the updating transaction commits, your view of the container becomes out of date. You can refresh the reading transaction's view of the container by calling the `refreshOpen` member function on a handle to the container.

Refreshing a container opens the most recently committed version. This means that each open object in the container must be closed so it can be reopened the next time you access it. The `refreshOpen` operation either closes all such objects on request, or signals an error if any objects are still open.

If the transaction's view of the container is still current, `refreshOpen` performs no action. The member function reports whether a refresh was actually performed in a Boolean parameter you pass to it.

EXAMPLE This code fragment shows a typical way of refreshing a container in an MROW read transaction.

```
// Application code file
#include "myClasses.h"
…
ooTrans trans;
ooHandle(ooFDObj) fdH;
ooHandle(ooDBObj) dbH;
ooHandle(myContainer) contH;

…
trans.start(oocMROW);
fdH.open("myFDB");    // Open the federated database for read
…
contH.open(dbH, "myContainer", oocRead);
…
ooBoolean beenUpdated;
contH.refreshOpen(oocRead, &beenUpdated);
if (beenUpdated == oocTrue) {
   // Container refreshed with current contents
}
```

If you want to test whether an MROW transaction's view of a container is out of date without actually refreshing it, you can call the `isUpdated` member function on a handle to the container.

Lock Conflicts

A *lock conflict* occurs when two concurrent transactions request incompatible locks on the same object. As you design your application to meet your concurrency requirements, you should consider strategies for reducing the probability of causing, or being blocked by, lock conflicts. Where lock conflicts are unavoidable, you must decide how your application will respond to them.

Strategies for Avoiding Lock Conflicts

You should analyze your application requirements and, when possible, employ locking strategies that reduce the probability of lock conflicts. In addition to the following strategies, see “Maximizing Concurrency” on page 506.

- **Use MROW whenever appropriate.**

  If reading potentially stale data is acceptable, the you can use the MROW concurrent access policy to enable transactions to read objects in a container that has been locked for update by another transaction. (Standard transactions cannot read an update-locked container or any basic objects in that container.)

- **Hold update locks for short time periods.**

  You can avoid lock conflicts by minimizing the time that objects are locked for update. You can minimize this time period in any or all of the following ways:

  - Keep the transaction short. 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.
  - Delay requests for update locks until you actually need to modify the objects. If your application first needs to read an object and then later needs to update the object, you can upgrade the lock from read to update. Of course, if another transaction also has the object locked for read, you may not be able to upgrade the lock.
  - Checkpoint the transaction to downgrade the locks when you have finished making modifications.

- **Open objects for read when iterating.**

  You can avoid lock conflicts by enabling Objectivity / DB to use a special locking procedure during certain iteration operations. When this procedure is used, Objectivity / DB locks and then immediately unlocks each container that is visited by an object iterator finding the persistent objects in a database or a federated database—for example, when you call:

  - The `scan` member function on an object iterator of class `ooItr(ooObj)`, `ooItr(ooContObj)`, or `ooItr(`*`appClass`*`)`.

- The `contains` member function on a database handle.

You enable Objectivity/DB to unlock each visited container by:

- Specifying the iterator's open mode to be either `oocNoOpen` or `oocRead`, so that the container is, at most, locked for read. If the iterator's open mode is `oocUpdate`, the resulting update lock is held until the end of the transaction.
- Ensuring that any open handles to the container or to a basic object in it are explicitly closed or set to null before the iterator visits the next container. For example, if you set a handle to reference a found object during the iteration, and then use the handle to access one of the object's members, you must explicitly close the handle before the iterator advances.

Note: Each visited container must be locked only as a result of the iteration itself, and not because of some prior operation. A pre-existing read lock is not removed.

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 your application to respond to this situation in one of two ways:

- Immediately give up on the operation and return to the application with an error condition. This is the default behavior.
- Wait until the desired object is available.

Lock Waiting

You can activate *lock waiting* to queue lock requests for an object in the lock server. Lock waiting allows a transaction to wait for an object that is locked by another transaction. When the object is unlocked, the waiting transaction is granted a lock on the object.

You can activate lock waiting for an individual transaction by setting the *`waitOption`* parameter of the transaction object's `start` member function. You can specify that the transaction is to wait indefinitely for locks or you can specify a finite timeout period (in number of seconds).

Alternatively, you can activate lock waiting for all subsequent transactions in the same Objectivity context by using the `ooSetLockWait` global function to specify either a finite or an infinite timeout period. Subsequent transactions use this timeout period if they are started with the default *`waitOption`* value. If you call `ooSetLockWait` within a transaction, it takes effect during that transaction, overriding any *`waitOption`* you specified when starting the transaction.

You can turn off lock waiting for an individual transaction by passing `oocNoWait` as the *`waitOption`* parameter to `start`. Similarly, you can turn off lock waiting for subsequent transactions by calling the `ooSetLockWait` global function with the value `oocNoWait`.

NOTE An MROW transaction determines immediately whether it can get a requested lock, and ignores any timeout period specified by the `ooSetLockWait` function.

Deadlock Detection

A *deadlock* is a circular condition in which two or more transactions are each waiting indefinitely for a lock that will never become available because of the circularity.

For example, a deadlock is created when both of the following conditions are simultaneously true:

- Standard (non-MROW) transaction T1 has a read lock on container A and is waiting indefinitely for an update lock on container B.
- Standard (non-MROW) transaction T2 has a read lock on container B and is waiting indefinitely for an update lock on container A.

Because both transactions have lock waiting activated for an indefinite timeout period, neither transaction will have a chance either to withdraw its lock request or to terminate and release its locks.

When a lock is requested by a transaction with indefinite lock waiting enabled, Objectivity / DB checks to see whether queueing the request would result in a deadlock. If so, an error is signaled, and that transaction is aborted.

No deadlock checking is performed for transactions with finite lock waiting enabled. In this case, Objectivity / DB assumes that any deadlock will be broken when the timeout period expires.

Disabling the Locking Mechanism

If your application is guaranteed exclusive access to a federated database and requires maximum performance, you can consider disabling the locking mechanism for your application. For example, you might disable locking for a single-user, single-threaded application in which only one user has file-system permissions to access the federated database files and directories, and that user never runs concurrent applications. An application in which locking is disabled is sometimes called a *standalone application*.

WARNING Do not disable locking if there is *any* chance that another application will access the same data at the same time as your application. Disabling the locking mechanism can result in data corruption.

You disable locking for your application by calling the `ooNoLock` global function anytime after calling `ooInit`, but before starting a transaction.

Disabling the locking mechanism improves performance by removing the runtime overhead associated with managing lock; however, it also removes the concurrent access protection afforded by locking. If another process is accessing the same data as your application, unpredictable results may occur, including corruption of your data. For most applications, the benefits of locking (data integrity, concurrent access, and so on) far outweigh the slight performance gain they could obtained if they disables locking. For improved performance without disabling the locking mechanism, you can consider starting an in-process lock server (see Chapter 29, “In-Process Lock Server”).

Part 3 OBJECT MODEL

This part describes the classes and mechanisms that Objectivity/C++ uses to model persistent data, and the auxiliary classes through which an Objectivity/C++ application works with persistent objects.

7

Organization

Because Objectivity / DB is an object-oriented database management system, it organizes persistent data into objects. Interrelationships between the various objects are stored in the federated database explicitly as links between related objects. In contrast, a relational database management system organizes persistent data into rows of tables; interrelationships between rows are not stored explicitly, but can instead be computed by join operations.

The individual persistent objects in a federated database can be organized in various ways to improve the performance of applications. This chapter introduces the Objectivity / C++ classes and mechanisms used to model and organize data. It describes:

- How the Objectivity / DB object model organizes persistent data
- Object graphs
- Grouping the persistent objects in a federated database to limit search
- Assigning basic objects to containers
- Persistence-capable classes, which define the object model for an Objectivity / C++ application

Understanding the Object Model

The fundamental units of organization in the Objectivity / DB object model are *persistent objects* (basic objects and containers). Like the objects in a nondatabase application, persistent objects have *attributes* for containing application-specific data. For example, an `Employee` object can have attributes for storing a name, an address, an identification number, and so on. In an Objectivity / DB application, the data in a persistent object's attributes is saved persistently and shared across applications.

The Objectivity/DB object model allows you to further arrange persistent objects into higher-level structures within the federated database:

- Linking mechanisms allow you to model relationships between objects, connecting them to form an *object graph.* See "Object Graphs" on page 127.
- Grouping mechanisms allow you to define arbitrary groups of objects that support the manner in which you expect applications to use the objects. You can make it easy and efficient to find the objects that are relevant for a particular task without searching the entire federated database. See "Grouping Persistent Objects to Limit Search" on page 130.
- The Objectivity/DB storage hierarchy allows you to distribute basic objects among multiple containers to improve the runtime performance of applications that use those objects. See "Assigning Basic Objects to Containers" on page 131.

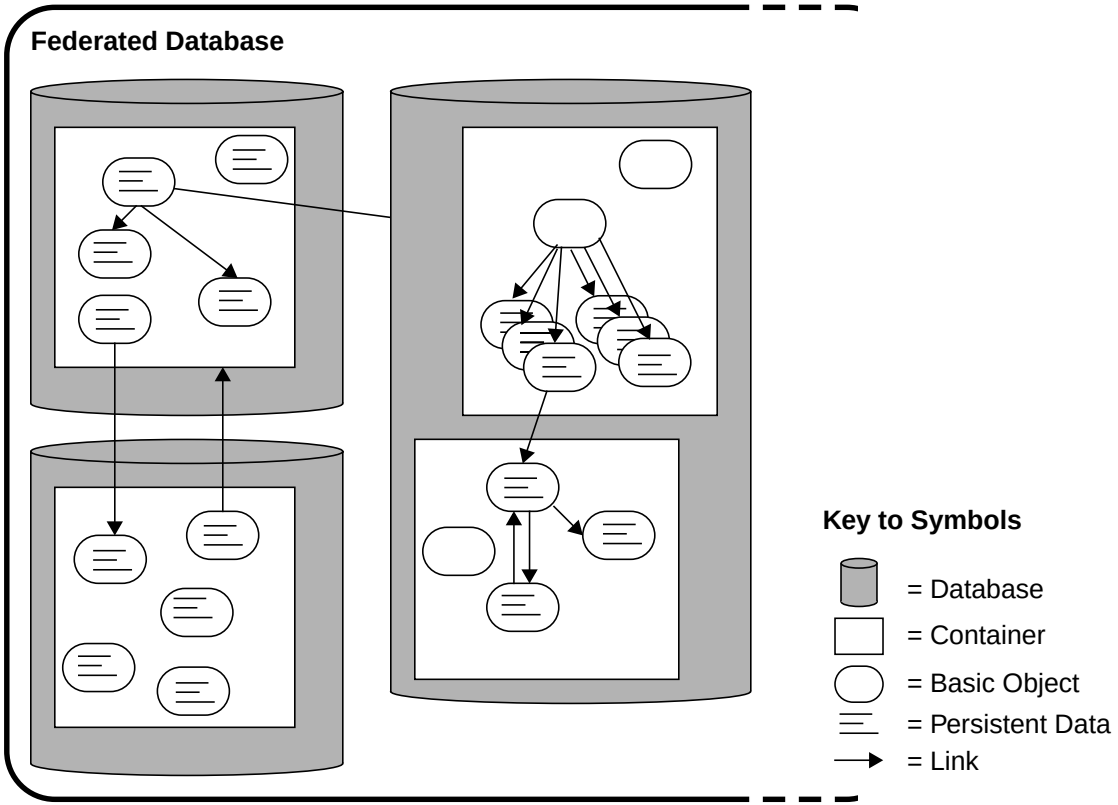


Figure 7-1 Federated Database With Multiple Levels of Organization

Every persistent object is an instance of a *persistence-capable class*, which defines both the data to be saved persistently for the object and the links it can have to other persistent objects. Thus, when defining persistence-capable classes in Objectivity / C++, you must take into account not only the kind of persistent data to be saved, but also the higher-level structures into which instances are to be organized. See “Persistence-Capable Classes” on page 140.

NOTE Objectivity / DB does not save the “behavior” of persistence-capable classes. That is, it does not save the member functions of a C++ class.

Object Graphs

An *object graph* is a directed-graph data structure that models a group of related persistent objects. Each node in the graph is a persistent object. Each arc is a link from a *source object* to a *destination object*; it represents a relationship that exists between the two objects. In a typical object graph, one persistent object, considered the “root,” is the starting point for links that connect the objects together. In the simplest graph, which contains a single link, the source object is the root.

Two common kinds of object graph are:

- Composite objects, whose links represent a “part-of” or “has-part” relationship among different kinds of object. See “Composite Objects” on page 128.
- Persistent collections, whose links represent the membership of elements in an aggregate. See “Persistent Collections” on page 129.

An individual composite object or persistent collection can be embedded within a larger object graph—that is, some source object within the larger graph can have a link to the root of the composite object or the persistent collection.

In general, the links in an object graph can model any kind of relationship between the source and destination objects. For example, if a building is being designed by an architect, the corresponding building object could have a link to the appropriate architect object representing a “designed-by” relationship. Furthermore, the various links in an object graph can represent a variety of different relationships. For example, the architect object that is the destination of the designed-by link could be the source object of an employed-by link to a company object.

Linking Mechanisms

The object model provides three alternative mechanisms for linking objects together:

- Persistent collections (page 129) provide their own linking mechanism to link a persistent collection (the source object) to the objects it contains (the destination objects).
- An application-defined class can contain reference attributes (page 145) that link a source object of the class to destination objects.
- An application-defined class can contain associations (page 145) that link a source object of the class to destination objects.

An object graph is created at run time, when the application creates persistent objects and forms links between them. The various linking mechanisms can be combined in any way. For example, an attribute or association can link the source object to a persistent collection; one persistent collection can be an element of another persistent collection; an element of a persistent collection can have a reference attribute linking it to an object that has an association linking it to another object. Chapter 15, "Creating and Following Links," describes how to construct and traverse an object graph using the various linking mechanisms.

Composite Objects

A *composite object* is a group of persistent objects that, together, contain the information about one complex entity. For example, consider an architectural design application that works with a group of objects that together contain information about the design of one building. In this object model:

- A building object has information about the size and floor plan of the building as a whole.
- Floor objects contain information about the different floors of the building.
- A separate room object contains information about each individual room.
- Additional objects contain information about items in a room such as doors, windows, fireplaces, built-in appliances, plumbing fixtures, and so on.
- The building object is root of the composite object; it has links to its component floor objects, which have links to their room objects, which have links to objects for their doors, windows, and other features.

Although a composite object can be constructed using any combination of linking mechanisms, associations can facilitate deletion and locking operations on all component objects in the graph. See "Propagating Operations" on page 148.

Persistent Collections

A *persistent collection* is an aggregate persistent object that contains a variable number of elements. An element of a persistent collection can be any kind of persistent object—a basic object of an application-defined class or an object of an Objectivity/C++ persistence-capable class, for example, a container or another persistent collection. The various elements of a given persistent collection can be objects of the same class, or of a number of different classes.

The persistent collection is the source object of links to the objects it contains; it creates a link when an object is added to the collection and it deletes a link when an object is removed from the collection. Because of its relationship to its element persistent objects, a persistent collection can be viewed as the root object of a simple object graph—a flat tree structure with all leaf nodes directly below the root node.

For example, a company's human resources application might need to deal with the employees who are eligible for promotion. A persistent-collection object could represent this aggregate of employees objects. At each yearly performance review, the collection would contain a link to each employee who is currently eligible for promotion.

Objectivity/C++ provides various classes from which you can create persistent collections. Among these classes are classes for *scalable collections*, which can increase in size with minimal performance degradation, and *nonscalable collections*, which cannot.

All scalable-collection classes are derived from `ooCollection`; they are defined in the `ooCollections.h` header file. Some represent collections of persistent objects:

- `ooTreeList` represents lists of persistent objects.
- `ooHashSet` represents unordered sets of persistent objects.
- `ooTreeSet` represents sorted sets of persistent objects.

Other scalable-collection classes represent object maps—that is, collections of key-value pairs in which both keys and the values are persistent objects:

- `ooHashMap` represents unordered object maps.
- `ooTreeMap` represents sorted object maps.

The nonscalable persistence-capable collection class `ooMap` represents *name maps*—that is, collections of key-value pairs in which the keys are strings (or names) and the values are persistent objects. The class `ooMap` is defined in the `ooMap.h` header file.

Chapter 11, "Persistent Collections," describes the various persistent-collection classes.

Grouping Persistent Objects to Limit Search

You can group persistent objects according to the kinds of lookup operations you expect. Typically, you group the objects that are relevant to a particular task or operation that you expect applications to perform. Doing so enables an application performing a task to focus its search on the group of relevant objects instead of searching the entire federated database.

The search itself can look either for a single object within the group or for all objects in the group.

- If the application typically performs a given task for one particular object selected from the group, each object that is relevant to the task can be given a unique key that can be used to find that individual object among all relevant objects. The key can be a name, a numeric index, or another persistent object.
- If the application typically performs a given task for each relevant object in the group, individual keys are not necessary. The relevant objects can be grouped in some way so that an application can initialize an iterator to find all objects in the group. (Chapter 14, “Iterators,” describes iterators of various kinds.)

Persistent objects can be grouped by the storage hierarchy, by persistent collections, and by name scopes. These various kinds of grouping can be used independently or combined. They can be combined to form different levels in a single hierarchical organization of objects. Alternatively, they can be combined to provide a number of independent organizations that define different groupings over the same objects.

Within the groups defined by name scopes and some kinds of persistent collections, individual objects can be given unique keys. Chapter 16, “Individual Lookup of Persistent Objects,” describes how to assign keys to the persistent objects within these groups and how to look up an object by its key.

Chapter 17, “Group Lookup of Persistent Objects,” describes how to create and look up groups of persistent objects that need not have unique keys.

Grouping in the Storage Hierarchy

The Objectivity / DB storage hierarchy provides two levels of grouping within the federate database. If only one level of organization is needed, the objects relevant to different tasks can be stored in different databases. For example, a nation-wide company might run the same application in each of its regional offices. If each office processes objects relevant to its local region, the federated database could be organized with objects for each region in a different database. This would allow an application to scan the local region’s database for objects instead of having to scan the entire federated database, searching for those objects that are relevant to the local region.

If a second level of grouping is needed, the objects in a given database can be subdivided and grouped by container. In the preceding example, different containers within each regional database could contain objects relevant to different tasks.

Grouping in Persistent Collections

Persistent collections provide another mechanism for grouping objects. The objects in a particular group may be stored in different storage objects. The composition of a persistent collection can be fixed or can vary over time. For example, an interactive application might allow the user to group objects together in various ways and later perform some operation on each different group.

Grouping in Name Scopes

A *name scope* is a group of objects that have been given unique names within the scope of a particular Objectivity / C++ object, called the *scope object*. The objects in a given name scope need not reside in the same container or in the same database.

Different name scopes can be used to group different objects. For example, an architectural design application might need to look up both individual buildings and individual architects. It could name buildings in the scope of one object and name architects in the scope of a different object. This would allow any name lookup to search only the relevant names instead of having to search through names of both buildings and architects.

Alternatively, a given persistent object can be named in two or more different name scopes. For example, a company's federated database might include one name scope for sales representatives. Any employee who is the sales representative for a client company could be in that name scope identified by its client's name or account number. The same federated database might have another name scope for employees who are division managers. Any employee who is the manager for a division could be in this second name scope identified by the name of the division. An employee who is both a sales representative and a division manager would be in both name scopes.

Assigning Basic Objects to Containers

Part of organizing a federated database is to develop a plan for *clustering* the persistent objects in it. Clustering is the process of specifying where to physically store each new persistent object you create. Clustering a container assigns it to a database; clustering a basic object assigns it to a container. The organization of

basic objects into containers can profoundly affect the concurrency and runtime efficiency of the applications that use the objects, and storage requirements of the federated database.

No matter how simple your plan for clustering basic objects, you should isolate the implementation of that plan within your application framework. That way, you can tune your plan easily as experience reveals opportunities for improvement. If at some point you determine that you need to reassign 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.

Planning for Concurrent Access

For many federated databases, concurrency is the most important consideration. If multiple applications may access particular objects concurrently, you must identify:

- Objects that are common resources that many applications may need to use simultaneously
- Groups of objects that will most likely be accessed at the same time
- Objects that are likely to be modified frequently

If objects are to be accessed by various concurrent users, you should consider creating different containers for objects with different usage profiles. For example, suppose that a number of architects simultaneously run an application for developing the designs of buildings to be constructed. The objects relevant to each building could be clustered together in a separate container. This organization allows all objects relevant to the design of a particular building to be read more quickly than if the various component objects were distributed in different containers. Furthermore, an application that modifies the design of a building would need an update lock on only one container; a different application could simultaneously modify the design of a different building (stored in a different container). In contrast, if all room objects were stored in one container and all floor objects were stored in a different container, an application modifying one design would need update locks on both those containers, preventing other applications from modifying other designs simultaneously.

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.
- Consider assigning a large number of objects to the same container only if these objects will be updated infrequently.

- Consider assigning objects to the default container of a database only if these objects will be updated infrequently.
- Distribute objects that require frequent update among as many containers as reasonably possible.
- Isolate shared resources from objects that are frequently updated.
- Use multiple readers, one writer (MROW) concurrent access policy for transactions that lock containers for long periods of time.

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

Shared Resources

When a persistent collection groups objects to limit search, it is a shared resources—that is, various applications can use it concurrently for finding objects. Objects of application-defined classes may also represent shared resources that various applications can use concurrently, for example, an assembly line or loading dock in a manufacturing company.

If your federated database contains 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 elements to a collection, or to 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).

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.

Figure 7-2 illustrates a database in which objects are separated into read-intensive and update-intensive containers.

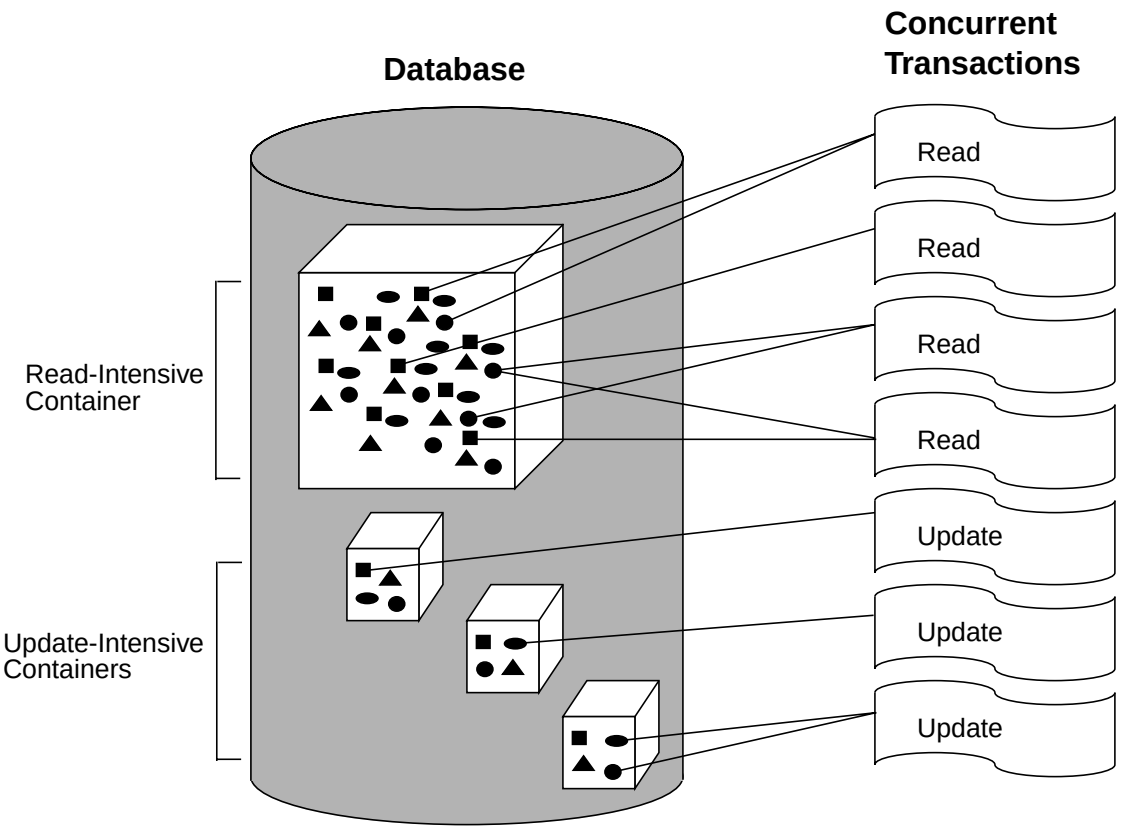


Figure 7-2 Read-Intensive and Update-Intensive Containers

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 fixed 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 all applications that read from the container use 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 are locked when a transaction updates it. In the case of highly volatile objects, you can consider isolating 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 stored with other signed contracts.

Such applications can store young objects in update-intensive containers. Figure 7-3 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.

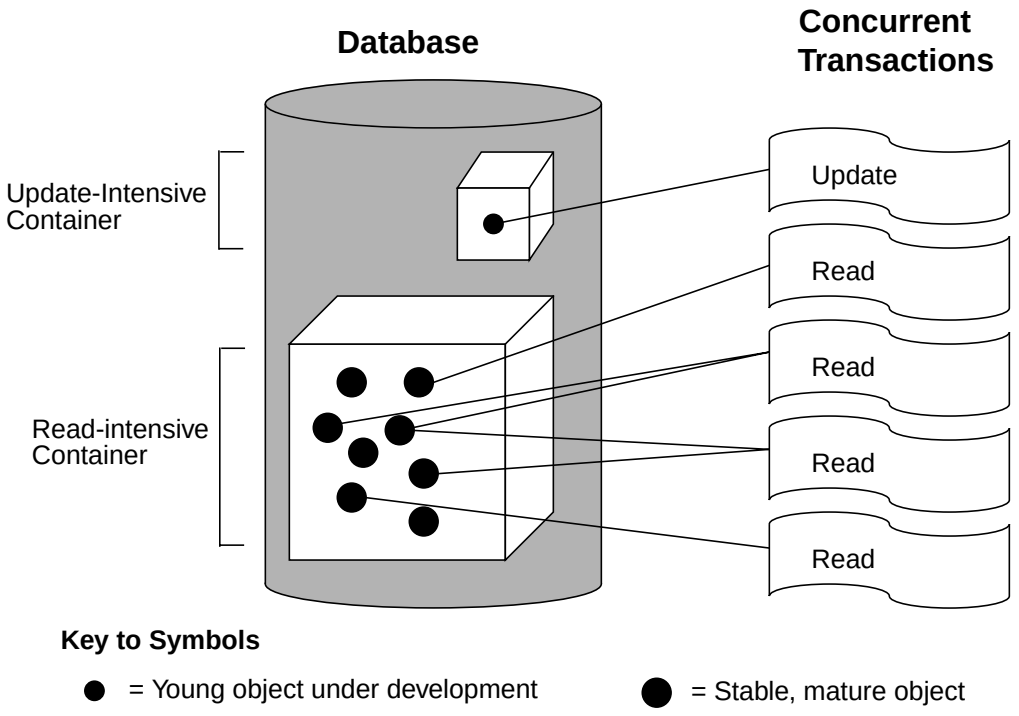


Figure 7-3 Young-Object and Mature-Object Containers

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 Basic Object” on page 201 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, 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 transactions, which ensures them read access to the mature contracts while another application has write access to the developing young contract.

Figure 7-4 illustrates a database in which objects under development are assigned to containers using a round-robin approach.

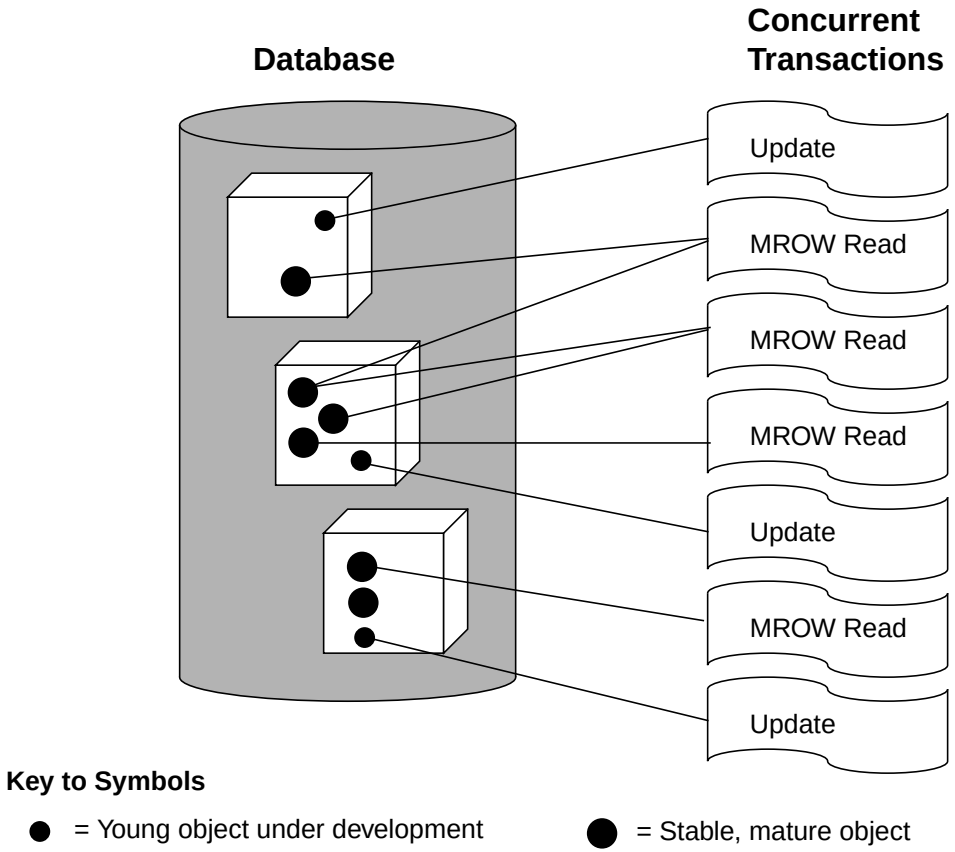


Figure 7-4 Round-Robin Assignment for Developing Objects

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 a total (`T`) of 40 containers (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 using the following reasoning:

- A given user competes with 19 other users:

  ```
  competingUsers = P - 1
  ```

- Competing users have locked 38 containers:

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

- That leaves just 2 containers available:

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

- Therefore, the odds of either desired container being available are 2 in 40 or 0.05:

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

- Because the user comes up against these odds twice—once for each desired container—the odds must be squared, resulting in 0.0025 as the odds that both containers are available:

  ```
  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 29.)

- Page map

  Each container maintains a table that maps logical object identifiers to physical device locations. This page map enables the container to quickly locate the object with a particular object identifier. 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 / DB. 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 transaction, 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 name 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 logical pages. (You can control the number of logical 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 objects that you know will eventually 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 when the temporary objects are deleted. Devoting a container to the temporary objects also gives you the option of simply deleting the entire container when the design is approved.

In deciding to devote a container to temporary objects, however, you are trading off runtime performance to meet secondary storage requirements. You may instead prefer to reduce the performance overhead by clustering the temporary objects with their design.

Persistence-Capable Classes

As an application developer, you may be involved in creating the logical model for the federated database. If so, you use the separately purchased option Objectivity/C++ Data Definition Language (Objectivity/DDL) to define the persistence-capable classes in your logical model. Alternatively, you may be building applications to access a federated database whose logical model was developed by a separate team of developers. In that case, your applications use definitions of persistence-capable classes that the modeling team created using Objectivity/DDL. The Objectivity/C++ Data Definition Language book contains a complete discussion of the process of defining persistence-capable classes. For convenience, this section summarizes the aspects of that process that you should understand before developing applications that use persistence-capable classes.

An application needs a persistence-capable class for each kind of object that is to be saved persistently. Most such objects are basic objects. However, an application that needs to save application-specific data for a container can include a persistence-capable container class whose data members contain the desired data.

A persistence-capable class derives from the base class `ooObj` and may have any number of application-defined *attributes* and *associations*:

- The attributes of a class constitute the data to be saved persistently for each object of the class. Attributes are implemented as standard C++ data members; some restrictions apply to the types of data that can be stored persistently. See“Attributes” on page 141.
- The associations of a class describe how an instance of that class can be related to instances of a specified class. Associations provide a mechanism for linking persistent objects together; they support functionality that is not available when attributes are used to link the objects. “Associations” on page 145 describes this additional functionality.

Attributes

Most nonstatic data members of a persistence-capable class correspond to attributes of the class—that is, they contain data that is saved persistently. The Objectivity / DB object model supports attributes of the following data types:

- Objectivity / C++ primitive types for integers, floating-point numbers, Booleans, and characters
- C++ enumerations and numeric types
- Object-reference types
- Embedded-class types in which the embedded class (or structure) is non-persistence-capable

In addition, a persistence-capable class can contain pointer data members for transient data.

For a detailed description of the data types that can be used for members of a persistence-capable class, see the Objectivity / C++ Data Definition Language book. The remainder of this section introduces various Objectivity / C++ classes that can be used for attributes.

Arrays of Values

An attribute can contain either a single value or an array of values. A fixed-size array is declared using the standard syntax. In addition, Objectivity / C++ defines array classes that can be used for attributes.

A *variable-size array* or *VArray* is similar to a one-dimensional C++ array, except that its size can change dynamically. The most commonly used kind of VArray is the *standard VArray*, which can be embedded in a persistent object. Thus, a persistence-capable class may define or inherit an embedded-class attribute whose type is a standard VArray class.

Standard VArray classes are created from the ooVArrayT<*element_type*> template; *element_type* is the type of elements in the array. For example, ooVArrayT<int32> is a class of standard VArrays whose elements are 32-bit signed integers.

NOTE Although Objectivity for Java and Objectivity / Smalltalk do not support embedded-class attributes, both are able to convert VArray attributes into attributes of the appropriate Java or Smalltalk array class. However, neither of these programming interfaces support fixed-size arrays. If you need to interoperate with applications written in other languages, you should use VArrays instead of C++ fixed-size arrays.

Objectivity / C++ also includes persistence-capable Java-compatibility classes for arrays. These classes are used primarily for interoperability with Java applications that have added class descriptions to the schema. A Java array attribute is described in the schema as an object reference to a Java-compatibility array of the appropriate class. Each Java array class is a wrapper for a VArray; it has a member function for obtaining that VArray.

Java Array of Class	Is a Wrapper for VArray of Class
oojArrayOfBoolean	ooVArrayt<uint8>
oojArrayOfCharacter	ooVArrayt<uint16>
oojArrayOfInt8	ooVArrayt<int8>
oojArrayOfInt16	ooVArrayt<int16>
oojArrayOfInt32	ooVArrayt<int32>
oojArrayOfInt64	ooVArrayt<int64>
oojArrayOfFloat	ooVArrayt<float32>
oojArrayOfDouble	ooVArrayt<float64>
oojArrayOfObject	ooVArrayt<ooRef(ooObj)>

As the table shows, there are Java array classes for arrays of primitive values and object references. A Java string array is described in the schema as an object reference to an array of class oojArrayOfObject. The object references in that array are references to the persistence-capable class oojString.

The Java-array classes and the class oojString are defined in the javaBuiltins.h header file.

See Chapter 12, “Variable-Size Arrays,” for information about using the various array classes. See Chapter 13, “Objectivity / C++ Strings,” for information about using the class `oojString`.

String Attributes

A string attribute can be implemented as a fixed-size character array or as an embedded-class attribute whose type is one of the Objectivity / C++ string classes. Note that a simple C++ `char *` string cannot be used as an attribute. Memory pointers can be used only for transient data because they are not valid outside the process that sets them.

The primary Objectivity / C++ string classes are:

- The class `ooVString`, which represents *variable-size strings* of ASCII characters. You can convert transparently between a variable-size string and a `const char *` string.
- The parameterized *optimized-string classes*. The optimized-string class `ooString(N)` is a variable-size string optimized to store strings that are generally less than *N* characters long. For example, the class `ooString(20)` is optimized to store strings that are generally less than 20 characters long. You can convert transparently between an optimized string and a `const char *` string.

NOTE Although Objectivity for Java and Objectivity / Smalltalk do not support embedded-class attributes, both are able to convert `ooVString` attributes into attributes of the appropriate Java or Smalltalk string class. However, Java and Smalltalk applications cannot access classes with optimized-string attributes or fixed-size character arrays.

Objectivity / C++ also includes the string class `ooUtf8String`, which represents variable-size Unicode strings. This class is defined in the `javaBuiltins.h` header file. It is used primarily for interoperability with Java applications that added class descriptions to the schema. A Java string attribute is described in the schema as an embedded-class attribute of type `ooUtf8String`.

See Chapter 13, “Objectivity / C++ Strings,” for information about using string classes.

Date and Time Attributes

If your persistence-capable classes need to store information about dates or times, they can use an embedded-class attribute whose type is the appropriate date/time class.

Objectivity/C++ provides non-persistence-capable classes that represent information about date and time, as described in the ODMG standard.

Class	Description
`d_Date`	Calendar date; no representation of null
`d_Time`	Clock time; no representation of null
`d_Timestamp`	Instant in time to the nearest millisecond; no representation of null
`d_Interval`	The duration of elapsed time between two points in time

The ODMG date and time classes are defined in the `ooTime.h` header file.

NOTE Because Objectivity for Java and Objectivity/Smalltalk do not support embedded-class attributes, an application that uses these date/time classes cannot interoperate with Java or Smalltalk applications.

Objectivity/C++ also includes persistence-capable Java-compatibility classes for date and time data. These classes are used primarily for interoperability with Java applications that added class descriptions to the schema. A Java date, time, or timestamp attribute is described in the schema as an object reference to the corresponding Java date/time class.

Java Date/Time Class	Description
`oojDate`	Date: Instant in time with millisecond precision
`oojTime`	Time: Clock time with millisecond precision
`oojTimestamp`	Timestamp: Instant in time with nanosecond precision

The Java date and time classes are defined in the `javaBuiltins.h` header file.

The Objectivity/SQL++ book describes additional date/time classes that are included with that separately purchased product.

Reference Attributes

A *reference attribute* can contain one or more object references. The type of a reference attribute can be:

- An object-reference class
- A fixed-size array of object references
- A VArray of object references

A reference attribute can contain either standard or short object references:

- A *standard object reference* uniquely identifies the referenced persistent object in the entire federated database. Standard object references are instances of the parameterized classes `ooRef(`*`className`*`)`.
- A *short object reference* uniquely identifies the referenced basic object in a given container. Short object references are instances of the parameterized classes `ooShortRef(`*`className`*`)`.

The *`className`* parameter to an object-reference class is the name of a persistence-capable class, called the *referenced class*. A reference attribute serves to link an object of the class defining the attribute to an instance of the referenced class or one of its derived classes. In the case of a short object reference, the two objects linked by the attribute must be located in the same container. See "Linking With Reference Attributes" on page 314.

Associations

Objectivity / DB *associations* constitute a linking mechanism that provides a higher level of functionality than reference attributes. Objectivity / DB maintains the association links in the federated database and supports a variety of operations on linked objects, thus reducing the amount of work you have to do to perform those operations. The following sections describe the various characteristics and behavior of associations. This information will enable you to decide which links between persistent objects to model with associations and which with reference attributes.

You use Objectivity / DDL to define an association in some particular persistence-capable class, called the *source class* of the association. The association definition declares that a directional link with particular characteristics can exist from an instance of the source class to an instance of a *destination class*. The destination class can be any persistence-capable class, including the source class itself. See Chapter 3, "Defining Associations," in the Objectivity / C++ Data Definition Language book for information about how to define associations and how association links are stored in the federated database.

As new instances of the source and destination classes are created dynamically, actual links between instances can also be created dynamically. Your application uses member functions generated by the DDL processor to create and delete links

and to find destination objects. See “Linking With Associations” on page 317 for information on how to create and follow association links from a source object to its destination objects.

Association Directionality

An association is a directional link from a source object to a destination object. An application can use the association to find the destination object from the source object. The *directionality* of the association determines whether an inverse link exists that allows the application to find a source object from a destination object.

Syntax for defining an association’s directionality is given in “Basic Association Syntax” on page 68 of the Objectivity / C++ Data Definition Language book.

Unidirectional Associations

A *unidirectional association* links a source object to a destination object, but provides no mechanism for linking the destination object back to the source object. As a consequence, an application cannot find a source object from a destination object. For example, a car might have a unidirectional relationship to its manufacturer, allowing an application to find the manufacturer of any car.

Unidirectional associations correspond closely to reference attributes, or, in a standard C++ object model, to data members that use pointers to link objects. They require somewhat less overhead than bidirectional associations and so offer better performance.

Bidirectional Associations

Each *bidirectional association* has a corresponding *inverse association*; this pair of associations together provide bidirectional links between two objects. The source class of a bidirectional association is the destination class of its inverse association, and vice versa. For example, suppose each senior employee in a particular company teaches an apprentice the skills of the trade. A bidirectional link modeling the relationship between a teacher and apprentice could be represented by the following pair of bidirectional associations, each of which is the inverse of the other:

- The `apprentice` association could link a `SeniorEmployee` object to an `Apprentice` object.
- The `teacher` association could link an `Apprentice` object to a `SeniorEmployee` object.

Objectivity / DB operates on a pair of bidirectional associations in parallel; creating or deleting a link in one direction causes the simultaneous creation or deletion of the inverse link. For example, you would call a *single* member function to create both the `apprentice` association from a senior employee to an

apprentice and the inverse `teacher` association from the apprentice to the senior employee.

Bidirectional associations allow an application to use either of two linked objects as the starting point for finding the other. For example, an application could use the `apprentice` association of a senior employee to find that employee's apprentice, and it could use the `teacher` association of an apprentice to find that apprentice's teacher.

Bidirectional associations also enable Objectivity / DB to maintain referential integrity, so that deleting a destination object automatically deletes the source object's link to it, reducing the likelihood of dangling links. A *dangling link* is one that references a nonexistent object. In contrast, Objectivity / DB cannot prevent dangling links by reference attributes or unidirectional associations.

Association Cardinality

The *cardinality* of an association indicates the number of destination objects that can potentially be linked to a given source object: one or many. Objectivity / DB associations support four categories of cardinality:

- One-to-one—for example, if a car can have only one manufacturer, then a `Car` class could define a one-to-one association `madeBy` to a `Manufacturer` class.
- One-to-many—for example, if a company can have many employees, then a `Company` class could have a one-to-many association `employs` to an `Employee` class.
- Many-to-one—for example, if each of many employees can work for a single company, then an `Employee` class could have a many-to-one association `employedBy` to a `Company` class. This is the inverse of a one-to-many bidirectional association.
- Many-to-many—for example, if each of many patrons is allowed to use each of many libraries, a `Patron` class could have a many-to-many association `canUse` to a `Library` class, and the `Library` class could have an inverse many-to-many association `members` to the `Patron` class.

NOTE Many-to-one and many-to-many associations *must be* bidirectional.

The term *to-one association* means either a one-to-one association or a many-to-one association. The term *to-many association* means either a one-to-many association or a many-to-many association.

Syntax for defining an association's cardinality is given in "Basic Association Syntax" on page 68 of the Objectivity / C++ Data Definition Language book.

Propagating Operations

Associations can be defined so that delete or explicit lock operations will *propagate* along the link(s) from a source object to its destination object(s). An association definition specifies which operations should be propagated. Propagation along an association is optional, and the default behavior for both delete and lock operations is no propagation. Syntax for enabling propagation is given in “Requesting Propagation Operations” on page 71 of the Objectivity/C++ Data Definition Language book.

Delete propagation occurs when you delete a source object. See “Deleting a Persistent Object” on page 196. If the deleted object has any associations that propagate delete operations, the destination object(s) of those associations are also deleted.

Lock propagation occurs when you explicitly lock a source object. See “Explicitly Locking a Persistent Object” on page 110. If the locked object has any associations that propagate lock operations, the destination object(s) of those associations are also locked.

When a propagating operation is applied to an object, Objectivity/DB first identifies all objects that are affected (by identifying associations that are declared to propagate the operation). It then applies the operation to all affected objects in a single atomic operation. This approach guarantees that a propagating operation will eventually terminate, even though the propagation graph may contain cycles.

Propagation is particularly useful for associations that link the component objects of a composite object. An application can find a single “root object” of the composite and then follow links to find the other components. If the component objects are linked together by associations for which propagation is enabled, deleting the root object deletes the entire composite object; explicitly locking the root object locks all the component objects.

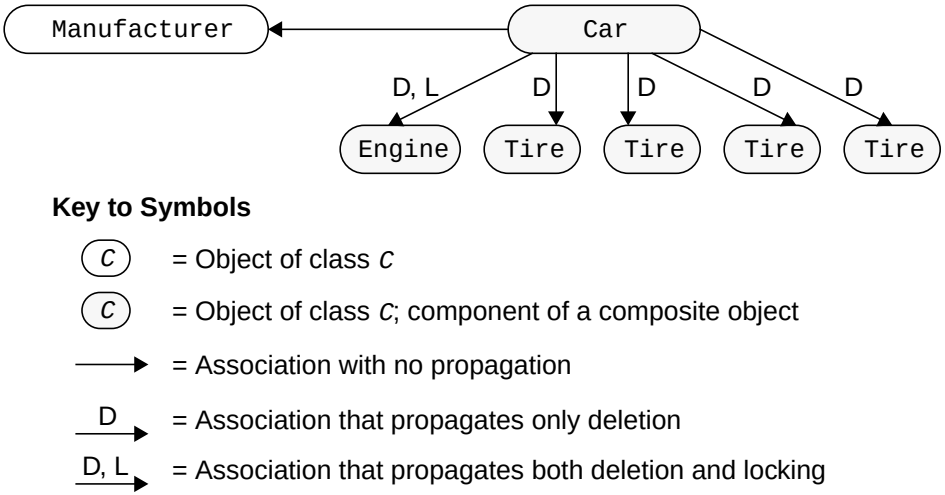
EXAMPLE An application represents automobiles as composite objects. The root of each composite automobile is an instance of the class `Car`. Associations link a `Car` object to the components of the automobile—namely, an `Engine` object and four `Tire` objects. Another association, representing the automobile’s manufacturer, links a `Car` object to a `Manufacturer` object. Because an automobile is a composite object, deleting the root `Car` object should delete all component objects; explicitly locking the `Car` should lock all components. On the other hand, deleting a `Car` should not delete its `Manufacturer` and locking a `Car` should not lock its `Manufacturer`.

Suppose the application always stores an automobile’s `Tire` components in the same container as the root `Car` object and always stores the `Engine` component in

a different container. The class `Car` would define associations with the following propagation behavior:

- The `tires` association from `Car` to `Tire` would propagate deletion but not locking. Lock propagation is unnecessary for this association because a `Car` and its associated `Tire` objects are stored in the same container. Locking the `Car` locks its container, which effectively locks all related `Tire` objects.
- The `engine` association from `Car` to `Engine` would propagate both deletion and locking.
- The association from `Car` to `Manufacturer` would not propagate either deletion or locking.

The following figure illustrates the associations that can exist from a `Car` object to various destination objects and the propagation behavior of each association.



The propagation behavior of a bidirectional association affects only that association, not its inverse. For example, assume the `tires` association from a car to its tires is made bidirectional with the inverse `inCar` association from `Tire` to the `Car`. When a car's tires are replaced, the old `Tire` objects are deleted, but the `Car` object should not be deleted. Therefore, the `inCar` association would not propagate deletion (although its inverse `tires` association would propagate deletion).

Copying and Versioning Behavior

An association definition specifies how a link is handled when a copy or new version of a source object is created. See “Copying a Basic Object” on page 197 and Chapter 20, “Versioning Basic Objects”.

The possible copying or versioning behaviors are:

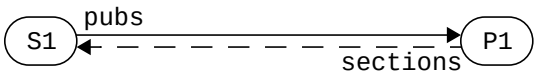
- The link is deleted from the copy or new version, and left with the original source object. After the operation, the original source object is linked to the destination object(s), but the new object is not. This is the default behavior.
- The link is moved from the original source object to the copy or new version. After the operation, the new object is linked to the destination object(s), but the original source object is not.
- The link is copied from the original source object to the copy or new version. After the operation, both the original source object and the new object are linked to the same destination object(s).

For information about specifying one of these actions for an association, see “Specifying Object Copying and Versioning Behavior” on page 72 of the Objectivity/C++ Data Definition Language book.

When a source object with a bidirectional association is copied or versioned, that association’s copy behavior affects both that association’s link and its inverse: both links are deleted, moved, copied. Note, however, that the inverse association may define a different copy or versioning behavior.

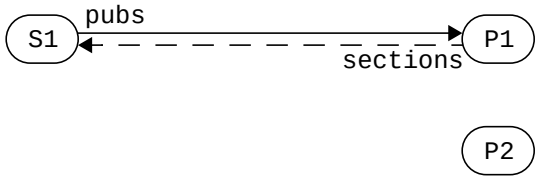
EXAMPLE This example illustrates a pair of inverse bidirectional associations with different copy behavior. The application models a distributor that ships various kinds of publications (books, magazines, newspapers, and so on) for display in various sections (sports, fiction, business, and so on) of assorted retail outlets (bookstores, newsstands, and so on).

Because a given publication may be displayed in multiple sections, and a given section may display multiple publications, a pair of many-to-many bidirectional associations models the relationship between the `Section` and `Publication` classes. The `pubs` association links a section to the publications that it displays; its inverse `sections` association links a publication to the sections in which it is displayed. The links between a section `S1` and a publication `P1` in that section can be illustrated as follows:



The application uses containers to model retail outlets. When a retail outlet changes management, existing publications are copied into a new container so they can be reorganized into new sections. Therefore, when a publication is

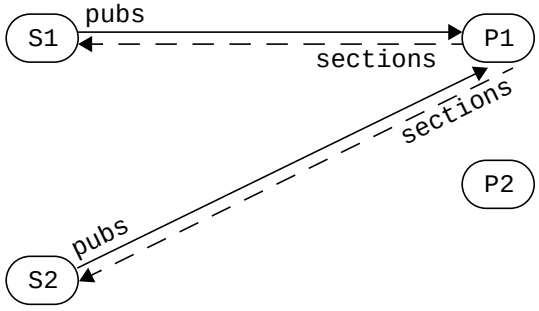
copied, any existing associations to the sections that display it are deleted from the new publication. After copying publication P1 to create a new publication P2, the following links would exist:



Copy Behavior:

Delete the sections association from the new publication P2 (and also delete its inverse association)

When a retail outlet is successful, it adds a mail-order service for certain sections. This service is modeled as a new container with copies of the sections that are available by mail order. Any publication displayed in a section should be available in the corresponding mail-order section. Therefore, when a section is copied, both the original and the copied sections have associations to the same publications. After copying section S1 to create a new mail-order section S2, the following links would exist:



Copy Behavior:

Copy S2's pubs association; that is, create a link from the new section S2 to P1 (and also create its inverse association)

Association Storage

Associations can be defined with different storage properties:

- A non-inline association stores links in the source object's *system default association array*.
- A to-one inline association stores links in the source object's association data member.
- A to-many inline association stores links in an association-specific array.

Associations are non-inline by default. Syntax for specifying inline associations is given in "Inline Association Syntax" on page 70 of the Objectivity/C++ Data Definition Language book.

The storage properties for the various associations defined by a given source class may limit the total number of links for any single source object of that class.

In particular, all links for non-inline associations are stored in the source object's system default association array. This array is opened whenever associations are added to it or deleted from it, or when a destination object is found through it. When open, the array must fit into available swap space, so there is an implied limit on the number of associations that a source object can have.

See "Storage Requirements for Associations" in Chapter 3 of the Objectivity/C++ Data Definition Language book for information about estimating the size of an object's system default association array.

Member Functions

An application-defined persistence-capable class can have any member functions that the application requires. Note, however, that Objectivity/DB saves only data persistently, not member functions. 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 attributes and associations and implementations for its application-defined member functions.

Defining Persistence-Capable Classes

All application-defined persistence-capable classes must be declared using Objectivity/C++ Data Definition Language (DDL) in a DDL file. The file must be processed by the DDL processor. Processing a DDL file generates the C++ definitions of the classes and adds descriptions of the classes to the federated-database schema. Processing a DDL file also generates definitions for a handle class, an object-reference class, and an object-iterator class for every application-defined persistence-capable class; it defines a short object-reference class for every application-defined basic-object class.

Your application must include the header files produced by the DDL processor before it can create persistent objects.

If a DDL file uses collection, date/time, or Java compatibility classes, it must include (directly or indirectly) the corresponding Objectivity/C++ header file.

To Use	You Must Include
Name-map class `ooMap`	`ooMap.h`
Scalable-collection classes	`ooCollections.h`
Date and time classes	`ooTime.h`
Java compatibility classes	`javaBuiltins.h`

If one DDL file uses an application-defined persistence-capable class *appClass* that is declared in a second DDL file, the first file must include the appropriate DDL directives to ensure that the generated C++ definition of *appClass* is available to it.

See the Objectivity/C++ Data Definition Language book for instructions on defining persistence-capable classes, writing DDL files, and using the DDL processor.

8

Storage Objects

Objectivity / DB federated databases, databases, and containers are storage objects. This chapter describes:

- General information about storage objects
- Federated database tasks, such as creating, opening, finding, and deleting
- Database tasks, such as creating, finding, opening, and deleting
- Container tasks, such as creating, finding, opening, and deleting

This chapter discusses the operations that typical applications perform on storage objects. Chapter 26, “Writing Administration Tools,” describes additional administration operations.

You can increase the robustness of your application by making use of Objectivity / C++ features that support partitioning a federated database into units that can operate independently and replicating an individual database within those units. These features are transparently accessible through the Objectivity / C++ programming interface, but require two additional Objectivity products, Objectivity / DB Fault Tolerant Option and Objectivity / DB Data Replication Option, to gain access to the functionality within Objectivity / DB. Further information on how to use these features can be found in Chapter 27, “Autonomous Partitions,” and Chapter 28, “Database Images”.

Understanding Storage Objects

Storage objects serve to group other objects to achieve performance, space utilization, and concurrency requirements. The three kinds of storage objects correspond to three levels of grouping in the Objectivity / DB storage hierarchy.

Storage Hierarchy

A *federated database* is the highest level of the storage hierarchy; it is the unit of administrative control for Objectivity / DB. A federated database:

- Maintains the object model (or schema) that describes all the objects stored in the databases.
- Maintains configuration information (where Objectivity / DB files physically reside). All recovery and backup operations are performed at this level.
- Consists of system-created databases and databases created by applications.

Databases form the second level of the storage hierarchy. 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.

A database consists of system-created containers and containers created by applications.

Containers form the third level of the storage hierarchy; they 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 smallest 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 Storage Objects

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

1. Obtain a handle, object reference, or object iterator that references the storage object.

   Operations on a storage object (other than creation and deletion) are invoked indirectly through a reference to that object. An application normally obtains a handle to the federated database or to an existing database by opening it; references to containers can be obtained in a variety of ways.

   For more information, see Chapter 10, “Handles and Object References”.

2. Obtain a lock on the storage object for the desired level of access.

   An operation on a storage object implicitly requests the locks it needs at the point at which the lock is required. For example, if you scan a database, your

application obtains the appropriate locks on the database and all the containers in the database.

If your application cannot obtain a lock due to a conflict with an existing lock, an error is signaled. You can explicitly reserve locks in advance if you wish to avoid such errors. For more information, see Chapter 6, “Locking and Concurrency”.

Any operation on a storage object (including creating or finding a storage object or obtaining a lock) must be performed within a transaction. Changes to a storage object are visible only within the transaction 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 by aborting the transaction.

In general, you can perform the following operations on storage objects:

- Read a property of the storage object, such as the system name.
- Create an object within the storage object.
- Find all the objects contained within the storage object.
- Scan the storage object for all objects below it in the storage hierarchy.
- Name a persistent object within a scope defined by the storage object and find a persistent object by its name.

Federated Databases

A federated database is the highest level in the Objectivity / DB storage hierarchy; each federated database logically contains one or more databases. An Objectivity / C++ application represents a federated database as an object of the class `ooFDObj` and works with that object through a handle.

Physically, an Objectivity / DB federated database is maintained in a *system-database file*, which stores the schema for the federated database, a catalog of all the databases, and the scope names of persistent objects named in the scope of the federated database. Configuration information for the federated database is maintained in a second file (the *boot file*), along with various other properties, including:

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

See the Objectivity / DB administration book for more information about a federated database's files and properties.

Any operation that creates, deletes, or modifies the various files used by the federated database requires the normal file-system permissions.

- If a tool performs the operation, the user account under which the tool is run must have the appropriate permissions.
- If an application performs the operation:
  - If the files are being accessed remotely by the Advanced Multithreaded Server (AMS), the user account under which AMS is running needs the appropriate permissions.
  - Otherwise, the user account running the application needs the appropriate permissions.

  For information about AMS, see the Objectivity / DB administration book.

Creating a Federated Database

A federated database can be created only with the `oonewfd` administration tool, which is described in the Objectivity / DB administration book. Creating a federated database creates its boot file and its system database file. The arguments to the `oonewfd` tool specify the host, directory, and file names for these files.

Applications use the pathname of the boot file to find the federated database; the simple file name of the boot file serves as the federated database's system name.

Opening a Federated Database

An application must open the federated database to be accessed at the beginning of every transaction. *Only one* federated database can be open in a process, although it may be opened and closed multiple times. In the first transaction of the application, opening the federated database initializes Objectivity / DB with schema information and storage page size.

You open a federated database by calling the `open` member function on a federated-database handle. You identify the federated database by specifying its boot file. The boot file may, but need not, reside in the directory where you will run your application, so you must specify its location using an appropriate pathname (for information about path and filenames, see the Objectivity / DB administration book). The `open` member function allows you to omit the pathname parameter and instead use the value of the `OO_FD_BOOT` environment variable.

You specify the *open mode* for the federated database as a parameter to the `open` member function. The open mode determines the transaction's level of access to persistent objects. An open mode of `oocRead` (the default) identifies the

transaction as a read transaction and an open mode of `oocUpdate` identifies the transaction an update transaction.

You can optionally check whether a federated database exists before attempting to open it by calling the `exist` member function on a federated-database handle.

EXAMPLE This example opens a federated database whose boot file pathname is `/net/design/ECAD`.

```
// Application code file
#include "myClasses.h"
...
ooTrans transaction;     // Define a transaction object
ooHandle(ooFDObj) fdH;   // Define a federated-db handle

transaction.start();     // Start a transaction
fdH.open(                // Open the federated database
   "/net/design/ECAD",      // Pathname for boot file
   oocUpdate);              // Open mode
...
```

This call to open:

- Initializes the handle `fdH` to reference the specified federated database.
- Identifies the transaction as an update transaction.
- Implicitly places an intention update lock on the federated database, which allows other transactions to concurrently open it, but prevents any other transaction from locking it for read or update.
- Opens the system-database file, provided that appropriate access permissions are set on it.

Enabling Automatic Recovery

When you open a federated database for the first time in an application, you should enable automatic recovery from local C++ application failures. Doing so causes Objectivity/DB to automatically roll back any incomplete *local* transactions against the federated database (transactions that were started by other applications running on the same host). See the Objectivity/DB administration book for more information about automatic recovery. See “Creating a Recovery Application” on page 534 for information about writing a recovery application.

To enable automatic recovery, you invoke the open member function with the *`recover`* parameter set to `oocTrue`. For performance reasons, you should do this only one time per application.

Promoting the Open Mode

Within a read transaction, you can promote the open mode of the federated database to update by calling `open` on the federated-database handle with the *`openMode`* parameter set to `oocUpdate`. You do not need to close the federated database first. Note that you may not demote the open mode from update to read.

You can find out the current open mode by calling the `openMode` member function on the federated-database handle.

Finding a Federated Database

The primary way to find a federated database is by opening it, which sets the federated database's identifier in the handle. Because only one federated database can be open in a process, no other federated databases can be found. You can, however, find the same federated database through additional handles in any of the following ways:

- Find the federated database from a particular database. To do this, you call the `containedIn` member function on a handle to the database.
- Find the federated database from a particular autonomous partition. To do this, you call the `containedIn` member function on a handle to the partition.

Administering a Federated Database

You can get the various attributes of a federated database, such as its lock server host, identifier, system name, and page size. Furthermore, you can change certain federated-database attributes, such as the lock server host or the boot file location. See “Federated Database Administration” on page 527.

Closing a Federated Database

You can optionally close a federated database by calling the `close` member function on a federated-database handle. Closing a federated database explicitly closes all persistent objects that are open at this point in the current transaction. Note, however, that locks on the closed objects are retained until the transaction commits or aborts.

You cannot open a different federated database within a process even if you have closed the first one. That is, you can only open one federated database per process, although you can open and close the same federated database as often as you like.

Deleting a Federated Database

A federated database can be deleted only with administration tool `oodeletefd`. Deleting a federated database deletes all relevant files from the file system:

- The federated database's boot file and system database file
- The database file for each database in the federated database
- All files for all autonomous partitions of the federated database

If there are any outstanding journal files for the federated database, these must be cleaned up before the federated database can be deleted. You can clean up the journal files using the `oocleanup` administrative tool.

See the Objectivity / DB administration book for a discussion of journal files and a description of the administrative tools.

Databases

A database is the second highest level in the Objectivity / DB storage hierarchy. An Objectivity / C++ application represents a database as an object of the class `ooDBObj` and works with that object through a handle.

When a database is created, its default container is also created. When a basic object is clustered near a database, it is stored in the default container of that database. The default container is also used by the scope-naming mechanism when persistent objects are named in the scope of the database. You normally add one or more application-specific containers to a database.

A database is physically maintained in a *database file*. This file contains a catalog of all the application-specific containers, and all the containers and basic objects stored in the database.

Each database belongs to exactly one federated database and is listed in that federated database's catalog by its system name. The system name must be unique within the federated database. A system name does not normally change for the lifetime of the database, although it is possible to use administration tools to create a duplicate database with a new name; see the Objectivity / DB administration book. When a database is deleted, its system name is also removed from the federated database catalog.

Any operation that creates, deletes, or modifies a database file requires the normal file-system permissions.

- If a tool performs the operation, the user account under which the tool is run must have the appropriate permissions.

- If an application performs the operation:
  - If the file is being accessed remotely by the Advanced Multithreaded Server (AMS), the user account under which AMS is running needs the appropriate permissions.
  - Otherwise, the user account running the application needs the appropriate permissions.

  For information about AMS, see the Objectivity / DB administration book.

Unit of Distribution

Because databases are maintained as files on the host file system, they provide a convenient way to administer related persistent objects at a particular physical location. That is, you can create multiple databases 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.
- Subdivide 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 basic objects is divided among databases within a federation, and one or more name maps are used to associate the databases with various search keys. Within each database, the basic objects are subdivided among a set of containers, and one or more name maps are used to associate the containers with search keys. Assigning a basic object to a container consists of identifying the subdivision to which the object belongs and clustering the object in the corresponding container.

Creating a Database

You create a database in the currently open federated database by calling operator new on class ooDBObj within an update transaction. This operator is generally used as follows:

```
// Create a new database referenced by database handle dbH
ooHandle(ooDBObj) dbH = new ooDBObj(initializers);
```

The *initializers* you specify are the parameters to the ooDBObj constructor. At a minimum, you must specify a unique system name for the new database. You may optionally specify nondefault values for the:

- Initial number of logical pages to allocate for the default container. A *logical page* is a storage page that contains either one or more small objects or the header information for a large object. Small and large objects are described in "Cache Components" on page 72.

  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 and path of the directory where the database file is to be located. By default, the new database file is created in the same location as the system-database file of the federated database.
- Weight of the first database image if Objectivity / DB Data Replication Option (DRO) is being used. If Objectivity / DRO is not being used, you must specify 1.
- 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.

The newly created database is automatically opened for update, and is made permanent on disk when you commit or checkpoint the transaction. You work with the new database through the database handle to which you assign the result of operator new.

You can optionally specify one or two clustering directives after the keyword new. The first directive is a handle to the currently open federated database; if Objectivity / DB Fault Tolerant Option is being used, the second directive allows you to specify an autonomous partition.

EXAMPLE This example creates a database named PartsDB in the same location as the system-database file.

```
// Application code file
#include "myClasses.h"
...
ooTrans transaction;                  // Transaction object
ooHandle(ooFDObj) fdH;                // Federated DB handle
ooHandle(ooDBObj) dbH;                // Database handle

transaction.start();                  // Start a transaction
fdH.open(                             // Open the federated DB
   "Inventory",                          // Boot file name
   oocUpdate);                           // Open mode
dbH = new(fdH) ooDBObj("PartsDB");    // Create a database
transaction.commit();                 // Commit the transaction
```

A database can also be created with the oonewdb administration tool (see the Objectivity / DB administration book).

Checking Whether a Database Exists

You can test whether a particular database exists by calling the exist member function on a database handle, specifying a handle to the currently open federated database and the system name of the desired database. If the specified database is found, this member function returns oocTrue and also sets the database handle to reference the specified database; otherwise, oocFalse is returned.

Testing for a database's existence helps to avoid the errors that are signaled if you attempt to create a database with a nonunique system name or if you attempt to open a nonexistent database.

EXAMPLE This example opens a federated database with the system name `shapeExample`, and checks to see whether a database exists whose system name is `simpleShapes`. If not, it creates a database with that name. In either case, it sets the `dbH` handle to reference the database named `simpleShapes`.

```
// Application code file
#include "myClasses.h"
...
ooTrans transaction;         // Define a transaction object
ooHandle(ooFDObj) fdH;       // Define a federated db handle
ooHandle(ooDBObj) dbH;       // Define a database handle

// Start an update transaction and open the federated database
transaction.start();
fdH.open("shapeExample",oocUpdate);
if (!dbH.exist(fdH, "simpleShapes"))
   dbH = new(fdH) ooDBObj("simpleShapes");
transaction.commit();
```

Finding a Database

You can find a database in any of the following ways:

- Look up a database by its system name. To do this, you call the `exist` or `open` member function on a database handle. (You should use `exist` if you want to find a database without opening it.) If the specified database is found, the handle is set to reference it.
- Initialize an object iterator of class `ooItr(ooDBObj)` to find all databases in the federated database. You can do this in either of the following ways:
  - Call the `contains` member function on a handle to the federated database, passing the object iterator as a parameter.
  - Call the `scan` member function on the object iterator, passing a handle to the federated database as a parameter.

  After initializing the object iterator, you advance it through the iteration set by calling the iterator's `next` member function. See "Object Iterators" on page 293 for information about working with an object iterator.
- Find the database that contains a particular container. To do this, you call the `containedIn` member function on a handle to the container.

Opening a Database

Opening a database makes it available to an application. Opening a database:

- Locates and opens the database file, provided that appropriate access permissions are set on it.
- Places an intention lock on the database for the requested level of access (read or update).

Operations that Open Databases Implicitly

A database is implicitly opened for read or update by operations that access the database or its contents. If you have obtained a handle to a database (for example, by finding the database from one of its containers), operations such as the following will implicitly open the referenced database for the appropriate level of access:

- Reading a property of the database, such as the system name.
- Creating a container or basic object in it.
- Opening a container or basic object in the database. However, the reverse is not true—opening a database does not open its containers or basic objects.
- Initializing an object iterator to find all the containers in the database.
- Naming or looking up a persistent object in the scope defined by the database.

Creating a database leaves the new database open for update.

Explicitly Opening a Database

In general, you open a database explicitly only when:

- You want to obtain a handle to the database through which you can access your data. For example, at the beginning of a transaction, you might explicitly open a database so you can look up one of its containers by system name and then iterate over the basic objects in the container.
- You want to guarantee access to the database in advance—for example, before starting a complex operation. Opening a database ensures that the database file can be found.

To find and explicitly open a database, you specify the desired level of access as a parameter to the `open` member function on a database handle. By default, the database is opened for read access; you must pass the parameter value `oocUpdate` to open the database for update.

EXAMPLE This code fragment starts an update transaction that opens a federated database named ECAD. This transaction then opens the database named UPROC for read and the database named EPROM for update. Each open operation:

- Initializes a handle to reference the specified database.
- Implicitly places an appropriate intention lock on the database (read or update).
- Opens the database file, provided that appropriate access permissions are set on it.

```
// Application code file
#include "myClasses.h"
...
ooTrans transaction;            // Transaction object
ooHandle(ooFDObj) fdH;          // Federated database handle
ooHandle(ooDBObj) dbH1, dbH2;   // Database handles

// Start an update transaction and open the federated database
transaction.start();
fdH.open("ECAD", oocUpdate);
...
// Open the database named UPROC for read (the default access)
if (!dbH1.open(fdH, "UPROC")) {
   transaction.abort();
}
// Open the database named EPROM for update,
// creating it if it doesn't exist
if (!dbH2.exist(fdH, "EPROM")) {
   dbH2 = new(fdH) ooDBObj("EPROM");
}
else {
   dbH2.open(oocUpdate);
}
...
transaction.commit();
```

If you have obtained a handle to a database without opening it, you can open the database by calling the open member function on the database handle, specifying the desired access level. For example, call open(oocRead) to open the referenced database for read access.

NOTE Opening a database does not automatically secure access rights to the containers in it. To obtain access rights to all the containers in a database, you must explicitly open or lock each container.

Checking and Promoting the Level of Access

After you have obtained a handle to a database, you can determine the current level of access by calling the `openMode` member function.

If a database is open for read, and you want to secure update access to it, you can call the `update` member function on a handle to the database. This member function is equivalent to `open(oocUpdate)`.

Administering a Database

You can get the various attributes of a database, such as its system name and location, and you can change the location of the database file. See "Database Administration" on page 531.

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` member function on a handle to the database, specifying the parameter `oocTrue`.

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 or change its attributes for administrative purposes, you must change the database back to read-write. To do this, you call the `setReadOnly` member function specifying the parameter `oocFalse`.

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 attributes of individual images.

Closing a Database

An open database is closed automatically when the transaction in which it was opened commits or aborts.

Deleting a Database

An application can delete a database by calling the `ooDelete` global function. This function requires that you specify a handle to the database to be deleted. Alternatively, you can delete a database from the command line with the `oodeletedb` or `oodeletedbimage` administration tool (see the Objectivity / DB administration book).

WARNING Deleting a database cannot be undone. If an application deletes a database and then aborts the transaction, the database remains deleted.

Deleting a database:

- Deletes all of the containers and basic objects in the database. Note that:
  - All associations from these objects to their destination objects are also deleted. Bidirectional associations are updated to maintain referential integrity; however, you must clean up any dangling references resulting from unidirectional associations to deleted destination objects. See “Associations” on page 145.
  - If any associations on these objects have delete propagation enabled, the associated destination objects are deleted as well. See “Propagating Operations” on page 148.
- Deletes the database’s system name from the federated-database catalog.
- Deletes the database file from the file system.

When an application deletes a database, the destructors of the containers and basic objects in the database are *not* called. You can ensure that destructors are called by explicitly deleting each contained object before deleting the database.

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. An Objectivity / C++ application represents a container as an object of the class `ooContObj` on one of its derived classes and works with that object through a handle.

Containers are the smallest 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.

Your use of containers can affect the performance and concurrency of your applications, so you should give some thought to how you organize basic objects into containers; see “Assigning Basic Objects to Containers” on page 131.

A container is physically maintained within a database file. It is implemented as a *container object* that manages a particular group of disk pages allocated within a particular database file. An Objectivity / C++ application represents a container object as an instance of the class `ooContObj` or one of its derived classes.

Every container is both a storage object and a persistent object:

- Because a container is a storage object, you can give it a system name, cluster basic objects in it, iterate over those objects, and so on. As with other storage objects, you cannot create versions of a container.
- Because a container is a persistent object, you can give it a scope name, save an object reference to it in an attribute of another persistent object, establish associations to it from other persistent objects, add it to a persistent collection, and so on. In addition, you can update any application-specific attributes or associations on a container of an application-defined class.

Hashed and Nonhashed Containers

Containers can be either *hashed* or *nonhashed*. A hashed container provides an efficient lookup mechanism for finding objects in name scopes; a nonhashed container occupies less storage than a hashed container, but does not support name scopes. If you plan to use a container as the scope object for a name scope, it must be hashed. If you plan to use a basic object as a scope object, it must be stored in a hashed container. See “Scope Objects” on page 333.

When you create a container, you specify a *hash value* that determines whether the container is to be hashed. A nonhashed container has a hash value of 0; a hashed container has a hash value of 1.

Kinds of Container

An application typically creates or accesses one or more containers in each database. In most cases, these containers are represented by instances of the Objectivity / C++ class `ooContObj`; such containers are called *standard containers* in this book.

An application that interoperates with Objectivity for Java or Objectivity / Smalltalk applications may choose to create *garbage-collectible containers* instead of standard containers; garbage-collectible containers are created as instances of class `ooGCContObj`, which is a predefined class derived from `ooContObj`.

If containers are to have application-specific data or associations, an application can define its own persistence-capable container classes derived from `ooContObj` (or `ooGCContObj`); such classes are defined in DDL files along with other persistence-capable classes. However, most applications have no need to define their own container classes.

A *default container* is created automatically by Objectivity / DB for each database. The default container holds:

- Basic objects that are clustered with the database but not explicitly assigned to an application-created container.
- A hash table for scope names of persistent objects that are named in the scope of the database. Consequently every default container is hashed, with a hash value of 1.

Every default container is an instance of `ooDefaultContObj` and has the system name `_ooDefaultContObj`. You cannot create additional default containers in a database, although you specify a default container's initial number of logical pages and growth factor when you create the database.

Creating a Container

You create a standard container by using operator new on class ooContObj within an update transaction. This operator is generally used as follows:

```
// Create a new container referenced by container handle contH
ooHandle(ooContObj) contH = new(parameters) ooContObj;
```

The *parameters* allow you to specify:

- A system name for the container

  If you intend to find the container by name, specify a system name; otherwise specify an empty string or null. You can also use scope names to look up containers, but the system name mechanism provides better concurrency.

- Whether the container is to be hashed

  A hashed container provides an efficient lookup mechanism for scope-named objects, but occupies more storage than a nonhashed container. If you intend to use the container or any object it contains as a scope object, the container must be hashed; otherwise, the container should not be hashed. See “Scope Objects” on page 333.

- The initial number of logical 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 the container’s size must be increased to enable it to accommodate more basic objects, it will grow by the specified amount.

- A clustering directive

  The directive indicates where to locate the new container. Specify an object reference or handle to a database, container, or basic object; the new container is created in the referenced database or in the same database as the referenced container or basic object. If you omit this directive, the new container is created in the most recently opened or created database.

You can omit all *parameters* or specify just the clustering directive to create an unnamed, nonhashed container with an initial size of 2 logical pages and a growth factor of 10%. You can specify 0 for any parameter to use the default value (the default initial size is 2 logical pages for a nonhashed container and 4 logical pages for a hashed container.

The newly created container is automatically opened for update, and is made visible to other transactions when you commit or checkpoint the transaction.

Note that `operator new` returns a memory pointer, which *must* be assigned to a container handle. (If an object reference is desired, you can then assign the handle to an object reference.) Although direct assignment to a pointer or object reference does not raise compile-time or runtime errors, such assignments can eventually cause the Objectivity/DB cache to run out of memory. This is because Objectivity/C++ handles perform memory management for persistent objects whereas object references and pointers do not.

Creating a Container of an Application-Specific Class

If your application has defined its own container classes, `operator new` is available on each such class. To create a container from an application-specific container class *`contClass`*, the `new` operator is generally used as follows:

```
ooHandle(contClass) contH =
      new(parameters) contClass(initializers);
```

The *`parameters`* are the same as those for creating a standard container; the *`initializers`* are any values you want to pass to a *`contClass`* constructor.

A source file that creates or finds instances of an application-specific container class must include the primary header file that contains the generated definition of that class. For example, assume you define a container class `Cell` in a DDL file `myClasses.ddl`. A source file that creates and finds instances of `Cell` must include the primary header file `myClasses.h`, which is generated from the DDL file by the DDL processor. See the Objectivity/C++ Data Definition Language book for a description of the DDL processor and the files that it generates.

EXAMPLE This example defines a persistence-capable container class `Cell` and creates an instance of this class.

```
// DDL file myClasses.ddl
class Cell : public ooContObj {
   public:
   Cell(char *cellName) { strcpy(name, cellName); }
   char name[32];
};
```

```
// Application code file
#include "myClasses.h"
...     // Start an update transaction
ooHandle(ooDBObj) dbH;
...                          // set dbH to reference a database
// Create a new container with the following characteristics
// and set cellH to reference it
//    System name: adder
//    Hashed (hash value = 1)
//    10 initial logical pages
//    Grow by 20%
//    Cluster in database referenced by dbH

ooHandle(Cell) cellH;
cellH = new("adder", 1, 10, 20, dbH) Cell("adder");
```

Creating Multiple Containers

You can create multiple containers in a single operation by using the ooNewConts macro. This macro provides better performance than calling `operator new` repeatedly. All the resulting containers are unnamed and are created in the same database. You can request that they be closed upon creation.

EXAMPLE This example creates 10 closed instances of the container class `Computer`. Handles to the new containers are placed in the array `compH`.

```
// DDL file computer.ddl
class Computer : public ooContObj {
   public:
      Computer();
      Computer(char* name);
      char name[32];
      uint32 id;
};
```

```
// Application code file
#include "computer.h"
...    // Start an update transaction
ooHandle(ooDBObj)dbH;
ooHandle(Computer)compH[10];
...                    // Set dbH to reference a database

// Create 10 unhashed Computer containers
ooNewConts(Computer,  // Create containers of class Computer
          10,         // Create 10 containers
          dbH,        // Cluster containers in the database
                      // referenced by dbH
          0,          // Not hashed
          0,          // 2 initial logical pages (the default)
          0,          // Grow by 10% (the default)
          oocFalse,   // Do not open the new containers
          compH);     // Array of handles to be set
if (compH[0]==0) {
   fprintf(stderr, "ooNewConts failed\n");
}
```

Creating a Transient Container

You can create a transient instance of a container class by invoking `operator new` with a clustering directive of 0. This essentially creates a transient object containing just the container's application-specific data and associations (if any), and so is of interest only for application-defined container classes. Transient containers are not actual storage objects, so basic objects cannot be clustered in them.

Checking Whether a Container Exists

You can test whether a particular container exists by calling the `exist` member function on a container handle, specifying a handle to the containing database and the system name of the desired container. If the specified container is found, this member function returns `oocTrue` and also sets the container handle to reference the specified container; otherwise, `oocFalse` is returned.

Testing for a container's existence helps to avoid the errors that are signaled if you attempt to create a container with a nonunique system name or if you attempt to open a nonexistent container.

Finding a Container

You can find a container in any of the following ways:

- If the container has a system name, look it up by name by calling the `exist` or `open` member function on a container handle. (You should use `exist` if you want to find a container without opening it.) If the specified container is found, the handle is set to reference it.
- Initialize an object iterator of class `ooItr(ooContObj)` to find all containers in a database by calling the `contains` member function on a handle to the database.

  After initializing the object iterator, you advance it through the iteration set by calling the iterator's `next` member function. See "Object Iterators" on page 293 for information about working with an object iterator.
- Scan a database or the entire federated database for containers by calling the `scan` member function on a container iterator.
- Find the container that contains a particular basic object by calling the `containedIn` member function on a handle to the basic object.
- Find the default container in a database by calling the `getDefaultContObj` member function on a handle to the database.

Because a container is a persistent object as well as a storage object, you can also find a container in any of the ways that you would find a persistent object. See "Finding Persistent Objects" on page 185. Although you can use scope names to find containers, you should consider using system names instead for better concurrency. This is because setting and looking up scope names may lock containers that contain basic objects, whereas operations on system names do not.

Once you have an object reference, handle, or object iterator that references a container, you can get a string containing the container's system name by calling the `name` member function. A null pointer is returned if the container does not have a system name.

Opening a Container

Opening a container makes it available to an application. Opening a container:

- Implicitly locks the container for the requested level of access (read or update).
- Obtains a representation of the container in memory, either by fetching buffer pages from the database or reusing a cached memory representation that is guaranteed current. This memory representation includes buffer pages describing the container and any application-specific data it may have (if the container is an instance of an application-defined container class).

Opening a container for update marks it as modified, causing any application-specific data to be written to the database when the transaction commits. You must be in an update transaction to open a container for update. If necessary, you can promote a read transaction to an update transaction by promoting the open mode of the federated database.

Because opening a container locks it and the objects it contains, you should consider the implications for concurrency when you open a container, taking into account the concurrent access policy of the transaction. Note, however, that opening a container does *not* automatically open the basic objects in it. Therefore, you must explicitly open individual basic objects for update before modifying them, even if the containers in which they reside are already open for update.

Operations that Open Containers Implicitly

In general, Objectivity / DB opens containers implicitly. A container is:

- Opened for update when it is created.
- Opened for read when an application accesses any of its members—for example, by dereferencing a container handle using `operator->`.
- Opened for read or update when a basic object in it is opened for read or update. However the reverse is not true—opening a container does not open the basic objects in it.

Furthermore, operations such as scope-name lookup, traversal, or scanning optionally open the found containers. The member functions for these operations usually provide an *`openMode`* parameter for specifying whether the found containers should be opened, and, if so, for which level of access.

Explicitly Opening a Container

You must open a container explicitly when:

- You require update access so you can modify the container's application-specific data or associations.
- You want to reserve either read or update access to the container in advance—for example, before starting a complex operation. You do not otherwise need to explicitly open a container for read, because simply accessing the container or an object in it opens it for read.

To find and open a container explicitly, you call the `open` member function on a container handle; you must provide a handle to the database to be searched and the system name of the desired container. If you already have an object reference, handle, or object iterator that references the container, you can call `open` without these parameters. In either case, you specify the parameter value `oocRead` to open the database for read or `oocUpdate` to open the database for update.

Checking and Promoting the Level of Access

After you have obtained a handle to a container, you can determine the current level of access by calling the `openMode` member function.

If a container is open for read, and you want to secure update access to it, you can call the `update` member function on a handle to the container. This member function is equivalent to `open(oocUpdate)`.

Closing a Container

Because a container is a persistent object as well as a storage object, you can close it as you would a persistent object. See "Closing a Persistent Object" on page 195. Closing a container affects just the container's application-specific data, if any, and has no effect on the basic objects it contains.

Deleting a Container

Deleting a container causes it to be removed from the federated database when the transaction commits. You can use `operator delete`, the `ooDelete` global function, or the `ooDeleteNoProp` global function to delete a container. The `delete` operator requires that you extract a pointer from a handle to the container to be deleted. The `ooDelete` and `ooDeleteNoProp` functions require that you specify a handle to the container to be deleted.

Deleting a container calls the destructor, if any, on the container class and deletes any associations from the deleted container to destination objects. Furthermore, deleting a container deletes all of the contained basic objects and any associations from the deleted basic objects to destination objects. However, for performance reasons, the destructors of the contained objects are *not* called. You can ensure that destructors are called by explicitly deleting each contained object before deleting the container.

If any deleted object (including the container) has a bidirectional association, the deleted object is removed from the inverse association of 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.

Both the `ooDelete` function and the `delete` operator propagate deletion along associations that have delete propagation enabled. This means that, if any associations on the container have delete propagation enabled, their destination objects are deleted as well. If you do not want deletion to be propagated along any associations, use the `ooDeleteNoProp` function instead. For more on delete propagation, see "Propagating Operations" on page 148.

You must use the `delete` operator of the container class to delete a transient container instance.

EXAMPLE This example defines a container class `Cell`, creates two `Cell` containers, and deletes them using `operator delete` and `ooDelete`.

```
// DDL file myClasses.ddl
class Cell : public ooContObj {
public:
   char name[32];
   Cell(char* cellName) { strncpy(name, cellName, 32); }
};
```

```
// Application code file
#include "myClasses.h"
...
ooHandle(ooDBObj) dbH;
...     // Start an update transaction
...     // Set dbH to reference a database

// Create a cell container with default characteristics in dbH
ooHandle(Cell) tempCellH = new(dbH) Cell("temp");
if (tempCellH == 0) {printf("Error !\n");}

// Create a cell container with nondefault characteristics in dbH
ooHandle(Cell) cellH;
cellH = new("adder", 1, 10, 40, dbH) Cell("adder");
if (CellH == 0) {printf("Error !\n");}

// Delete the temp cell container using ooDelete function
ooDelete(tempCellH);

// Delete the adder cell container using operator delete
delete (Cell*)cellH;
```

9

Persistent Objects

Objectivity / DB basic objects and containers are persistent objects. This chapter describes Objectivity / C++ facilities for managing persistent objects. Some operations are applicable to all persistent objects; others, to basic objects only:

- General information about persistent objects and persistence-capable classes
- Creating a new basic object
- Finding, opening, examining, modifying, closing, and deleting a persistent object
- Copying and moving a basic object

Understanding Persistent Objects

A *persistent object* continues to exist and retain its data 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.

A basic object or a container is made persistent at creation time when it is assigned a storage location in a federated database. The process of specifying where to store a persistent object is called *clustering*. When you commit the transaction in which you create a persistent object, that object's data is saved in the federated database; the object can then be accessed by other processes.

Persistent objects reside in an Objectivity / DB federated database and are brought into virtual memory when requested by an application. Once a persistent object is in virtual memory the application can manipulate it.

NOTE You work with a persistent object through a handle to the object. For details about using handles, see Chapter 10, "Handles and Object References".

Persistence-Capable Classes

Only instances of *persistence-capable classes* can be persistent objects. Objectivity / C++ includes persistence-capable classes for containers and a few kinds of basic objects, for example, persistent collections. An Objectivity / C++ application can define additional persistence-capable classes.

As is the case with any C++ object, a source file that creates or works with an instance of a persistence-capable class must include (directly or indirectly) the header file containing the definition of that file.

To use instances of persistence-capable classes defined by Objectivity / C++, the source file must include the corresponding Objectivity / C++ header file.

To Use	You Must Include
Container classes (Chapter 8)	`oo.h`
Name-map class `ooMap` and related classes (Chapter 11)	`ooMap.h`
Scalable-collection classes and related classes (Chapter 11)	`ooCollections.h`
Java-compatibility classes (Chapter 7)	`javaBuiltins.h`
Classes to create and search indexes (Chapter 18)	`ooIndex.h`

To use instances of the application-defined persistence-capable class *`appClass`*, the source file must include the primary header file that contains the generated definition of *`appClass`*. For example, assume you define a persistence-capable class `Shape` in a DDL file `myClasses.ddl`. A source file that creates or finds instances of `Shape` must include the primary header file `myClasses.h`, which is generated from the DDL file by the DDL processor.

See the Objectivity / C++ Data Definition Language book for a description of the DDL processor and the files that it generates.

Persistence Behavior

The Objectivity / C++ class `ooObj` and its handle class `ooHandle(ooObj)` together define persistence behavior. Every persistence-capable class is derived from the class `ooObj`; its handle class is derived from `ooHandle(ooObj)`. A persistence-capable class thus inherits persistence behavior directly from `ooObj` and indirectly (through its handle class) from `ooHandle(ooObj)`.

The following persistence behavior is available for all persistent objects (basic objects and containers):

- You can create links to a persistent object. See Chapter 15, “Creating and Following Links”.

- You can find a persistent object within its containing storage object. See “Finding Contained Objects” on page 357.
- You can find a persistent object within any storage object above it in the storage hierarchy. See “Scanning a Storage Object” on page 360.
- You can give a persistent object a name in the scope of some other Objectivity / DB object, allowing you to look it up by that name. See “Finding an Object by Scope Name” on page 335.

Because containers are storage objects as well as persistent objects, various aspects of their persistence behavior are affected by their storage concerns. For a complete description of containers, see “Containers” on page 170.

The following additional persistence behavior is available for basic objects only:

- You can copy a basic object, creating a new basic object. See “Copying a Basic Object” on page 197.
- You can move a basic object from one container to another. See “Moving a Basic Object” on page 201.
- You can create different versions of a basic object. See Chapter 20, “Versioning Basic Objects”.

Transient Instances

If you instantiate an application-defined basic-object class without clustering it in or near an existing Objectivity / DB object, the resulting object is transient. A transient object has no persistence behavior. An application works with a transient object just as it would with any standard C++ object—*not* as it would work with an Objectivity / C++ persistent basic object. In particular:

- You may not reference a transient object with a handle or an object reference.
- A transient object may not be the source object or the destination object of any association.

You should avoid using transient instances of persistence-capable classes. If you do create such transient objects, you should use them only as isolated scratch pads for temporary data values. An alternative to using transient objects for this purpose is to create short-lived persistent objects in a scratch container that is deleted before the transaction commits. Doing so allows you to take advantage of Objectivity / DB cache management for handling data that exceeds the size of virtual memory.

Creating a Basic Object

You create any kind of a persistent object—either a basic object or a container—by using `operator new` on a persistence-capable class in an update transaction. This operator is defined on the Objectivity/C++ class `ooObj` and is made available in all basic-object classes and container classes that derive from `ooObj`. This section focuses on basic objects; "Creating a Container" on page 172 describes how to create a container.

To create a basic object of a persistence-capable class *`myClass`*, you generally use `operator new` as follows:

```
// Assume myClass derives from ooObj; create a new basic object
// referenced by handle objH
ooHandle(myClass) objH = new(near) myClass(initializers);
```

The *`initializers`* are any parameter values you want to pass to the *`myClass`* constructor. The *`near`* parameter is a *clustering directive* that allows you to specify an existing object with which to cluster the new basic object. The clustering directive can be an object reference, a handle, or a pointer to a database, a container, or a basic object.

- When you cluster a basic object with a database, the object is stored in that database's default container.
- When you cluster a basic object with a container, the basic object is stored in that container.
- When you cluster a new basic object with an existing basic object, the new object is stored in the same container as the existing object; the new object is placed on the same logical page as the existing object or on a nearby page, depending on available space.

If you omit the clustering directive, the new basic object is stored in the default container of the most recently opened or created database. (If you specify a null pointer, you create a transient object.)

The newly created basic object is automatically opened for update; when you commit or checkpoint the transaction, the new basic object is made visible to other transactions.

Note that `operator new` returns a memory pointer, which *must* be assigned to a handle. Although direct assignment to a pointer or object reference does not raise compile-time or runtime errors, such assignments can eventually cause the Objectivity/DB cache to run out of memory. This is because Objectivity/C++ handles perform memory management for persistent objects whereas object references and pointers do not.

EXAMPLE This example creates an instance of the persistence-capable basic-object class `Employee`.

```
// DDL file company.ddl
class Employee : public ooObj {
   ...
};
```

```
// Application code file
#include "company.h"
...
ooHandle(Employee) emp1H, emp2H;
...     // Start an update transaction
...     // Set emp1H to reference an existing Employee object

// Create a new Employee object clustered near the one that
// emp1H references and set emp2H to reference the new object
emp2H = new(emp1H) Employee();
```

Finding Persistent Objects

You can find existing persistent objects in the database using several techniques. The approaches that you choose reflect the organization of objects within the federated database. Typically, you first find an object of interest either by individual lookup (Chapter 16) or by iterating over the objects in a particular group (Chapter 17). If a found object has links to other objects, you find those objects by following the links (Chapter 15).

Certain searches—scanning a storage object or following links defined by a to-many association—can be limited by content-based filtering (Chapter 18). That is, instead of finding all objects of a given class at any level in the storage hierarchy below a storage object, you can find only those objects with a certain combination of attribute values. Similarly, instead of finding all destination objects linked to a particular source object by a to-many association, you can find only those destination objects with a certain combination of attribute values.

When you find a persistent object, you obtain either an object reference or handle to that object. If you plan to work with the found object, performing more than one Objectivity / DB operation on it, you need a handle to it. If you obtained an object reference instead, you can assign it to a handle.

Opening a Persistent Object

Once you have found a persistent object and obtained a handle to it, you must open the object to make its data available for reading or modifying. Opening an persistent object:

- Implicitly locks the object for the requested level of access (read or update).
- Obtains a representation of the persistent object in the Objectivity / DB cache, either by fetching the necessary buffer page(s) from the database or by reusing an existing memory representation that is guaranteed current.
- Produces a memory pointer to the object. This pointer is encapsulated by the handle through which the object is opened.
- If the persistent object is a basic object, opens the container in which the basic object resides.

Read and Update Access

A persistent object can be opened either for read or for update:

- Opening an object for read indicates that you intend to view the object without modifying it.
- Opening an object for update indicates that you intend to modify the object and causes Objectivity / DB to save your changes in the federated database when the transaction commits.

You can open an object for read in either a read transaction or an update transaction. To open an object for update, however, you must be in an update transaction; otherwise an error is signaled. If necessary, you can promote a read transaction to an update transaction by promoting the open mode of the federated database.

WARNING Although you are not prevented from doing so, you should never modify an object that is open only for read. Modifications to such an object are written to the federated database only if the object resides on the same memory page as another object that is open for update. If you want to modify an object that is open only for read, you should explicitly promote it to update access first; otherwise, your changes may be lost.

If a persistent object is a basic object, opening it for update may create a new version of the object. Versioning is supported for basic objects, but not for containers. If a basic object is an instance of a versionable class and has versioning enabled, opening that object for update causes a new version of the object to be created. See Chapter 20, “Versioning Basic Objects”.

Locks

Opening a persistent object implicitly obtains an appropriate lock on that object:

- Opening an object for read implicitly obtains a read lock.
- Opening an object for update implicitly obtains an update lock.

Whether the persistent object being opened is a basic object or a container, opening it locks a container—either the container in which the opened basic object resides or the container being opened. Because locking a container effectively locks *all* the basic objects in it, you should consider the implications for concurrency when you open a persistent object, taking into account the concurrent access policy of the transaction. See "Concurrent Access Policies" on page 113.

In general, you can improve concurrency by minimizing how long objects are open for update—for example, by opening them for read until they are to be modified and only then promoting them to update access. However, this must be balanced against the need to guarantee update locks in advance. For more information, see Chapter 6, "Locking and Concurrency".

Opening a Persistent Object Implicitly

In general, Objectivity / DB opens objects implicitly:

- A persistent object is opened for update when the application creates it.
- A persistent object is opened for read when the application accesses any of its members—for example, by using `operator->`.
- If the persistent object being opened is a basic object, its container is also opened. However the reverse is not true—if the object being opened is a container, the basic objects in it are not opened.

Furthermore, many operations that find objects can optionally open the found objects. The member functions for these operations usually provide an *openMode* parameter for specifying whether the found objects should be opened, and if so, for which level of access.

Opening a Persistent Object Explicitly

You must open a persistent object explicitly when:

- You require update access so you can modify the object.
- You want to reserve either read or update access to the object in advance—for example, before starting a complex operation. You do not otherwise need to explicitly open an object for read, because simply accessing the object allows you to read it.

You explicitly open a persistent object by calling the `open` member function on a handle to the object. This member function allows you to specify an open mode

parameter, either `oocRead` to open the object for read or `oocUpdate` to open the object for update.

For convenience, you can use the `update` member function on a handle to explicitly open the referenced object for update. This member function is equivalent to calling the `open` member function and specifying the open mode `oocUpdate`.

Opening a persistent object of any kind—basic object or container—allows you to modify that object and informs Objectivity/DB that the object must be written to the federated database when the transaction commits. Note, however, that you must explicitly open each basic object to be modified. This is true even if all the objects reside in the same container—that is, opening a container for update is not a shortcut for opening the objects in it. If you modify both a container that is open for update and a basic object in it that is open for read, only the changes to the container itself are written to the federated database when the transaction commits; changes to the basic object are lost.

EXAMPLE In this example, handles `rectH1` and `rectH2` are set to reference basic objects located in the `Rectangles` container. The object referenced by `rectH1` is explicitly opened for update.

Note that the object referenced by `rectH2` is implicitly opened for read when it is accessed. It is opened only for read even though the container containing it is open for update.

```
// Application code file
#include "geometry.h"
...
ooTrans trans;
ooHandle(ooFDObj) fdH;
ooHandle(ooDBObj) dbH;
ooHandle(ooContObj) contH;
ooHandle(Rectangle) rectH1;  // Rectangle is a subclass of ooObj
ooHandle(Rectangle) rectH2;
trans.start();                  // Start a transaction
fdH.open("ECAD", oocUpdate);    // Make it an update transaction
dbH.open(fdH, "EPROM");         // Open database
contH.open(dbH,"Rectangles", oocUpdate); // Find container
...     // Set rectH1, rectH2 to reference basic objects
rectH1.open(oocUpdate);  // Open basic object explicitly
rectH1->x = 0;           // Set the value of its data member
// Open basic object for read implicitly
printf("Value of x is %d\n", rectH2->x);
...
```

Getting Information About a Persistent Object

Various member functions of `ooObj` and its handle class `ooHandle(ooObj)` allow you to get information about a persistent object.

Runtime Type Identification

When a persistence-capable class is added to the schema of a federated database, it is assigned a unique *type number* that identifies the class within the schema. The global type `ooTypeNumber` represents a type number.

Objectivity/C++ uses type numbers to provide runtime type identification (RTTI) for persistent objects. A handle of type `ooHandle(ooObj)` can be set to reference a persistent object of any class; RTTI allows you to determine the class of the object that a handle is currently referencing. This feature is useful when you need to take different actions depending on the class of the referenced object.

The following member functions of a handle enable you to find the class of the referenced persistent object:

- `typeName` gets the name of the referenced object's class.
- `typeN` gets the type number of the referenced object's class.

The following member functions of a persistent object enable you to find its class:

- `ooGetTypeName` gets the name of the object's class.
- `ooGetTypeN` gets the type number of the object's class.
- `ooIsKindOf` tests whether the object belongs to the class with the specified type number. An object *belongs to* a class `C` if it is an instance of the class `C` or an instance of a class derived from `C`.

Because you work with a persistent object through a handle, you typically use the handle member functions `typeName` and `typeN` to get the class name or type number of an object's class. Within a member function of a persistence-capable class, however, you would use the persistent-object member functions `ooGetTypeName` and `ooGetTypeN` instead.

To obtain the type number of a particular persistence-capable class, you can call the global macro `ooTypeN`, passing the class name as the parameter. You can use a series of `if-else` conditional statements to test whether a particular type number is the type number of various classes.

EXAMPLE This example initializes an object iterator to find a group of persistent objects of classes derived from the abstract class `Fruit`. It performs one operation on apples, a different operation on oranges, and a still different operation on berries. An apple is an instance of the class `Apple`; an orange is an instance of the class `Orange`; a berry is an instance of any class derived from the abstract class `Berry`.

```
// Application code file
#include "fruits.h"
...
ooItr(Fruit) fruitI;
ooTypeNumber typeNum;
... // Initialize the object iterator fruitI;
while (fruitI.next()) {
   // Set typeNum to the type number of the current fruit
   typeNum = fruitI.typeN();
   if (typeNum == ooTypeN(Apple)) {
      ... // Perform operation for apples
   }
   else if (typeNum == ooTypeN(Orange)) {
      ... // Perform operation for oranges
   }
   else if (fruitI->ooIsKindOf(ooTypeN(Berry))) {
      ... // Perform operation for berries
   }
}
```

You cannot use the global macro `ooTypeN` as a label in a `switch` statement because `ooTypeN` is expanded into a variable name. For example, the following `switch` statement causes a compilation error:

```
ooHandle(ooObj) objH;
switch(objH.typeN()) {
   case ooTypeN(A):      // Error! Can't use ooTypeN as label
      ...
   case ooTypeN(B):      // Error! Can't use ooTypeN as label
      ...
}
```

Getting the Object Identifier

For testing purposes, you may want to get the object identifier of a persistent object. You can do so by calling either of the following member functions on a handle to the object:

- The `print` member function prints the object identifier.
- The `sprint` member function returns a string containing the object identifier.

Both these member functions use the following string representation of an object identifier:

"*#D-C-P-S*"

The components of the object identifier indicate the object's location in the federated database. The following table gives the meaning of each component symbol in the object identifier of a basic object and in the object identifier of a container.

Symbol	Object Identifier of Basic Object	Object Identifier of Container
D	Identifier of the object's database	Identifier of the container's database
C	Identifier of the object's container	Identifier of the container
P	Number of the logical page on which the basic object is stored	1 for an unhashed container; a low integer for a hashed container
S	The slot number on the page in which the basic object is stored	1

See the Objectivity/DB administration book for a description of the storage layout of a federated database, including pages within containers and slots on a page.

Testing a Persistent Object for Validity

If your application needs to check the validity of persistent objects of some application-defined class, you can override the `ooValidate` member function in that class. As defined by the class `ooObj`, `ooValidate` simply returns `oocTrue`. You can override `ooValidate` to perform whatever checks are necessary to test whether an object of your class is valid. It should return `oocTrue` for a valid object and `oocFalse` for an invalid one.

Do not confuse the `ooValidate` member function of a persistent object with the `isValid` member function of a handle. The `ooValidate` member function checks that the *persistent object* is valid; the application defines what it means for an object of the class to be "valid" in its implementation of this function. In contrast, the `isValid` member function of a handle tests whether the *handle* is valid. Objectivity/C++ defines a valid handle to be one that references an existing persistent object that the application can access; see "Testing Whether a Handle is Valid" on page 220.

Getting a Handle in a Member Function

To perform many Objectivity / DB operations on a persistent object, you do not call a member function on the object itself; instead, you call a member function on a handle to the object. If you want to perform such an operation from within a member function of a persistence-capable class, you must first get a handle to the object on which the member function was called. The `ooThis` member function allows you to do so.

WARNING You should not call `ooThis` from the constructor of a persistence-capable class; doing so may cause the application to terminate in an error condition.

A persistent object's `ooThis` member function is analogous to the C++ keyword `this` used within a member function. Whereas `this` is a pointer to the object, `ooThis` returns a handle to the object. The class `ooObj` defines `ooThis` to return a handle of the class `ooHandle(ooObj)`. The DDL processor redefines `ooThis` in each application-defined persistence-capable class. For example, the persistence-capable class `Library` would define `ooThis` to return a handle of the class `ooHandle(Library)`.

If you want to set a particular handle to reference a persistent object, you can pass that handle as the optional parameter to the persistent object's `ooThis` member function. If `objH` is a handle, the following two statements are equivalent; the former is more efficient:

```
ooThis(objH);      // More efficient way to set handle
objH = ooThis();   // Less efficient way to set handle
```

A variant of `ooThis` sets and returns an object reference to the persistent object.

EXAMPLE The persistence-capable class `Company` defines a member function `printInfo`, which prints a company's scope name and object identifier. When called on an instance of `Company`, the `printInfo` function calls `ooThis` to obtain a handle to the instance, and then uses the handle to get information about the instance.

```
// DDL file
class Company : public ooObj {
public:
   ...
   void printInfo() {
      ...
      ooHandle(Company) this_companyH;
      // Set this_companyH to reference this Company object
      ooThis(this_companyH);
```

```
      // Get handle to the appropriate scope object
      ooHandle(ooObj) scopeH = ...;
      // Print the company's scope name
      cout << this_companyH.getObjName(scopeH);
      // Print the company's object identifier
      cout << " (" << this_companyH.sprint() << ")" << endl;
   } // end printInfo
}; // End class Company
```

Modifying a Persistent Object

You should ensure that a persistent object is open for update before you modify its attributes or associations. When a persistent object is open for update during a transaction, any modifications to that object made during that transaction will be written to the federated database when the transaction is committed.

You need to consider two situations in which modifications can occur:

- Application code can obtain a handle to a persistent object and then modify the object through that handle.
- A member function defined in the persistence-capable class can modify the object on which the function was called.

Modifying Through a Handle

When you modify a persistent object using a handle to it, you should first check its access level; if the object is not already open for update, you should open it for update before making the modification.

You can determine the current level of access to an object by calling the `openMode` member function on an open handle to the object. This member function returns `oocRead`, `oocUpdate`, or `oocNoOpen` (if the handle is not open).

If a persistent object is open for read, you can promote it to update access simply by reopening it. You can use either the `open` member function (with the parameter set to `oocUpdate`) or the `update` member function. You do not need to close the object first.

EXAMPLE In this example, the basic object referenced by `rectH` is implicitly opened for read when its `draw` member function is called. The object is then promoted to update access so that modifications to the object will be saved persistently when the transaction is committed.

```
// Application code file
#include "geometry.h"
...
ooHandle(Rectangle) rectH;  // Rectangle is a subclass of ooObj
...                    // Set rectH to reference a Rectangle
rectH->draw();         // Open object for read implicitly
rectH.update();        // Promote access to update
rectH->layerN = 10;    // Modify attribute of object
```

Objectivity/DB automatically promotes a container from read to update access when a basic object in it is opened for update.

Modifying Within a Member Function

When you modify a persistent object from within a member function defined on the object's class, that member function must ensure that the object is open for update.

- If your member function modifies *only* associations of the object, no additional operation is required. Each generated member function that sets an association of the object first opens the object for update.
- If your member function modifies one or more attributes of the object, that member function should open the object for update by calling its `ooUpdate` member function. The `ooUpdate` member function is defined by `ooObj` and inherited by every persistence-capable class.

EXAMPLE This example shows an accessor member function called `setCount` in a persistence-capable class called `Inventory`. The member function sets the `count` attribute of the `Inventory` object for which it is called; to ensure that modifications to the object will be saved persistently when the transaction is committed, `setCount` calls `ooUpdate`.

```
// DDL file
class Inventory : public ooObj {
public:
   ...
   uint32 count;
   void setCount(uint32 newCount) {
      ooUpdate();        // Open this object for update
      count = newCount;  // Modify its 'count' attribute
   }
};
```

Closing a Persistent Object

An open persistent object is closed when all handles to the object are closed. You can explicitly close each handle using the `close` member function on that handle. However, you rarely need to do this because Objectivity / DB automatically closes handles when they go out of scope, when they are set to reference other objects, or when the transaction that opened them commits or aborts.

Explicitly closing a handle informs Objectivity / DB that the application no longer requires access to the referenced object through this handle. Closing does not, however, release any locks; locks are released only by committing or aborting a transaction.

Closing a persistent object permits Objectivity / DB to swap it out of the cache as necessary to make room for other objects. More precisely, if multiple persistent objects are on the same buffer page in memory, *all* of them must be closed before Objectivity / DB can swap out the page. You should consider closing unused objects within a transaction if you need to reduce your application's virtual memory requirements.

If you need to access a persistent object after it has been closed, you must open it again.

EXAMPLE This example demonstrates reopening an object after it has been closed. In this case, the object referenced by the handle `polyH` is reopened implicitly for read access.

```
// Application code file
#include "geometry.h"
...
ooHandle(Polygon) polyH;
...
// Set polyH to reference a polygon.
...
// Open the polygon explicitly through its handle and modify it
polyH.open(oocUpdate);
polyH->set_origin(0,0,5,5);
// Close the polyH handle, closing the polygon if polyH is the
// only handle to it
polyH.close();
// Open the same polygon implicitly and then explicitly
// close the polyH handle. This will close the polygon if polyH
// is the only handle to it.
polyH->draw();
polyH.close();
```

Deleting a Persistent Object

Deleting a persistent object causes it to be removed from the federated database when the transaction commits. You can delete a persistent object even if multiple open handles reference the object. That is, the object is deleted no matter how many times it has been opened.

Deleting a persistent object calls the destructor, if any, on the object's class and deletes any associations 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 persistent object, you typically call the `ooDelete` global function, passing a handle to the object to be deleted.

EXAMPLE This example calls the global function `ooDelete` to delete a `Rectangle` object.

```
// Application code file
#include "geometry.h"
...
// Set rectH to reference the rectangle to be deleted
ooHandle(Rectangle) rectH = ...;

ooDelete(rectH);                    // Delete the rectangle
```

As required by C++, `operator delete` is defined in the Objectivity/C++ class `ooObj` and is available on all application-defined subclasses of `ooObj`. As usual, the parameter to `delete` is a pointer to the object to be deleted. You should use `operator delete` to delete transient instances of a persistence-capable class and use `ooDelete` to delete persistent instances.

Functions that are indifferent to persistence (for example, in legacy code or a third-party library) may use `operator delete` to delete a persistent object. Typically, a pointer to the persistent object is extracted from a handle and passed as a parameter to such a function. The handle class for each persistence-capable class defines a conversion operator that returns a pointer to the referenced object. See "Extracting a Pointer to a Persistent Object" on page 233.

NOTE The behavior of `delete` operator on a persistent object is different from the standard C++ `delete` operator. Namely, it removes the object from the federated database as well as deleting the object from memory.

When you use either the `ooDelete` function or the `delete` operator to delete a persistent object that has associations for which delete propagation is enabled, you also delete the destination objects linked by those associations. If you want to delete a source object without deleting its destination objects, you can call the `ooDeleteNoProp` function instead. For additional information about delete propagation along association links, see "Propagating Operations" on page 148.

Copying a Basic Object

You can copy basic objects only, not containers. To create a copy of a basic object, you call the `copy` member function on a handle to the object. The first parameter to `copy` specifies the database, container, or basic object with which to cluster the new copy of the basic object.

The `copy` member function returns a handle to the newly created copy. Alternatively, you can pass an object reference or handle as the optional second parameter to `copy`; the specified object reference or handle is set to reference the new copy.

The `copy` member function copies the original object *only*. If the original object has associations or object references to other persistent objects, those objects are *not* copied. However, you can arrange for postprocessing to propagate the copy operation to associated and referenced objects; see "Customizing the Copy Operation" on page 199.

Copied Attributes and Associations

The attributes of the new object created by `copy` are set to *bit-wise* copies of the corresponding attributes of the original basic object. Consequently, the new object contains an exact copy of the value of each attribute in the original object. Such values include primitive values, fixed-size arrays, VArrays, embedded objects of non-persistence-capable classes, and object references. If bit-wise copying invalidates any copied data, you can arrange for postprocessing to fix these values; see "Customizing the Copy Operation" on page 199.

If the original object has associations to other objects, each association is treated according to its copy behavior in the object's class definition. As described in "Copying and Versioning Behavior" on page 150, the association's definition can specify that links should be retained by the original object only, transferred to the

new copy, or duplicated so that both the original and the copy are associated with the same destination object(s).

EXAMPLE The class `Door` has attributes `width` and `height` that contain the dimensions of the door. It has an association `inRoom` that links a door to the room in which it is located. The `inRoom` association has the default copy behavior, namely, when a door is copied, the original door is linked to the room, but the new copy is not.

```
// DDL file house.ddl
class Room : public ooObj {
   …
};

class Door: public ooObj {
public:
   uint16 width;
   uint16 height;
   // Define the 'inRoom' association with default copy
   // behavior: Delete any 'inRoom' link from a new copy
   ooRef(Room) inRoom : copy(delete);
   Door(uint16 w, uint16 h) { width = w; height = h; }
   …
};
```

The application code finds a room, creates a door in that room whose width is 36 and whose height is 84, and copies the door to another room.

```
// Application code file
#include "house.h"
…
ooTrans trans;
ooHandle(Door) dH, new_dH;

trans.start();
…           // Open the federated database for update
// Get a handle to the first room
ooHandle (Room) r1H = … ;
// Create a new persistent door, clustered with the first room
dH = new(r1H) Door(36, 84);
// Get a handle to the second room
ooHandle (Room) r2H = … ;
// Copy the Door object
dH.copy(r2H,       // Cluster near the second room
        new_dH);   // Set new_dH to reference the new door
// The new door has the same width (36) and height (84)
// as the original; it is not linked to any room
```

```
// Now link the new door to the second room
new_dH2->set_inRoom(r2H);
// Commit the transaction
trans.commit();
```

Customizing the Copy Operation

In a standard C++ application, a class can customize the C++ copy operation by defining a copy constructor or overloading the assignment operator (=). Similarly, in an Objectivity / C++ application, a persistence-capable class can customize the Objectivity / DB copy operation by overriding the virtual `ooCopyInit` member function.

After a basic object has been copied, the `ooCopyInit` member function of the new copy is called automatically to perform any class-specific postprocessing. As defined by the class `ooObj`, `ooCopyInit` simply returns the success status. A persistence-capable class can customize the copy operation by overriding this virtual function.

A custom `ooCopyInit` function can perform any necessary operations on attribute data members for which bit-wise copying is inadequate. For example, the `ooCopyInit` function might initialize attributes that should not be copied. Similarly, the `ooCopyInit` function could propagate the copy operation to associated or referenced objects. Propagating the copy operation is sometimes called creating a *deep copy*.

The `ooCopyInit` member function is always called as part of a copy operation. Because a persistent object can only be copied during an update transaction, `ooCopyInit` is always called from within a transaction, so it does not have to start a transaction itself.

EXAMPLE The class `B` has an attribute `numUsers` that records the number of users who have looked up the object. Its attribute `myA` contains an object reference to an instance of class `A`. Class `B` overrides the virtual function `ooCopyInit` to initialize the `numUsers` data member to 0 and to copy the referenced `A` object.

```
// DDL file myClasses.ddl
class A : public ooObj {
   ...
};
```

```
class B: public ooObj {
public:
   …
   uint32 numUsers;
   ooRef(A) myA;
   ooStatus ooCopyInit() {
      // Initialize 'numUsers' to 0 (instead of
      // the number of users of the original B object)
      numUsers = 0;
      // Set this_bH to reference this B object
      ooHandle(B) this_bH;
      ooThis(this_bH);
      // Make this B object a deep copy of the original B object
      ooRef(A) new_aR;
      if (myA) {
         // Copy the A object referenced by 'myA'
         myA.copy(this_bH,  // Cluster new A copy with this B
                  new_aR);  // Set new_aR to reference the copy
         // Replace the 'myA' data member with an
         // object reference to the new A copy
         myA = new_aR;
      }
      return(oocSuccess);
   } // End ooCopyInit
}; // End class B
```

The application code finds an object of class `B` and creates a deep copy whose `numUsers` data member is initialized to 0.

```
// Application code
#include "myClasses.h"
…
ooTrans trans;
ooHandle(B) bH, new_bH;

trans.start();
…           // Open the federated database for update
bH = …;     // Find the desired B object

// Copy the B object.
bH.copy(bH,        // Cluster new B copy with original B object
        new_bH);   // Set new_bH to reference the new B copy
trans.commit();    // Commit the transaction
```

The `copy` member function first creates a bit-wise copy of the attributes `numUsers` and `myA`, so the new `B` copy has the same number of users and references the same `A` object as does the original `B` object. Then, the new `B` copy's `ooCopyInit`

member function is called automatically. That function resets `numUsers` to zero, copies the referenced `A` object, and sets the `myA` data member of the new `B` copy to reference the new `A` copy.

Moving a Basic Object

You can move a basic object to a different container; you cannot, however, move a container to a different database. To move a basic object, call the `move` member function on a handle to that object. The parameter to `move` is a handle to the database, container, or basic object with which to cluster the moved object. The object is moved from its current location to the location that results from clustering it with the specified object—that is, to the default container of the specified database, to the specified container, or to the container of the specified basic object.

A successful move operation gives the basic object a new object identifier that indicates its new storage location; the operation modifies the handle to reference the basic object by its new object identifier. If the move is unsuccessful, the object identifier is unchanged; the handle remains valid and still references the object by the original object identifier.

EXAMPLE This example moves a basic object of the `Publication` class from the `bookstore` container to the `newsstand` container.

```
// Application code file
#include "publications.h"
...
ooTrans trans;
ooHandle(ooFDObj) fdH;
ooHandle(ooDBObj) dbH;
ooHandle(ooContObj) booksH, newsH;
ooHandle(Publication) pubH;

trans.start();
fdH.open("SHIP", oocUpdate);
dbH.open(fdH, "retail", oocUpdate);

// Open the container with the system name "bookstore"
booksH.open(dbH, "bookstore", oocUpdate);

// Open the container with the system name "newsstand"
newsH.open(dbH, "newsstand", oocUpdate);
```

```
// Find the publication named "Exploring the Web" in the
// bookstore container
pubH.lookupObj(booksH, "Exploring the Web")

// Move the publication from the bookstore to the newsstand
pubH.move(newsH);

trans.commit();       // Commit the transaction
```

Preserving Referential Integrity

Because a moved basic object has a new object identifier, all references containing the old identifier become invalid; furthermore, Objectivity / DB may eventually reassign the old identifier to a new persistent object. Therefore, when you move an object, you should, within the same transaction, update references to the object within all relevant attributes, unidirectional associations, persistent collections, and indexes. Objectivity / DB automatically maintains referential integrity for bidirectional associations.

The following subsections outline general techniques for preserving referential integrity when moving an object. You can implement these techniques using mechanisms described in “Customizing the Move Operation” on page 204.

Reference Attributes

If any attribute of another persistent object contains an object reference to the moved object, that attribute must be modified, removing the existing object reference and replacing it with a new object reference containing the new object identifier. Remember that affected attributes may contain a single object reference, a fixed-size array of object references, or a VArray of object references.

Unidirectional Associations

If the moved object is a destination object in a unidirectional association from some source object, you must delete the association from the source object before the move and then set the association again after the move. For more information, see “Linking Objects by To-One Associations” on page 321.

Persistent Collections

If you want to move an object that is an element, key, or value in a persistent collection, you must remove the object from the persistent collection before the object is moved, and then add the object back to the persistent collection after the

move is successfully completed. See “Building a Persistent Collection” on page 242 for information about adding objects to persistent collections and removing objects from persistent collections.

Indexes

A predicate scan using an index that references a moved object will yield undefined results. In contrast to associations, persistent collections, and name scopes, it is not possible to delete a reference to an individual object from an index. Therefore, you must drop the entire index before moving an object that is referenced by the index and recreate the index after the object is moved. For information about creating and dropping indexes, see “Indexes” on page 390.

Preserving Scope Names

Scope names are deleted when you move *either* a scope object or an object that has scope names. If you want to preserve scope names when you move such an object, you should reestablish the names within the same transaction.

Moving a Named Object

To maintain referential integrity in name scopes, Objectivity / DB deletes a named object from a name scope when the named object is moved. If you want to preserve scope names when you move a named object, you must:

1. Before the named object is moved, find all scope objects that define a name scope in which the object is named, and determine the object’s scope name in each scope. See “Finding Scope Objects” on page 372.
2. After the named object is moved, reestablish, in each scope, the scope name of the moved object. See “Building a Name Scope” on page 333.

Moving a Scope Object

If a basic object is used as the scope object for a name scope, it uses the hashing mechanism of its container to maintain its name scope. See “Scope Objects” on page 333. If you move the scope object to a different container, Objectivity / DB deletes its entire name scope from its old container. If you want to preserve the name scope when you move the scope object, you must:

1. Before the scope object is moved, find all the objects named in that scope and obtain the scope name of each object. See “Finding Named Objects” on page 370.
2. Move the scope object to another hashed container. See “Hashed and Nonhashed Containers” on page 170.

3. After the scope object is moved, reestablish each scope name in the scope of the moved object. See “Building a Name Scope” on page 333.

Customizing the Move Operation

When you move a basic object, the move operation consists of three steps. First, a call to the `ooPreMoveInit` member function of the object to be moved performs any class-specific preprocessing. Next, the object is moved to its new location. After the object has been successfully moved, its `ooPostMoveInit` member function is called to perform any class-specific postprocessing.

As defined by the class `ooObj`, `ooPreMoveInit` and `ooPostMoveInit` simply return the success status. Your persistence-capable class can override one or both of these virtual member functions to customize the move operation. For example, your functions can execute any operations required for preserving referential integrity. These member functions are called from within an update transaction, so they do not have to start a transaction themselves.

EXAMPLE This example shows how to preserve a moved object’s scope name. Basic objects of the `Building` class are named in the scope of the `Designs` database; a building’s scope name is also saved in its `name` data member. The `Building` class overrides the `ooPreMoveInit` member function to remove the building’s scope name; it overrides the `ooPostMoveInit` member function to restore the moved building’s scope name.

```
// DDL file
class Building : public ooObj {
public:
   char name [32];
   ...
   ooStatus ooPreMoveInit() {
      ooHandle(ooFDObj) fdH;
      ooHandle(ooDBObj) designH;
      ooHandle(Building) thisH;
      // Get a handle to the Designs database
      fdH.open("Architecture", oocUpdate);
      designH.open(fdH, "Designs", oocUpdate);
      // Set thisH to reference this Building object
      ooThis(thisH);
      // Remove the scope name for this Building object
      // from the Designs database
      thisH.unnameObj(designH);
      return(oocSuccess);
   } // End ooPreMoveInit
```

```
   ooStatus ooPostMoveInit() {
      ooHandle(ooFDObj) fdH;
      ooHandle(ooDBObj) designH;
      ooHandle(Building) thisH;
      // Get a handle to the Designs database
      fdH.open();
      designH.open(fdH, "Designs", oocUpdate);
      // Set thisH to reference this Building object
      ooThis(thisH);
      // Set the scope name for this Building object
      thisH.nameObj(designH,   // Scope object
                    name);     // Scope name
      return(oocSuccess);
   } // End ooPostMoveInit
}; // End class Building
```

10

Handles and Object References

Following the ODMG standard, Objectivity/C++ uses a reference-based approach to working with Objectivity/DB objects. Objectivity/C++ provides two choices for referencing Objectivity/DB objects: *handles* and *object references*.

This chapter describes:

- General information about handles and object references
- Guidelines for choosing a handle or an object reference
- How to work with a handle or an object reference
- Casting a handle based on the class of the referenced object
- Restrictions on using pointers to persistent objects
- Saving space with short object references

Understanding Handles and Object References

An Objectivity/C++ application works with an Objectivity/DB object indirectly through one or more handles or object references that are set to reference the object. In general, every handle or object reference serves to both:

- Identify the referenced object to the application or to another object.
- Provide an interface for operating on the referenced object.

In addition, a handle or object reference to a persistent object serves as a type-safe smart pointer that:

- Manages the memory pointer to the object.
- Provides an indirect member-access operator (->) for accessing the object's public member functions and data members.

Although handles and object references can be used to accomplish many of the same tasks, they are optimized for very different purposes, so you should sometimes use a handle instead of an object reference, and vice versa. This choice is summarized in "Choosing a Handle or Object Reference" on page 216. A

simple guideline is to use object references when linking persistent objects together and to use handles when operating on objects in memory.

Handle and Object-Reference Classes

Handles and object references are instances of the parameterized classes `ooHandle(`*`className`*`)` and `ooRef(`*`className`*`)`, which exist for every class *`className`* derived from `ooObj`. An instance of `ooHandle(`*`className`*`)` or `ooRef(`*`className`*`)` can reference an instance of *`className`* or any of its derived classes.

Objectivity / C++ provides predefined handle and object-reference classes for every predefined class of Objectivity / DB objects. For example, the handle class `ooHandle(ooContObj)` and the object-reference class `ooRef(ooContObj)` are defined for referencing instances of the standard container class `ooContObj`.

The DDL processor generates handle and object-reference classes for every persistence-capable class defined by an application. For example, if a DDL file contains the definition of a basic-object class `Library`, the DDL processor generates the definitions of the corresponding handle class `ooHandle(Library)` and object-reference class `ooRef(Library)`.

NOTE Because of similarities between each handle class `ooHandle(`*`className`*`)` and the corresponding object-reference class `ooRef(`*`className`*`)`, the documentation sometimes refers to the two classes as *`ooRefHandle`*`(`*`className`*`)`.

Object Identification

Handles and object references use unique *Objectivity/DB identifiers* as the basis for referencing objects. Such identifiers are the means by which Objectivity / DB distinguishes every object from other objects of the same type. Thus, a federated database's identifier distinguishes it from the other federated databases using the same lock server; a database's identifier distinguishes it from the other databases in the same federated database; a container's identifier distinguishes it from the other containers in the same database; and so on.

For most types of Objectivity / DB objects, the identifier is a single integer that serves as a key for locating each object relative to the storage object that contains it. For example, within a federated-database catalog, each database identifier is mapped to a particular database file; within a database catalog, each container identifier is mapped to a particular page within the database file.

However, the identifier of a basic object, called an *object identifier* or *OID*, contains enough information to distinguish it from every other basic object within the entire federated database, not just within the object's container. The object

identifier of a basic object consists of four components that identify the database, container, logical page (within the container), and logical slot (within the page) where the identified object resides. For example, 78-112-8-3 identifies the basic object stored in slot 3 of page 8 in container 112 of database 78.

For uniformity, every Objectivity / DB identifier can be expressed as an object identifier (that is, in a four-part format). For example, the database identifier 78 can be expressed as the object identifier 78-0-0-0. (The object identifier format for a container identifier is somewhat more complex; it contains the page and slot number of the container object itself.) For more information about object identifiers, see the Objectivity / DB administration book.

Within an Objectivity / C++ application, handles and object references use object identifiers in much the same way that pointers in a C++ program use memory addresses. Just as an object identifier uniquely identifies an Objectivity / DB object by its storage location in a federated database, a memory address can uniquely identify a transient object by its location in virtual memory. However, memory addresses are insufficient for identifying Objectivity / DB objects, which exist externally to the memory of any application. A single Objectivity / DB object can be represented in memory multiple times—for example, by different applications or threads running concurrently, or by the same application each time it executes—and every memory representation normally has a different address.

NOTE Handles and object references store the value 0 in place of an object identifier when no object is currently referenced. Analogous to null pointers, these are called *null handles* and *null object references*.

Referencing Databases, Federations, and Partitions

A handle or object reference to a database, federated database, or autonomous partition is essentially a wrapper for the referenced object's identifier. For example, when you set a handle to reference a particular database, the handle stores the object identifier of that database. If the database is then opened through the handle, Objectivity / DB uses the identifier to locate the database file on disk.

You can access a database, federated database, or autonomous partition only through a handle or object reference that references it. For example, you operate on a database by setting a handle to reference it and then calling member functions on the handle. The handle's member functions operate on the instance of `ooDBObj` that represents the referenced database in memory. The *`ooRefHandle`*`(ooDBObj)`,*`ooRefHandle`*`(ooFDObj)`,and*`ooRefHandle`*`(ooAPObj)` classes provide the complete interfaces for operating on databases, federated databases, and autonomous partitions, respectively.

When databases, federated databases, or autonomous partitions are referenced, handles are nearly equivalent to object references. That is, for most purposes, an application can use either a handle or an object reference to reference such objects; applications typically use handles. The key difference is that only object references (but not handles) can be passed between Objectivity contexts, because:

- A handles contains an identifier for the referenced object, along with cache-related state specific to the Objectivity context in which the handle was created.
- An object reference simply contains an identifier for the referenced object, but no additional cache-specific state.

Referencing Persistent Objects

Like handles and object references to other Objectivity / DB objects, a handle or an object reference to a persistent object (a basic object or container) contains the unique identifier of the referenced object and provides an interface for operating on that object:

- The interface for performing core persistence operations is defined by each of the *ooRefHandle*(ooObj) classes and is inherited by all the handle and object-reference classes for basic objects and containers. (Additional persistence operations are defined by ooObj itself.)
- The interface for performing container-specific operations is defined by each of the *ooRefHandle*(ooContObj) classes and is inherited by all the handle and object-reference classes for containers.

As part of the interfaces they define, the handle and object-reference classes for persistent objects overload the indirect member-access operator (->) so you can access the members of the referenced object. Thus, when you have a handle or object reference to a basic object or a container, you can use the indirect member-access operator to:

- Call the member functions of the referenced object.
- Get and set the values of any application-defined attributes that the referenced object may have.

You can access a persistent object only through a handle or object reference that references it; the accessing handle or object reference manages a pointer to the object's memory representation, so you do not need to create or destroy any pointers explicitly. However, pointer management is performed differently by handles and object references, because they are designed for different purposes:

- Handles are designed for use as smart pointers—that is, for performing multiple operations on a referenced object or repeatedly accessing the object's members (see "Reference Counting With Handles" on page 212).
- Object references are designed for use as persistent addresses—that is, for storing object identifiers persistently in reference attributes, in associations,

or as elements of a collection (see “Object References as Persistent Addresses” on page 213).

Handles as Smart Pointers to Persistent Objects

You use a handle (instead of an object reference) to operate on a persistent object or to access its members, because handles are designed for efficient in-memory access. In particular, a handle has extra cache-related state that enables you to *open* and *close* the handle and to *pin* a persistent object in memory.

Open and Closed Handles

When you set a handle to reference a particular persistent object, the handle stores just the object identifier of the referenced object. The first time you access the referenced persistent object through the handle, Objectivity / DB automatically:

- Opens the referenced object, if necessary. An *open persistent object* is locked and represented in memory. (If the referenced object is already open, Objectivity / DB simply finds the existing memory representation.)
- Opens the accessing handle. An *open handle* has a valid pointer to the memory representation of the referenced open object.

Subsequent operations performed through the open handle use the stored memory pointer for quick access to the referenced object. When you no longer need to access the referenced object through the handle, you can close the handle, which invalidates the pointer but preserves the object identifier. Figure 10-1 illustrates two handles, one open and one closed, both of which reference the basic object with object identifier `#7-12-132-6`.

A closed handle can be reopened on the same object or set to reference a different object. Handles that are not closed explicitly during a transaction are closed automatically at the end of the transaction.

Closing a handle may (but need not) close the referenced object, as described in the next section. Thus, a closed handle can reference either an open or a closed object, although an open handle by definition references an open object.

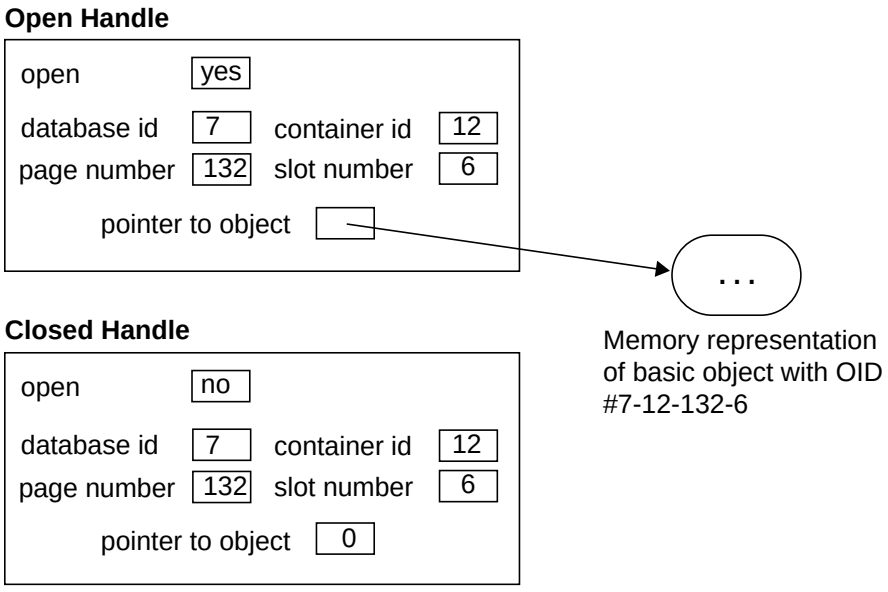


Figure 10-1 Open and Closed Handles

Reference Counting With Handles

Besides maintaining a pointer to the referenced persistent object, an open handle pins the object in the Objectivity / DB cache. *Pinning* is a reference-counting mechanism that keeps track of references to a persistent object in memory. Opening a handle to a persistent object adds a memory reference (a *pin*) to that object; closing a handle removes a pin from the referenced object.

A persistent object's *pin count* is the total number of memory references to the object within a particular Objectivity context. As long as a persistent object has a positive pin count, it is said to be *pinned* in the cache. A pinned object is considered to be in use and remains open (its persistent data is kept in the Objectivity / DB cache). When a persistent object's pin count falls to zero, the object itself is closed because it is no longer considered to be in use. A *closed persistent object* may be swapped out of the cache to free up space for other open objects. (More precisely, the buffer page containing a closed persistent object may be swapped out of the cache, but only if *all* persistent objects on the same page are closed.) Certain cache operations are normally performed on an object's memory representation when the object is closed, although these may be delayed for performance reasons; see "Timing of Cache Operations" on page 74.

Because handles are the primary means of adding and removing pins, a persistent object's pin count is normally equal to the number of open handles to

the object. However, certain other operations can affect a persistent object's pin count, as described in "Pointers, Handles, and Object References" on page 232.

NOTE Handles are the only means of managing pins on newly created persistent objects, so the result of `operator new` on a persistence-capable class must be assigned to a handle.

Summary of Open and Closed States

Open and closed states pertain both to handles and to persistent objects. The following table summarizes these states for each item.

	Open	Closed
Persistent Object	The object's persistent data is guaranteed to be available in the Objectivity/DB cache.	The object's persistent data is *not* guaranteed to be in the Objectivity/DB cache.
Handle to Persistent Object	The handle has a memory pointer to the referenced object and holds a pin on that object.	The handle has the object identifier of the referenced object, but no memory pointer or pin. The referenced object can be either open or closed.

Object References as Persistent Addresses

Object references are designed for referencing persistent objects independently of the Objectivity / DB cache, so they are primarily used as persistent addresses. Object references are also capable of accessing objects in memory, although they are not designed for this purpose.

Cache-Independent Reference

Like a closed handle, an object reference contains the object identifier of the persistent object it references. Unlike a handle, however, an object reference has no built-in cache-related state, so object references are much smaller than handles and cannot be opened—that is, an object reference can neither store a pointer nor pin an object in memory. An object reference to a persistent object is essentially a wrapper for that object's identifier, and is indifferent to whether the referenced object is open or closed.

Figure 10-2 illustrates an object reference that is set to reference the basic object with object identifier `#7-12-132-6`. The object reference contains the same information whether or not the persistent object is represented in the cache.

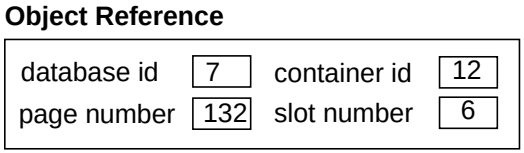


Figure 10-2 Object Reference

Accordingly, the primary use of an object reference is to store an object identifier persistently. In particular, when two persistent objects are linked together, the source object of the link maintains an object reference to identify the destination object of the link. The source object may be:

- A persistent object with an attribute data member that contains an object reference to the destination object.
- A persistent object with an association data member that maintains an object reference to each destination object in the association.
- A persistent collection that maintains an object reference to each persistent object in the collection.

Object references are used for storing object identifiers transiently whenever a cache-independent reference is required. In particular, you can pass an object reference between Objectivity contexts to make the stored object identifier available to the receiving context. An object reference can also be useful for keeping an object identifier in memory over a long period of time without occupying much cache space (that is, without pinning the object itself).

NOTE Objectivity / C++ enforces the use of object references instead of handles for linking persistent objects and for passing identifiers between Objectivity contexts.

Expensive In-Memory Access

Like a handle, an object reference can be used as a smart pointer to access a persistent object; in fact, object references provide essentially the same interface as handles. However, instead of acquiring a pointer to the object being accessed, an object reference creates a temporary handle to the object, performs the requested operation through the handle, and then destroys the handle. The temporary handle opens the persistent object, if necessary, and pins the object for the duration of the operation.

For performance reasons, you should avoid using an object reference for repeated accesses to a persistent object. Poor performance results from repeated

access, because *each* accessing operation causes a temporary handle to be created, used, and discarded. Performance may also be affected if the referenced object is swapped out between operations, when it is unpinned.

An object reference is appropriately used as a smart pointer when all you need is a single access to the referenced object—for example, when you want to conveniently call just one member function on the result of an operation that traverses an association link.

Syntactic Interchangeability

Although handles and object references are semantically very different, the Objectivity/C++ programming interface regards them as syntactically interchangeable. In particular, Objectivity/C++ functions accept a handle wherever an object reference is requested, and vice versa:

- You can specify a variable of type `ooRef(`*`className`*`)` to a function that accepts a parameter of type `const ooHandle(`*`className`*`) &`.
- You can specify a variable of type `ooHandle(`*`className`*`)` to a function that accepts a parameter of type `const ooRef(`*`className`*`) &`.

Implicit type conversion is performed for such parameters because:

- Every handle class has a constructor for creating a handle from an object reference. The constructed handle contains the same object identifier as the specified object reference.
- Every object-reference class has a constructor for creating an object reference from a handle. The constructed object reference contains the same object identifier as the specified handle.

In some cases, function overloading is used to achieve the same effect—that is, the function has one declaration that expects a handle, and a second declaration that expects an object reference.

NOTE In the Objectivity/C++ programmer's reference, when the declarations of an overloaded function are the same except for a parameter for specifying a handle or an object reference, the declarations are collapsed and the parameter type is given as the abbreviation *`ooRefHandle`*`(`*`className`*`)`.

The syntactic interchangeability of handles and object references exists for convenience—for example, to allow you to specify the result of one function as a parameter of another function without any intermediate steps. However, this convenience can come at a price. In particular, if you have an object reference to a persistent object and you pass the object reference to multiple functions that need to access the referenced object, *each* function will convert the object reference to a

handle and then open the persistent object through it. This is much more expensive than creating a single handle to be passed to each function.

Choosing a Handle or Object Reference

The following guidelines summarize when to use a handle or an object reference.

When referencing a persistent object:

- Use an **object-reference class** as the data type of an attribute data member or an association data member in a persistence-capable class.

  Only object references can be used for linking persistent objects together. In a persistence-capable class definition, the DDL processor accepts data members of type `ooRef(`*`className`*`)` and issues an error if it encounters a data member of type `ooHandle(`*`className`*`)`.

- Use a **handle variable** to reference a newly created persistent object.

  The result of `operator new` must first be assigned to a handle to manage the new object's pin count properly. (The handle can subsequently be assigned to an object reference—for example, in an attribute data member or persistent collection.)

- Use a **handle variable** to reference a persistent object whenever you intend to perform multiple operations on the referenced object or to repeatedly access its members with the indirect member-access operator (->).

  A handle is appropriate even if you don't know how often the referenced object will be accessed. Opening a handle has the same performance impact as a single access through an object reference; once the handle is open, it provides a significant performance advantage for any subsequent operations.

- Use an **object-reference variable** to reference a persistent object without pinning it in the Objectivity / DB cache.

  An object reference is appropriate for a static variable containing long-lived global state—for example, for storing the result of a lookup that will be used later to initialize variables. (The object reference should be assigned to a handle if the referenced object is to be accessed.)

- Use an **object-reference variable** when you need to pass an object identifier between Objectivity contexts.

  Objectivity / DB signals an error if you attempt to pass a handle variable from one context to another.

When referencing a database, a federated database, or an autonomous partition:

- Use either a **handle variable** or an **object-reference variable** within a single Objectivity context; use an **object-reference variable** when passing an object identifier between contexts.

Working With a Handle

Handles serve as smart pointers to Objectivity / DB objects. You work with a handle to an Objectivity / DB object by:

- Obtaining the definition of an appropriate handle class
- Creating a handle of the chosen class
- Setting the handle to reference the desired Objectivity / DB object
- Testing whether the handle is null, valid, or equal to another handle
- Getting the referenced object's class through the handle
- Operating on the referenced object through the handle

Obtaining a Handle Class Definition

You use a handle of class `ooHandle(`*`className`*`)` to work with an object of class *`className`* or its derived classes. If you are creating handles to predefined Objectivity / C++ classes such as `ooFDObj`, `ooObj`, `ooMap`, `ooTreeList`, and so on, your source file must include the appropriate Objectivity / C++ header file(s) to obtain the required class definitions (see Appendix A, "Objectivity / C++ Include Files").

If you are creating handles for an application-defined persistence-capable class *`appClass`*, your source file must include a generated header file to obtain the definition of `ooHandle(`*`appClass`*`)`. You normally include the primary header file that is generated from the DDL file containing *`appClass`*. If, however, you are simply creating and using handles of class `ooHandle(`*`appClass`*`)`, without actually accessing any instances of *`appClass`* itself, your source file can include just the references header file. For more information on including Objectivity / C++ header files and generated header files, see "Developing Application Source Code" on page 57.

Creating a Handle

You normally create a handle as a local variable on the stack, rather than allocating it on the heap. Creating a handle on the stack allows your application to destroy the handle automatically when it goes out of scope. Thus, the following definition creates a null handle called `dbH` that can be set to reference a database (an instance of class `ooDBObj`):

```
ooHandle(ooDBObj) dbH;
```

A null handle can, but need not, be created within a transaction.

You should not declare a handle as `const`, because its internal state will be changed by any operation that accesses the referenced object through it.

Setting a Handle

You set a handle to reference an Objectivity / DB object in any of the following ways:

- Find the object through the handle, by calling an appropriate member function on the handle or by passing the handle to a function that sets it.
  - Operations for finding the federated database, a database, or a container are described in Chapter 8, “Storage Objects”.
  - Operations for finding a persistent object are described in Chapter 15, “Creating and Following Links,” Chapter 16, “Individual Lookup of Persistent Objects,” and Chapter 17, “Group Lookup of Persistent Objects”.
  - Operations for finding an autonomous partition are described in Chapter 27, “Autonomous Partitions”.
- Create a new object and assign the result to the handle. (This does not apply to a federated database, which cannot be created from within an application.)
  - Operations for creating a database or container are described in Chapter 8, “Storage Objects”.
  - Operations for creating a basic object are described in Chapter 9, “Persistent Objects”.
  - Operations for creating an autonomous partition are described in Chapter 27, “Autonomous Partitions”.
- Use the handle’s overloaded assignment operator (=) to set the handle to the same object as an existing handle or object reference.
- While creating the handle, initialize it with a new object or with an existing handle or object reference. The handle’s constructor sets the handle just as the assignment operator does.

Operations that find or create objects must be performed within a transaction. A handle continues to reference the same object until it is set to another object or to null. The reference is preserved across transaction boundaries, unless the handle goes out of scope or is set to null as the result of an abort operation. Multiple handles can be set to the same object.

EXAMPLE This example shows several techniques for setting handles. Assume `Section` is a persistence-capable class defined in the DDL file `publications.ddl`.

```
// Application code file
#include "publications.h"    // Obtain handle class definition
...
ooTrans trans;
ooHandle(ooFDObj) fdH;       // Create federated-database handle
```

```
ooHandle(ooContObj) bookstoreH;    // Create container handle
ooHandle(Section) sectionH;  // Create handle for Section class

trans.start();
// Find and open federated database; set fdH to reference it.
fdH.open("SHIP", oocUpdate);

// Find and open container; set bookstoreH to reference it.
bookstoreH.lookup(fdH, "bookstore", oocUpdate);

// Create a new section and initialize sportsH with the resulting
// pointer.
ooHandle(Section) sportsH = new(bookstoreH) Section("sports");

// Assign sportsH to sectionH, so both handles reference the
// same section.
sectionH = sportsH;

// Set sportsH to null so it no longer references anything.
sportsH = 0;
```

Testing a Handle

Because you work with persistent objects through handles, you may need to test whether a given handle references an object.

Testing Whether a Handle is Null

Testing whether a handle is null is analogous to testing for a null pointer. Like a pointer, a handle is null if it does not reference any object; a null handle contains no object identifier and its internal memory pointer is null. A handle is null if it is not yet initialized, if it was set by an operation that failed to find or create an object, or if it was open when a transaction aborted.

You can use any of the following operations to test whether a handle is null:

- Call the handle's `isNull` member function. A null handle returns the value `oocTrue`; a nonnull handle returns `oocFalse`.
- Use the handle's equality operator (==) to compare the handle with 0. A null handle is equal to 0; a nonnull handle is not.
- Use the handle in a conditional expression. Because of the handle's conversion-to-pointer operator, a null handle returns 0 and a nonnull handle does not.

The conversion operator opens the object for read, whereas `isNull` and the comparison operators can perform the check without opening the handle.

EXAMPLE This example tests whether the handle `dbH` is null to find out whether a database was successfully created.

```
// Application code file
#include <oo.h>          // Obtain handle class definition
...
ooTrans transaction;
ooHandle(ooFDObj) fdH;
ooHandle(ooDBObj) dbH;

transaction.start();
fdH.open("shapeExample",oocUpdate);

dbH = new(fdH) ooDBObj("simpleShapes");
if (dbH.isNull()) {
   cerr << "Couldn't create database";
   cerr << endl;
   transaction.abort();
}

transaction.commit();
```

Testing Whether a Handle is Valid

A handle that is set during a transaction retains its reference when the transaction commits, so it can be reused (without being reset) to access the same object in a subsequent transaction. However, before you reuse such a handle, you should call its `isValid` member function to test whether the handle is still *valid*—that is, whether it continues to reference an existing Objectivity / DB object that the application can access. A handle is valid if all of the following conditions are true:

- The handle is nonnull, so it has an object identifier.
- The federated database contains an object with that object identifier.
- The application can obtain a read lock on the object's container (if the object is a persistent object).

A handle becomes invalid if it was set to null (for example, by an aborted transaction) *or* if another process has moved or deleted the referenced object between transactions.

Because Objectivity / DB reuses object identifiers, it is possible for a new object to acquire the object identifier of a moved or deleted object. Consequently, a valid handle could eventually point to a different object than the one to which it was

originally set. After testing for validity, it is the application's responsibility to test whether the handle actually points to the expected object.

Testing for Equality Between Two Handles

You can use the equality and inequality operators (== and !=) to test whether two handles (or a handle and an object reference) are set to the same object.

Getting the Class of the Referenced Object

A handle of class `ooHandle(`*`className`*`)` can be set to reference an object of class *`className`* or any class derived from *`className`*. For example, calling the `lookupObj` member function on a handle of class `ooHandle(Vehicle)` could find (and set the handle to reference) an object of class `Vehicle` or any derived class, such as a `Car` or `Truck`. Similarly, various operations can set a handle of class `ooHandle(ooObj)` to reference any kind of Objectivity / DB object.

You can get the class of an object using Objectivity / C++ runtime type identification (RTTI). RTTI is based on the *type numbers* that uniquely identify classes within the schema. Every predefined class for Objectivity / DB objects has a type number; every application-defined class is assigned a type number when the DDL processor adds it to the schema. Type numbers are represented by the global type `ooTypeNumber`. You can obtain the type number for any class of Objectivity / DB objects by calling the global macro `ooTypeN`, passing the class name as the parameter.

RTTI allows you to determine the class of the object that a handle is currently referencing. This feature is useful when you need to take different actions depending on the class of the referenced object. The following RTTI member functions can be called on any type of handle:

- `typeName` gets the name of the referenced object's class.
- `typeN` gets the type number of the referenced object's class

NOTE Persistent objects provide additional member functions for performing RTTI, as described in "Runtime Type Identification" on page 189.

To determine the class of a referenced object, you use `typeN` to obtain its type number and then compare the result with the type number of each class that might be referenced. (You can use the `ooTypeN` global macro to get the type numbers of the candidate classes.) You can perform these comparisons in a series of `if-else` conditional statements; note that you cannot use the `ooTypeN` macro as a label in a switch statement (see "Runtime Type Identification" on page 189).

You usually perform RTTI to find out whether you can safely cast a handle to a derived class. For an example that shows RTTI, see “Explicit Type Conversion (Casting)” on page 229.

Operating on an Object Through a Handle

After a handle has been set to reference an Objectivity / DB object, you can operate on that object by calling various member functions on the handle. To call a member function of a handle, you use the direct member-access operator (`.`).

If the handle is to a persistent object, you can use the handle’s overloaded indirect member-access operator (`->`) to access data members or member functions of the referenced persistent object. Accessing a member of a persistent object through a handle automatically opens an object for read if it is not already open.

EXAMPLE This example initializes a handle to reference a persistent object of the `Vehicle` class, then calls the `rentVehicle` member function of the referenced `Vehicle` object.

```
// Application code file
#include "vehicle.h"        // Obtain handle class definition
...
// Set the handle vH to reference a Vehicle
ooHandle(Vehicle) vH= ...;
// Use . to call member functions of the handle
if (vH.update()) {
   // Use -> to call a member function of the referenced object
   vh->rentVehicle();
}
```

Opening a Handle to a Persistent Object

A handle is automatically opened when a persistent object is opened through it. The open persistent object is both locked and represented in memory; the open handle manages a pointer to the persistent object, pinning the persistent object in memory until the handle is closed. As long as the handle is open, the persistent object remains open also.

You can open a persistent object and the referencing handle in any of the following ways:

- Use the handle’s indirect member-access operator (`->`) to access a member of the persistent object. Accessing implicitly opens the referenced object for read.

- Call the handle's `open` or `update` member function to open the persistent object explicitly.
- Find the persistent object with a function whose *`openMode`* parameter is either `oocRead` or `oocUpdate`. (Most functions that set a handle to a found persistent object provide an *`openMode`* parameter for specifying the desired level of access through that handle.)

In all cases, if the found or referenced persistent object is already open, the accessing handle simply gets a pointer to the persistent object's existing memory representation and increments the object's pin count.

You can test whether a particular handle to a basic object is open by calling the `openMode` member function on the handle. (Note that calling `openMode` on a container handle does not return the state of that handle, but instead reports whether the container itself is open through any handle.)

Closing a Handle to a Persistent Object

Closing a handle removes its pin and decrements the pin count of the referenced object. Closing the last open handle to a particular persistent object reduces the object's pin count to zero, which closes the object itself. A closed object can be swapped out of the Objectivity / DB cache to make room for newly created or opened objects; more precisely, the buffer page containing the closed object can be swapped out, provided that *all* objects on that page are closed. An application can minimize its virtual memory requirements by closing all handles to a persistent object when access to the object is no longer needed.

You obtain a closed handle to a persistent object by finding the object with a function whose *`openMode`* parameter is set to `oocNoOpen`. Such operations simply provide the handle with a persistent object's object identifier without adding a pin, even if the object is already open through another handle.

Objectivity / DB automatically closes an open handle:

- When the handle is destroyed (for example, by going out of scope).
- When the transaction that opened the handle commits or aborts.
- When the handle is set to reference another persistent object. The handle is first closed to unpin the original referenced object, and then either opened or closed depending on the operation that is setting it. For example, if the handle is set by a function with an *`openMode`* parameter, the open mode controls whether the handle is open after being set.

You can close an open handle explicitly by calling the handle's `close` member function.

Operating Through Multiple Handles

If multiple handles have been set to the same object, you can operate on the object through any of these handles. However, you must be aware that an operation performed through one handle could invalidate the other handles. For example, deleting or moving the referenced object through one handle may leave the other handles with an invalid reference.

Passing a Handle as a Parameter

A common practice is to define functions that accept a handle as input. For example, an application could define a function that accepts a null handle, initializes it, and returns it. Similarly, an application-defined function could accept an initialized handle and perform a series of operations on the referenced object.

When a function accepts a handle, the handle should be passed by reference—for example:

```
ooStatus Inspect(ooHandle(ooObj) &objH);     // By reference
```

Passing handles by reference avoids the copying and housekeeping that would be triggered if they were passed by value.

If you want the function to accept either a handle or an object reference, you should overload the function—for example:

```
ooStatus Inspect(ooHandle(ooObj) &object);
ooStatus Inspect(const ooRef(ooObj) &object);   // Overloaded
```

For performance reasons, you should not declare a handle parameter as `const`, if the input handle will be used to open the object it references. Because opening a handle modifies its internal state, a function with a `const` handle parameter would have to create a non-`const` copy of the input handle before the referenced object could be opened.

Working With an Object Reference

Object references serve primarily as persistent addresses for linking persistent objects together, and are sometimes used for operating on an Objectivity / DB object. You work with an object reference to an Objectivity / DB object by:

- Obtaining the definition of an appropriate object-reference class.
- Creating an object reference of the chosen class.
- Setting the object reference to the desired Objectivity / DB object.
- Testing whether the object reference is null, valid, or equal to another object reference.

- Operating on the referenced object through the object reference.

Obtaining an Object-Reference Class Definition

You use an object reference of class `ooRef(`*`className`*`)` to reference an object of class *`className`* or its derived classes. Because `ooRef(`*`className`*`)` is defined in the same header file as `ooHandle(`*`className`*`)`, your source file obtains an object-reference class definition as it would a handle class definition; see "Obtaining a Handle Class Definition" on page 217.

Creating an Object Reference

An object reference that links a source object to a destination object is usually created implicitly. For example:

- If a persistence-capable class defines a data member whose type is an object-reference class, an object reference is created when you create an instance of the containing class.
- If a persistence-capable class defines an association data member, an object reference to the destination object is created and stored in the source object when you create a link for the association.
- When you add a persistent object to a persistent collection, an object reference to that object is created and stored in the persistent collection.

If you use an object reference as a variable in a function, you should create it as a local variable on the stack, rather than allocating it on the heap. You may declare an object reference as `const`. The following definition creates a null object reference called `dbR` that can be set to a database (an instance of class `ooDBObj`):

```
ooRef(ooDBObj) dbR;
```

Setting an Object Reference

You can set an object reference to an Objectivity / DB object with almost any of the operations listed in "Setting a Handle" on page 218. That is, you can:

- Find the object through the object reference.
- Use the object reference's overloaded assignment operator (=) to set the object reference to the same object as an existing handle or object reference.
- Initialize the object reference from an existing handle or object reference.

However, unlike a handle, an object reference should *not* be set directly to a newly created object. Instead, you should:

1. Assign the result of `operator new` to a handle, so that the new object's pin count can be adjusted properly.

2. Assign the handle to the desired object reference. This assignment operation extracts the object identifier from the handle and sets it in the object reference.

EXAMPLE In this example, the `Vehicle` class has an attribute `fleet` to link a vehicle to its rental fleet. The application source code creates a new fleet, sets a handle to reference it, and then sets a vehicle's `fleet` data member by assigning the handle to the object reference in it.

```
// DDL file vehicle.ddl
class Fleet;                //Forward reference to class Fleet
class Vehicle : public ooObj {
public:
   ...
   ooRef(Fleet) fleet;
};
```

```
// Application code file
#include "vehicle.h"
...
// Create a new Vehicle, which implicitly creates a null
// object reference in the vehicle's fleet data member
ooHandle(Vehicle) vehicleH = new Vehicle;
...
// Create a new Fleet and assign it to a handle
ooHandle(Fleet) fleetH = new(vehicleH) Fleet(...);
...
// Assign the new fleet to the object reference
// in the vehicle's fleet data member.
vehicleH->fleet = fleetH;
```

Testing an Object Reference

You can test an object reference as you would a handle; see "Testing a Handle" on page 219 for operations you can use for testing whether an object reference is null, valid, or equal to another object reference or a handle.

Operating on the Referenced Object

After an object reference has been set to an Objectivity/DB object, you can operate on the referenced object as you would through a handle:

- You can call member functions on the object reference by using the direct member-access operator (`.`).
- You can access the members of a referenced persistent object by using the overloaded indirect member-access operator (`->`).

Object references are appropriate for performing single operations or for performing infrequent operations when continuous pinning isn't required.

EXAMPLE This application code uses the classes defined in the example on page 226. The code obtains the object reference that is stored in a vehicle's `fleet` data member and then uses the object reference to get the fleet's location.

```
// Application code file
#include "vehicle.h"
...
// Find a Vehicle, obtaining a handle to it
ooHandle(Vehicle) vH = ...;

// Access the Vehicle's fleet data member and get the fleet's
// location through the returned object reference
char *location = vH->fleet->getLocation;
...
```

Poor performance results when you access a persistent object repeatedly through an object reference (see "Expensive In-Memory Access" on page 214). You can avoid this performance penalty by assigning the object reference to a handle and then using the handle to operate on the referenced object.

EXAMPLE This application code uses the classes defined in the example on page 226. The code obtains the object reference that is stored in a vehicle's `fleet` data member, assigns it to a handle, and then uses the handle to call multiple member functions of the fleet.

```
// Application code file
#include "vehicle.h"
...
// Find a Vehicle, obtaining a handle to it
ooHandle(Vehicle) vH = ...;

// Access the Vehicle's fleet data member and assign the result
// to a handle
ooHandle(Fleet) fleetH = vH->fleet;

// Use the handle to access members of the fleet
fleetH->printVehicleSummary();
fleetH->printRentalHistory();
```

Class Compatibility and Casting

Handle and object-reference classes form inheritance hierarchies that correspond exactly to the hierarchy of classes derived from `ooObj`. Figure 10-3 shows the inheritance hierarchies for an application that manages fleets of rental vehicles.

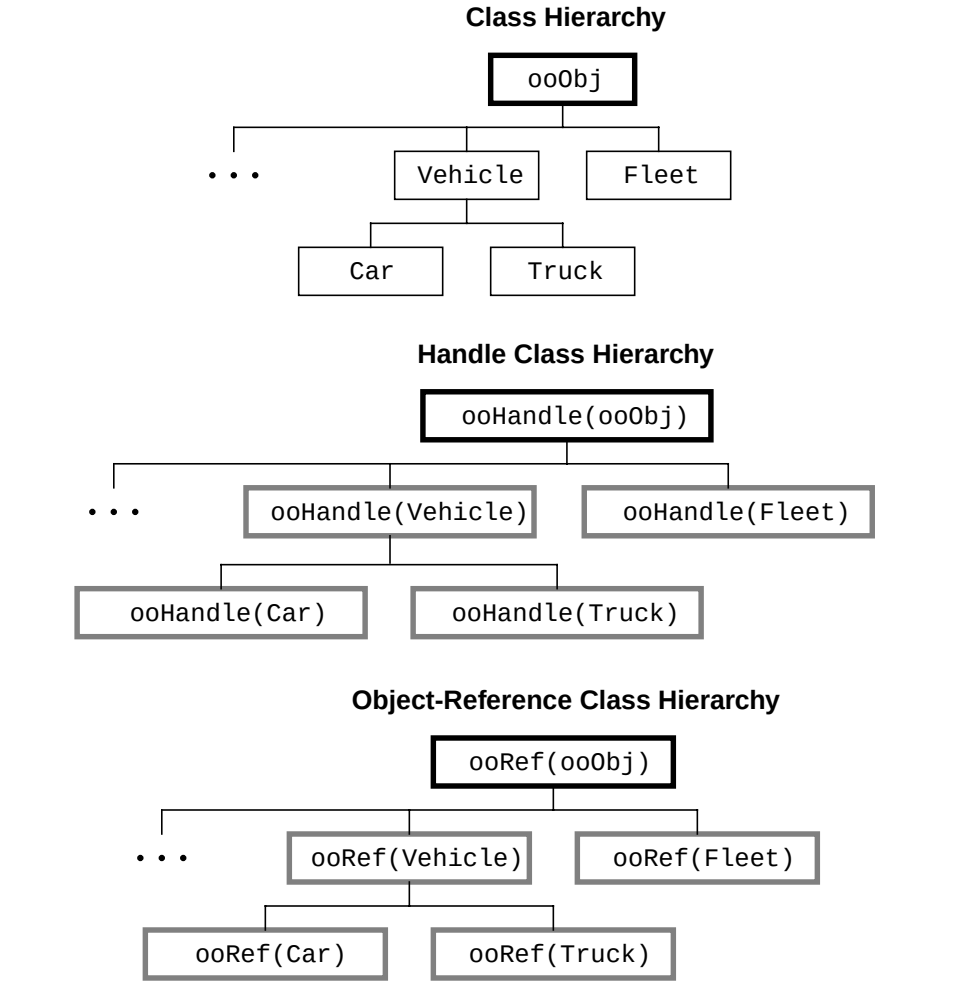


Figure 10-3 Hierarchies of Handle and Object-Reference Classes

Because of these inheritance relationships, you can use handle classes and object-reference classes just as you use pointers to ordinary C++ classes in an inheritance hierarchy. Thus, whenever you want to take advantage of inheritance in your persistence-capable classes, you can do so through their corresponding handles or object references.

NOTE To simplify the discussion, the following two sections describe only handles, although the same information applies to object references.

Implicit Type Conversion

When an application defines a hierarchy of persistence-capable classes, you can take advantage of implicit type conversion to set a base-class handle to reference an instance of any derived class. You can do this by assigning a new derived-class object directly to a base-class handle or by specifying a derived-class handle wherever a base-class handle is requested.

For example, if class `Car` is derived from class `Vehicle` as shown in Figure 10-3, you can perform any of the following operations to set a base-class handle of type `ooHandle(ooVehicle)` to reference an object of the derived class `Car`:

- Assign a new instance of `Car` to a variable of type `ooHandle(Vehicle)`.
- Assign a handle of type `ooHandle(Car)` to a variable of type `ooHandle(Vehicle)`.
- Use a handle of type `ooHandle(Car)` to initialize a variable of type `ooHandle(Vehicle)`.
- Specify a handle of type `ooHandle(Car)` to a parameter of type `ooHandle(Vehicle)`.
- Return a handle of type `ooHandle(Car)` from a member function whose return type is `ooHandle(Vehicle)`.

In each case, you can use the base-class handle to access the referenced object, but only to call member functions or access data members defined by the base class `Vehicle` itself. Member functions of the derived class `Car` are invoked only if they override virtual member functions defined by `Vehicle`.

Explicit Type Conversion (Casting)

If you have a base-class handle that references a derived-class object, and you want to access members defined by the derived class, you must explicitly convert (cast) the handle to the appropriate derived handle class. For example, given the base-class handle `ooHandle(Vehicle)` from the previous section, you must cast it to class `ooHandle(Car)` before you can use it to access the `Car` members of the

referenced object. You can cast the handle safely only if it actually references an instance of `Car`.

To cast a base-class handle to an appropriate derived handle class:

1. Test the class of the referenced object to ensure that it is of the desired derived class; see "Getting the Class of the Referenced Object" on page 221.
2. Use the C++ template function `static_cast` to cast the base-class handle to the appropriate derived handle class.

   Note: You cannot use the C++ template function `dynamic_cast` because Objectivity/C++ handles are C++ objects, not C++ pointers.

EXAMPLE This example defines a function with a parameter of type `ooHandle(Vehicle)`, which accepts a handle of the derived class `ooHandle(Car)` or `ooHandle(Truck)`. The function tests the input handle to determine the class of the referenced object, and then casts the handle so that type-specific operations can be performed.

```
// Application code file
#include "vehicle.h"
…
ooStatus printDescriptions(ooHandle(Vehicle) &vehicleH) {
   ooTypeNumber typeNum;
   ooHandle(Car) carH;
   ooHandle(Truck) truckH;

   // Get the type number of the object referenced by vehicleH
   typeNum = vehicleH.typeN();
   // Determine whether a car is referenced
   if (typeNum == ooTypeN(Car)) {
      // Cast vehicleH to a Car handle
      carH = static_cast<ooHandle(Car) &>(vehicleH);
      // Use carH to perform car-specific operations
      int16 p = carH->getPassengers();
      …
   }
   // Determine whether a truck is referenced
   else if (typeNum == ooTypeN(Truck)) {
      // Cast vehicleH to a Truck handle
      truckH = static_cast<ooHandle(Truck) &>(vehicleH);
      // Use truckH to perform truck-specific operations
      float32 c = truckh->getCapacity();
      …
   }
}
```

General-Purpose Handles and Object References

A handle of class `ooHandle(ooObj)` or an object reference of class `ooRef(ooObj)` can reference an instance of `ooObj` or any class derived from `ooObj`—that is, any kind of Objectivity / DB object. Such *general-purpose handles* and *general-purpose object references* are used for operations in which the specific class of the referenced object doesn't matter or can't be known until runtime. Although a general-purpose handle or object reference can be set to any kind of Objectivity / DB object (any storage object, persistent object, or autonomous partition), the normal practice is to reference just persistent objects (basic objects or containers of any class).

For example, Objectivity / C++ operations that find a persistent object of any class typically set a handle of class `ooHandle(ooObj)` to reference the found object. Similarly, operations that get an element from a persistent collection return an object reference of class `ooRef(ooObj)` that is set to the specified object. General-purpose handles and object references can be used to call any member function defined by `ooObj` or by `ooHandle(ooObj)`; they must be cast as appropriate to access type-specific members of the referenced objects.

EXAMPLE This example assumes a persistent `Fleet` object whose `vehicles` data member is a name map (`ooMap`) of vehicles in the fleet. Each element in the name map is a string containing a vehicle's license number paired with an object reference to the vehicle.

The code accesses the fleet's name map and looks up a particular vehicle; the call to `lookup` returns a general-purpose object reference to the found vehicle. Because multiple operations are to be performed on the vehicle, the object reference is assigned to a general-purpose handle; the handle is cast to a `Car` handle so type-specific operations can be invoked.

```
// Application code file
#include "vehicle.h"
...
ooHandle(ooObj) objH;        // General-purpose handle
ooHandle(Car) carH;          // Car handle
ooHandle(Fleet) fleetH = ...; // Set Fleet handle to a fleet.

// Look up a vehicle in the vehicles map of the fleet and assign
// the returned object reference to the general-purpose handle.
objH = fleetH->vehicles.lookup("456pqr");
```

```
// Test the class of the referenced object and cast to a
// type-specific handle
if (objH->ooIsKindOf(ooTypeN(Car))) {
   carH = static_cast<ooHandle(Car)>(objH);
   ... // Use carH for car-specific operations.
}
```

Guidelines for Multiple Type Conversions

The example in the preceding section illustrates two distinct type conversions:

- Explicit conversion of a base-class handle to a derived handle class.
- Implicit conversion of an object reference to a handle.

For safety, you should keep these conversions distinct by observing the following guidelines:

- Use the C++ template function `static_cast` to adjust the level within a single inheritance hierarchy—that is, to:
  - Cast a base-class handle to a derived handle class.
  - Cast a base-class object reference to a derived object-reference class.

  As always, you should cast only after you test the type of the referenced object (see “Explicit Type Conversion (Casting)” on page 229).
- Use the overloaded assignment operator (=) to switch between referencing mechanisms at the same level in their respective hierarchies—that is, to:
  - Assign a handle of class `ooHandle(`*`className`*`)` to an object reference of class `ooRef(`*`className`*`)`.
  - Assign an object reference of class `ooRef(`*`className`*`)` to a handle of class `ooHandle(`*`className`*`)`.
- Do not cast an object reference to a handle class or vice versa.

Pointers, Handles, and Object References

When an application uses handles and object references to work with persistent objects, Objectivity / DB takes care of memory management transparently—it creates and destroys pointers and memory representations and manages swapping from the Objectivity / DB cache. In contrast, using pointers to reference persistent objects bypasses these memory-management mechanisms, potentially causing persistent objects to be pinned in the Objectivity / DB cache unnecessarily or to be swapped out prematurely. For this reason, an application should avoid manipulating persistent objects through pointers. On occasion, however, explicit

use of pointers is required for performance reasons or for compatibility with functions that are indifferent to persistence.

The *only* pointers to a persistent object that an Objectivity/C++ application may use are a pointer to a newly created object (returned by `operator new`) and a pointer extracted from a handle or object reference.

NOTE This discussion does not apply to databases, federations, and partitions, which have no attributes for persistent data and so are not manipulated through pointers.

Using a Pointer to a New Persistent Object

An Objectivity/C++ application obtains a pointer to a persistent object when it creates that object. As required by C++, `operator new` on a persistence-capable class returns a pointer to the newly created object. You *must* assign the returned pointer to a handle, because Objectivity/C++ memory management depends on this practice. In particular, `operator new` pins the new object in the cache, and `operator=` leaves the object's pin count as is when assigning a pointer to a handle. Thus, after assignment, the handle references an object with a pin count of 1; when the handle is closed, the decremented pin count unpins the object correctly.

Failure to assign a new persistent object to a handle causes the new object to be pinned in the cache until the end of the transaction. Repetition of this practice could cause the application to run out of cache space during a transaction.

Extracting a Pointer to a Persistent Object

You can extract a pointer from a handle by using a conversion operator (`operator` *`className`*`*` for a referenced object that is an instance of *`className`* or a class derived from *`className`*), or by calling the `ptr` member function on the handle. The extracted pointer remains valid only as long as the handle exists, remains open, and references the same object.

WARNING Using an extracted pointer can be dangerous because the pointer is still dependent on the handle, even though they are now separate; if the handle goes out of scope, the pointer will be unsafe.

In keeping with the ODMG standard, you can also extract a pointer from an object reference by calling its `ptr` member function. This member function pins the referenced object in the cache until the end of the transaction; consequently, the extracted pointer remains valid until the end of the transaction. This ODMG

behavior contrasts with the usual behavior of Objectivity/C++ object references, which normally produce a pointer and pin within a temporary handle only for the duration of an access operation. Because `ptr` creates a pin that does not correspond to a handle, there is no mechanism to unpin the object explicitly within a transaction. Heavy use of pointers extracted from object references could therefore cause the application to run out of cache space by pinning too many objects during a transaction.

Some possible uses for an extracted pointer are:

- For performance reasons. For example, you might prefer to sort an array of pointers instead of an array of handles.
- For compatibility with functions that are indifferent to persistence (for example, in legacy code or a third-party library).
- To delete the referenced object from the federated database with the overloaded `operator delete`. (However, the preferred way to delete a persistent object is to simply specify the handle to `ooDelete` without extracting a pointer from it; see "Deleting a Persistent Object" on page 196.)

NOTE You should not use an extracted pointer in other persistence operations—that is, you should not pass it to any Objectivity/C++ member function other than the overloaded `operator delete`.

If necessary, you may assign an extracted pointer to an object reference (but *not* to a handle). For example, you might want to perform further persistence operations on an object after manipulating it through a pointer.

Summary of Restrictions on Pointer Usage

In general, you should never assign, initialize, or construct a handle from a pointer that was not returned by `operator new`, and you should avoid assigning, initializing, or constructing an object reference from a pointer unless you know that the pointer was extracted and is still a valid pointer to a persistent object.

In particular, you should observe the following restrictions on pointers to persistent objects:

- Never assign an extracted pointer to a handle. Because `operator=` does not increment the object's pin count when assigning a pointer to a handle, the handle would not pin the object; so, the object could be closed and swapped while one open handle still referenced it.
- Never assign the result of `operator new` to an object reference or pointer variable. Because `operator new` pins the new object, such an assignment would keep the new object pinned in the Objectivity/DB cache for the remainder of the transaction, possibly causing the cache to run out of

memory. If you need an object reference or pointer to a newly created object, you must first assign the result of `operator new` to a handle. You can then assign the handle to an object reference or extract a pointer from it.

- Never pass a pointer to a member function or operator that expects an object reference or handle. Doing so has the effect of assigning a pointer to a handle (specifically, the pointer is assigned to a temporary handle that is constructed by the function).
- Never assign the pointer returned by the C++ reserved keyword `this` to an object reference or handle. Instead, use `ooThis` in a member function definition to obtain an object reference or handle to the object on which the member function is called. See "Getting a Handle in a Member Function" on page 192.

Saving Storage Space When Linking

When you link persistent objects together through reference attributes or associations, the source object of the link stores an object reference to each destination object of the link. In most cases, the stored object references are instances of `ooRef(`*`className`*`)`—that is, they are *standard object references* containing the complete object identifiers of the referenced objects.

If physical storage space is a concern, you can consider using *short object references* (instances of `ooShortRef(`*`className`*`)`) for links to basic objects in the same container. A short object reference stores a *short object identifier*, which contains only the page and slot number of a basic object, omitting the database and container identifiers. A short object reference takes up approximately half as much space as a standard object reference.

Figure 10-4 illustrates a short object reference to that is set to reference the basic object with object identifier `#7-12-132-6`. It is meaningful only as a link from a source object stored in the container with identifier `12` in database with identifier `7`.

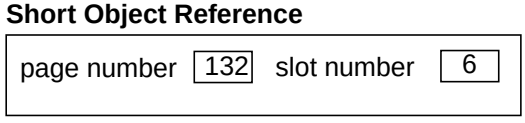


Figure 10-4 Short Object Reference

Short object references can be stored by attribute data members and by short inline associations. In either case, the destination object of the link is assumed to be located in the same container as the source object.

Short Object-Reference Classes

Short object references are instances of the parameterized classes `ooShortRef(`*`className`*`)`, where *`className`* is the name of a basic-object class. An instance of `ooShortRef(`*`className`*) can reference an instance of *`className`* or any of its derived classes.

The Objectivity/C++ programming interface includes a short object-reference class for `ooObj` and every predefined basic-object class. For example, a short object reference of the class `ooShortRef(ooTreeList)` references an instance of the Objectivity/C++ list class `ooTreeList`.

When the DDL processor encounters the definition of a basic-object class, it generates the definition of the corresponding short object-reference class. For example, if a DDL file contains the definition of a basic-object class `Library`, the DDL processor generates the corresponding object-reference class `ooShortRef(Library)`.

Working With a Short Object Reference

A short object reference is a truncated persistent address that links a source object to a destination basic object, provided that both objects reside in the same container. Unlike standard object references or handles, a short object reference cannot be used for operating on or accessing the members of the basic object it references.

Obtaining a Short Object-Reference Class Definition

A source file obtains the definition of `ooShortRef(`*`className`*`)` from the same header file that contains the definition of `ooHandle(`*`className`*`)`; see “Obtaining a Handle Class Definition” on page 217.

Creating a Short Object Reference

A short object reference is normally created implicitly:

- If a persistence-capable class defines an attribute data member whose type is a short object-reference class, a short object reference is created when you create an instance of the containing class.
- If a persistence-capable class defines a short inline association, a short object reference to the destination object is created and stored in the source object when you create a link for the association.

If you use a short object reference as a variable in a function, you should create it as a local variable on the stack, rather than allocating it on the heap. You may

declare a short object reference as `const`. The following definition creates a null short object reference called `temp` that can be set to any basic object:

```
ooShortRef(ooObj) temp;
```

Setting a Short Object Reference

You can set a short object reference to a basic object in either of the following ways:

- By assignment or initialization from a standard object reference or handle. Only the lower half of the object identifier is assigned to the short object reference; the upper half is ignored.
- By assignment or initialization from another short object reference.

It is the application's responsibility to ensure that a short object reference is set to a basic object that resides in an appropriate container.

Testing a Short Object Reference

You can test a short object reference as you would a handle; see "Testing a Handle" on page 219 for information about testing whether a short object reference is null, valid, or equal to another object reference or a handle.

Operating on the Referenced Object

A short object reference has no member functions for accessing or operating on the referenced object. If you want to operate on an object that is referenced by a short object reference, you must first assign the short object reference to a handle. To do so, you:

1. Prepare the handle by calling its `set_container` member function to specify the container in which the referenced object resides.
2. Assign the short object reference to the prepared handle.

11

Persistent Collections

Objectivity/C++ provides a variety of classes for creating *persistent collections*. A persistent collection is an aggregate persistent object that contains a variable number of elements; you can use a persistent collection to organize a large number of persistent objects for fast retrieval.

This chapter describes:

- General information about persistent collections
- Referential integrity of a collection
- Building a persistent collection
- Properties of a collection
- Application-defined comparator classes for scalable persistent collections

Understanding Persistent Collections

Persistent 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).

Scalability

A *scalable* persistent collection organizes its elements in segments that can be accessed, resized, and clustered independently of each other. This enables a scalable collection to increase in size—up to millions of elements—with minimal performance degradation.

A *nonscalable* persistent collection is appropriate for smaller numbers of elements (up to about 10,000 elements) because it must fit entirely in memory when accessed or resized. Furthermore, a nonscalable collection and all its internal data structures reside within a single container, reducing concurrent access to it (although the elements in the collection may themselves be clustered in other containers).

Objectivity/C++ persistent 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.

Persistent collection classes generally allow you to customize their comparison and hashing mechanisms, as applicable. If you do this for a persistent collection that is to be accessed by different applications, equivalent customizations must be made in all of the accessing applications. The accessing applications can be written in any Objectivity/DB programming interface (for example, some in C++, some in Java, and some in Smalltalk).

Element Structure

Persistent collections are named according to the structure of their elements:

- *Lists* and *sets* are persistent collections whose elements are individual persistent objects. List elements are always ordered and may include duplicates or null elements; set elements may be either unordered or sorted and may not include duplicates or null elements.
- *Maps* (sometimes called *dictionaries*) are persistent collections whose elements are key-value pairs. The values in these pairs are persistent objects; the keys may be either strings (*name maps*) or persistent objects (*object maps*). Map elements can be unordered or sorted.

A persistent collection is linked to the persistent objects it contains (its elements, keys, and/or values) by object references.

Summary of Persistent-Collection Classes

Objectivity/C++ provides the persistence-capable collection classes listed in the following table.

Class	Used for	Description
ooHashSet	Set	Scalable unordered collection of persistent objects with no duplicates and no null elements.
ooTreeSet	Sorted set	Scalable sorted collection of persistent objects with no duplicates and no null elements.
ooTreeList	List	Scalable ordered collection of persistent objects that can contain duplicates and 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.

Your application must include the `ooCollections.h` header file to use scalable persistent-collection classes, and the `ooMap.h` header file to use the name-map class. For UNIX linking information, see *Installation and Platform Notes for UNIX*.

Referential Integrity of a Collection

Referential integrity is a characteristic of a persistent collection that ensures that the collection has object references only to objects that actually exist. Maintaining referential integrity requires that, when an object is deleted, any object reference from a persistent collection to the deleted object is removed. Name maps can maintain referential integrity automatically; other kinds of persistent 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 `set_refEnable` member function 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.

Sets, Lists, and Object Maps

Sets, lists, and object maps do not maintain referential integrity. Before you delete an object from the federated database, you are responsible for removing it from any set, list, or object maps to which it belongs.

You can restore the referential integrity of any of these collections by calling its `removeAllDeleted` member function. If the collection contains any persistent objects that have been deleted, that member function removes the deleted objects from the collection.

Building a Persistent Collection

To build a persistent collection, you create a persistent instance of the appropriate class, then call member functions of the new persistent collection to add and remove the desired elements. You can make a persistent collection easy to find just as you would do for an object of any persistence-capable class. For example, you might give it a scope name, reference it from another persistent object, or use it as an element of another persistent collection. You find a persistent collection just as you would do for any persistent object; for example, by looking up its scope name or following links from another persistent object.

You must create a persistent collection during a transaction. As is the case for any basic object, you specify whether a collection is to be transient or persistent when you create it; all instances of the persistent-collection classes must be persistent. You create a collection with a call to the new operator specifying a clustering directive that indicates where in the federated database to store the new collection. You assign the pointer returned by new to a handle through which you work with the persistent collection. See “Creating a Basic Object” on page 184.

The following sections describe how to add and remove elements from persistent collections of various kinds. Chapter 16, “Individual Lookup of Persistent Objects,” explains how to find objects in a persistent collection by individual

lookup; Chapter 17, “Group Lookup of Persistent Objects,” explains how to find objects in a persistent collection by iterating over the contents of the collection.

Building a Set

A sorted set is an instance of `ooTreeSet`; an unordered set is an instance of `ooHashSet`. These classes provide the same interface for adding, removing, and testing elements.

Adding elements:

- Call a set’s `add` member function to add a persistent object. The parameter is a handle to the object to be added to the set. Because a set cannot contain duplicate elements, this member function returns `oocFalse` without modifying the set if you try to add an object that is already an element of the set.
- Call a set’s `addAll` member function to add all elements from a specified set or list, or to add all keys from a specified object map. The parameter is a handle to the scalable collection whose elements or keys are to be added to the set.

Removing elements:

- Call a set’s `remove` member function to remove a persistent object. The parameter is a handle to the object to be removed from the set.
- Call a set’s `removeAll` member function to remove all elements that are also elements of a specified set or list or that are keys of a specified object map. The parameter is a handle to the scalable collection whose elements or keys are to be removed from the set.
- Call a set’s `retainAll` member function to retain only those elements that are also elements of a specified set or list or that are keys of a specified object map, deleting all other elements. The parameter is a handle to the scalable collection whose elements or keys are to be retained in the set.
- Call a set’s `clear` member function to remove all elements it contains.

Testing elements:

- Call a set’s `contains` member function to test whether it contains a particular persistent object. The parameter is a handle to the object to be tested.
- Call a set’s `isEmpty` member function to test whether it is empty (contains no elements).
- Call a set’s `size` member function to get the number of elements it contains.

Building a List

A list is an instance of `ooTreeList`; its elements are ordered. An individual element can be specified by its index within the list. Member functions of ooTreeList allow you to add, remove, and test elements.

Adding elements:

- Call a list's `add` member function to add a persistent object, passing a handle to the object to be added to the list. One variant of this member function allows you to specify the index at which the new element is to be inserted; if you do not specify an index, the element is added to the end of the list.
- Call a list's `addFirst` member function to add a persistent object to the beginning of the list. The parameter is a handle to the object to be added to the list.
- Call a list's `addLast` member function to add a persistent object to the end of the list. The parameter is a handle to the object to be added to the list.
- Call a list's `addAll` member function to add all elements from a specified set or list, or to add all keys from a specified object map; specify a handle to the scalable collection whose elements or keys are to be added to the list. One variant of this member function allows you to specify the index at which the new elements are to be inserted; if you do not specify an index, the elements are added to the end of the list.
- Call a list's `set` member function to replace an element of the list. The parameters are the index of the element to be replaced and a handle to the object that is to replace the existing element at the specified index.

Removing elements:

- Call a list's `remove` member function to remove a persistent object. The parameter is a handle to the object to be removed from the list. If the list contains more than one occurrence of the specified object, only the first occurrence is removed.
- Call a list's `removeAll` member function to remove all elements that are also elements of specified set or list or that are keys of a specified object map. The parameter is a handle to the scalable collection whose elements or keys are to be removed from the list.
- Call a list's `retainAll` member function to retain only those elements that are also elements of a specified set or list or that are keys of a specified object map, deleting all other elements. The parameter is a handle to the scalable collection whose elements or keys are to be retained in the list.
- Call a list's `removeRange` member function to remove all elements with indexes in the specified range. The parameters are indexes of the first and last elements to be removed.
- Call list's `clear` member function to remove all elements it contains.

Testing elements:

- Call a list's `contains` member function to test whether it contains a particular persistent object. The parameter is a handle to the object to be tested.
- Call a list's `isEmpty` member function to test whether it is empty (contains no elements).
- Call a list's `size` member function to get the number of elements it contains.

Building a Name Map

A name map is an instance of `ooMap`; its elements are key-value pairs in which the key is a string (or name) and the value is a persistent object.

Adding elements:

- You can add a persistent object to a name map by calling its `add` or `forceAdd` member function. Both member functions take as parameters the proposed name and an object reference to the object to be named; they differ in their behavior when the proposed name is already a key in the map.
  - Call a name map's `add` member function to add an element with a name that is not already a key. The parameters are the name and an object reference to the object to be named.

    Note: This member function signals an error if the name map already contains an element whose key is the specified name. You can call the name map's `isMember` member function to test whether it already contains an element with a particular name.
  - Call a name map's `forceAdd` member function to add an element even if its name is a duplicate. The parameters are the name and an object reference to the object to be named.

    Note: You should call this member function only when you are certain that the name is not already a key; if you add more than one element with the same key, it is indeterminate which element would be found when you look up the key or when you replace or remove the element with the key.

  The `forceAdd` member function is faster than `add` and can be used to initialize name maps when they are created. You can then use `add` and `replace` methods for maintenance.
- Call a name map's `replace` member function to replace the persistent object that is paired with a particular name. The parameters are the name and an object reference to the object to be paired with that name. If the name map does not already contain an element with the specified name, this member function adds a new element to the map.

Removing elements:

- Call a name map's `remove` member function to remove a named object. The parameter is the name of the element to be removed.

Testing elements:

- Call a name map's `nElement` member function to get the number of elements it contains.

Building an Object Map

A sorted object map is an instance of `ooTreeMap`; an unordered object map is an instance of `ooHashMap`. The elements of an object map are key-value pairs in which both the key and the value are persistent objects. The two object-map classes provide the same interface for adding, removing, and testing elements.

Adding elements:

- Call an object map's `put` member function to add an element. The first parameter is a handle to the key for the element to be added; the second parameter is a handle to the value.
- Call an object map's `add` member function to add an element with the specified key and a null value. The parameter is a handle to the key.
- Call an object map's `addAll` member function, passing a handle to an object map, to add all elements from the specified object map. Call `addAll`, passing a handle to a set or list, to add each element of that collection as a key paired with a null value.

Removing elements:

- Call an object map's `remove` member function to remove the element with the specified key. The parameter is a handle to the key.
- Call an object map's `removeAll` member function to remove elements whose keys are also keys of a specified object map or that are elements of a specified set or list. The parameter is a handle to the scalable collection that indicates which elements should be removed from the object map.
- Call an object map's `retainAll` member function to retain only those elements whose keys are also keys of a specified object map or are elements of a specified set or list, deleting all other elements. The parameter is a handle to the scalable collection that indicates which elements should be retained in the object map.
- Call an object map's `clear` member function to remove all elements it contains.

Testing elements:

- Call an object map's `containsKey` member function to test whether it contains a particular persistent object as the key of some element. The parameter is a handle to the object to be tested.
- Call an object map's `containsValue` member function to test whether it contains a particular persistent object as the value of some element. The parameter is a handle to the object to be tested.

- Call an object map's `isEmpty` member function to test whether it is empty (contains no elements).
- Call an object map's `size` member function to get the number of elements it contains.

Whenever you add an element with a specified key, if the object map already contains an element with that key, the value paired with that key is replaced; otherwise, a new element is added.

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/C++ 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.

Growth Characteristics

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.

Hash Function

A name map hashes on the string keys of its elements. All name maps that the application accesses use the same hash function. If desired, an application can use its own hash function for name maps.

A hash function is an application-defined function that must conform to the calling interface defined by the ooNameHashFuncPtr function pointer type. The function takes two parameters: the string from which to compute the hash value, and the number of bins in the hash table. It returns the hash value for the specified string, which must be between 0 and one less than the number of bins.

If you want to use an application-defined hashing function, you can install the desired function by calling the static member function ooMap::set_nameHashFunction, passing a function pointer to the hashing function as the parameter.

EXAMPLE The function myNameHash is an application-defined hash function for name maps.

```
// Application code file
#include "myClasses.h"   // DDL file myClasses.ddl
                         // includes <ooMap.h>
...
uint32 myNameHash(const char *name, const uint32 modulus)
{
// This function should return a value between 0 and modulus-1
   ...
}
```

The following statement installs the function myNameHash as the application's hash function for name maps.

```
ooMap::set_nameHashFunction(myNameHash);
```

All applications that access a given name map must use the same hashing function. If the applications that use a given name map are all implemented in C++, they can all share the definition of the hashing function. If some applications are written in C++ and some in Smalltalk, their hashing function must use equivalent hashing algorithms.

WARNING Java applications cannot replace the default hashing function for name maps. If your application needs to interoperate with Java applications, you must not replace the default hashing function for name maps.

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 leaf node in the B-tree locates a disjoint group of elements whose indexes are within a certain range.

Every node in the B-tree has a corresponding array. The array for a nonleaf node contains object references to the first leaf-node descendants along each branch from the node; the array for a leaf node contains object 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.

The B-tree for 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 object references in a node's 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 object references as will fit on a single storage 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. See "Assigning Basic Objects to Containers" on page 131.

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. Initially, a collection's container is its current 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 non-garbage-collectible container, all its node containers are non-garbage-collectible. If the collection is clustered in a garbage-collectible container, all its node containers are garbage-collectible. Objectivity/C++ applications typically use non-garbage-collectable containers; garbage-collectible containers are provided for interoperability with Objectivity for Java or Objectivity/Smalltalk applications. See “Kinds of Container” on page 171.

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/C++ creates the array for the collection’s root B-tree node. By default, Objectivity/C++ 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 non-garbage-collectible container, its initial array container is non-garbage-collectible; if the collection is clustered in a garbage-collectible container, its initial array container is 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 non-garbage-collectible, all subsequent array containers will be non-garbage-collectible; if the initial array container is garbage collectible, all subsequent array containers will be 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 uses a persistent object, called a *tree administrator*, an instance of `ooTreeAdmin`, 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 pass a handle to that container as a parameter to the constructor that creates the ordered collection. For example, instead of having Objectivity/C++ 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.

Like other persistent objects, tree administrators are normally manipulated through handles or object references. You can call the `admin` member function on an ordered collection to obtain an object reference to its tree administrator.

A tree administrator has two properties that you can set to control when the ordered collection's current node container and the current array container are considered "full."

- The *maximum nodes per container* property specifies how many B-tree nodes can be clustered together in the same container. Because B-tree nodes are small objects, many of them can fit on a single storage 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 storage page size; it is calculated as:

  `pageSize / 47`

  To use a different value for this property, call the tree administrator's `setMaxNodesPerContainer` member function.

  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 storage 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 for the collection's tree administrator. To use a different value for this property,

call the tree administrator's `setMaxVArraysPerContainer` member function.

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 a *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.
- The default comparator for a sorted object map sorts elements by increasing object identifier of their keys.

You can implement an alternative sorting criteria with an application-defined comparator class. See "Comparator Class for Sorted Collections" on page 257.

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 268.

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 lookup in the top-level directory and one lookup 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 and, 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 131.

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 255.

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 non-garbage-collectible container, all its hash-bucket containers are non-garbage-collectible; if the collection is clustered in a garbage-collectible container, all its hash-bucket containers are garbage-collectible. Objectivity/C++ applications typically use non-garbage-collectable containers; garbage-collectible containers are provided for interoperability with Objectivity for Java or Objectivity/Smalltalk applications. See "Kinds of Container" on page 171.

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 uses a persistent object, called a *hash administrator*, an instance of `ooHashAdmin`, to manage the containers for the collection's hash buckets. 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.

Like other persistent objects, hash administrators are normally manipulated through handles or object references. You can call the `admin` member function on an unordered collection to obtain an object reference to its hash administrator.

A hash administrator has a *maximum buckets per container* property, which 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 by a single user, locking is not an issue. In that case, a larger value may be appropriate for the collection's hash administrator. To use a different value for this property, call the hash administrator's `setMaxBucketsPerContainer` member function.

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 hash buckets will be clustered in each.

Comparator

Every scalable unordered collection has a *comparator* that controls how an element's hash value is computed and to test elements for equality.

- The default comparator for an unordered set computes an element's hash value from its object identifier. Two elements are equal if their object identifiers are equal.
- The default comparator for an unordered object map computes an element's hash value from the object identifier of its key. Two elements are equal if the object identifiers of their keys are equal.

You can implement an alternative hashing algorithm with an application-defined comparator class. See "Comparator Class for Unordered Collections" on page 262.

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 268.

Application-Defined Comparator Classes

A *comparator* is a persistent object of a concrete derived class of `ooCompare`. It provides a comparison function for ordering elements of scalable sorted collections and a hashing function for computing the hash values for elements of scalable unordered collections. (Lists and name maps do not use comparators.)

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` member functions as appropriate.

An application-defined comparator can be defined to support content-based lookup—that is, you can find objects based on the persistent data in the element of a set or in the key of an object map. This ability enables individual lookup of objects in a set and enhances individual lookup in an object map; see Chapter 16, "Individual Lookup of Persistent Objects," for more information.

NOTE You should use an application-defined comparator only when your application *requires* the functionality it provides (such as the ability to perform content-based lookup). Using such a comparator has a performance overhead—most operations on a collection are slower if the collection uses an application-defined comparator.

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 attributes of each element or key. You might additionally want to use the sorting attributes to identify elements of the collection so that you can look up the element or key with particular values in its identifying attributes.

Comparing Elements of a Sorted Collection

Elements in a sorted collection are ordered based on the sorting criteria embodied in the `compare` member function of its comparator. The first variant of that member function compares two persistent objects and indicates their relative order in the collection; the second variant compares a persistent object to identifying data for another persistent object and indicates the relative order in the collection of the two objects.

When you define a comparator class to be used with sorted collections, you can override the first variant of the `compare` member function to compare two persistent objects based on whatever sorting criteria you choose. Typically, the comparison uses attribute values to sort the objects. The compare function should return a negative integer if the first object is less than (sorts before) the second; zero if the two objects are equal; and a positive integer if the first object is greater than (sorts after) the second.

You can override the second variant of `compare` to call the first variant, assuming that second parameter is a pointer to a persistent object. Alternatively, you could implement the second variant to support content-based lookup as described in “Supporting Content-Based Lookup in a Sorted Collection” on page 260.

NOTE Parameters to the `compare` member function are `const` handles. Because any operation that accesses the referenced object through a handle changes its internal state, *you must cast the parameters to non-`const` handles* and use the non-`const` handles to access the objects being compared.

The `compare` member function must impose a total ordering on objects; that is, it must indicate that two different objects are different even if they have the same values for the attributes used to sort them. As a consequence, unless different objects are guaranteed to have different combinations of values for the attributes used in sorting, the overriding `compare` function should call the inherited `compare` function for objects that have the same attribute values. The inherited function compares the objects' object identifiers (OIDs), so it finds the objects equal only if they have the same OIDs. If two objects have the same combination of attribute values, they are ordered by their OIDs.

EXAMPLE The comparator class `CompNames` compares two `Person` objects, sorting by last name, then first name. A comparator of this class could be used by a sorted set of `Person` objects or by a sorted object map whose keys are `Person` objects.

Because it is possible for two different `Person` objects to have the same first and last names, when the `CompNames::compare` function finds that both objects have the same names, it calls the inherited `ooCompare::compare` function to compare their OIDs.

If the `CompNames::compare` function cannot compare the two objects (because they are not `Person` objects or because they cannot be opened), it returns -999 by convention. Note that this value does not indicate an error condition to Objectivity/C++; it simply indicates that the two objects are different and the first one sorts before the second.

```
// DDL file person.ddl
#include <ooCollection.h>
...
class Person : public ooObj {
public:
   ooVString lastName;
   ooVString firstName;
   ...;
};
class CompNames : public ooCompare {
public:
   virtual int compare (const ooHandle(ooObj) &obj1H,
                        const ooHandle(ooObj) &obj2H) const;
   virtual int compare (const ooHandle(ooObj) &obj1H,
                        const void *&lookupVal) const;
};
```

```
// Application code file
#include "person.h"
…
// Compare two Person objects
int CompNames::compare (const ooHandle(ooObj) &obj1H,
                        const ooHandle(ooObj) &obj2H) const {
   ooHandle(Person) &pers1H;, &pers2H
   // Cast obj1H to a non-const handle before accessing the
   // referenced object
   ooHandle(ooObj) &objH = const_cast<ooHandle(ooObj)&>(obj1H);
   if (objH.open() == oocError)
      return -999;
   if (objH->ooIsKindOf(ooTypeN(Person)))
      pers1H = static_cast<ooHandle(Person)&>(objH);
   else
      return -999;
   // Cast obj2H to a non-const handle before accessing the
   // referenced object
   objH = const_cast<ooHandle(ooObj)&>(obj2H);
   if (objH.open() == oocError)
      return -999;
   if (objH->ooIsKindOf(ooTypeN(Person)))
      pers2H = static_cast<ooHandle(Person)&>(objH);
   else
      return -999;
   ooVString *s1 = pers1H->lastName;
   ooVString *s2 = pers2H->lastName;
   if (s1 < s2)
      return -1;
   else if (s1 > s2)
      return 1;
   else {       // s1 = s2; objects have same last name
      ooVString *s1 = pers1H->firstName;
      ooVString *s2 = pers2H->firstName;
      if (s1 < s2)
         return -1;
      else if (s1 > s2)
         return 1;
      else {    // s1 = s2; objects also have same first name
         // Compare the objects by their OIDs
         return ooCompare::compare(obj1H, obj2H)
      }
   } // End else objects have same last name
} // End compare function - variant 1
```

```
// Compare a Person object to identifying data; assume
// that lookupVal points to a Person object
int CompAccount::compare (const ooHandle(ooObj) &obj1H,
                          const void *&lookupVal) const {
   // Cast the pointer to a handle
   const ooHandle(ooObj) &obj2H =
      static_cast<const ooHandle(ooObj) &>(lookupVal);
   // Compare the two objects
   return compare(obj1H, obj2H);
} // End compare function - variant 2
```

Supporting Content-Based Lookup in a Sorted Collection

A comparator class for sorted collections can optionally provide the ability to identify an element or key based on the attribute values that are the comparator's 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.

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 second variant of the `compare` member function to compare an element or key with data that specifies the sorting criteria for an element or key.

EXAMPLE The comparator class `CompAccount` compares objects of the class `Client` and its derived classes. The class `Client` represents a company's client companies; the `accountNo` attribute of this class is the client's unique account number.

A comparator of class `CompAccount` could be used by a sorted set of `Client` objects or by a sorted object map whose keys are `Client` objects. It enables the collection to find a `Client` object by looking up its account number. The example on page 350 illustrates its use with a sorted object map.

```
// DDL file company.ddl
#include <ooCollection.h>
...
class Client : public ooObj {
public:
   ooVString companyName;
   uint32 accountNo;
   ...;
};
```

```
class CompAccount : public ooCompare {
public:
   virtual int compare (const ooHandle(ooObj) &obj1H,
                        const ooHandle(ooObj) &obj2H) const;
   virtual int compare (const ooHandle(ooObj) &obj1H,
                        const void *&lookupVal) const;
};
```

```
// Application code file
#include "company.h"
…
// Compare two Client objects, sorting by accountNo
int CompAccount::compare (const ooHandle(ooObj) &obj1H,
                          const ooHandle(ooObj) &obj2H) const {
   ooHandle(Client) &clientH;
   // Cast obj1H to a non-const handle before accessing the
   // referenced object
   ooHandle(ooObj) &objH = const_cast<ooHandle(ooObj)&>(obj1H);
   if (objH.open() == oocError)
      return -999;
   if (objH->ooIsKindOf(ooTypeN(Client))) {
      clientH = static_cast<ooHandle(Client)&>(objH);
      uint32 v1 = clientH->accountNo;
   }
   else
      return -999;
   // Cast obj2H to a non-const handle before accessing the
   // referenced object
   objH = const_cast<ooHandle(ooObj)&>(obj2H);
   if (objH.open() == oocError)
      return -999;
   if (objH->ooIsKindOf(ooTypeN(Client))) {
      clientH = static_cast<ooHandle(Client)&>(objH);
      uint32 v2 = clientH->accountNo;
   }
   else
      return -999;
   if (v1 < v2)
      return -1;
   else if (v1 == v2)
      return 0;
   else               // v1 > v2
      return 1:
} // End compare function - variant 1
```

```
// Compare a Client object to the specified account number
int CompAccount::compare (const ooHandle(ooObj) &obj1H,
                          const void *&lookupVal) const {
   ooHandle(Client) &clientH;
   // Cast obj1H to a non-const handle before accessing the
   // referenced object
   ooHandle(ooObj) &objH = const_cast<ooHandle(ooObj)&>(obj1H);
   if (objH.open() == oocError)
      return -999;
   if (objH->ooIsKindOf(ooTypeN(Client)))
      clientH = static_cast<ooHandle(Client)&>(objH);
   else
      return -999;
   uint32 v1 = clientH->accountNo;
   uint32 &v2 = *lookupVal;
   if (v1 < v2)
      return -1;
   else if (v1 == v2)
      return 0;
   else               // v1 > v2
      return 1;
} // End compare function - variant 2
```

For simplicity, the preceding example uses a single numeric attribute to identify a `Client` object. In general, however, a comparator can use any number of attributes to identify the objects, and the attributes can be of any data types.

If the comparator uses a combination of attributes to identify an object, the application must pack the desired values for those attributes together in some way to form the identifying data; it must pass a pointer to that data to the collection member functions that identify an element or a key by its component data. For example, the application might create a transient `Client` object with its identifying attributes set to the desired lookup values; the `compare` function would compare the identifying attributes of this transient object with the identifying attributes of the object being tested. Alternatively, the application might create a `struct` whose fields contain the values for the various identifying attributes; the `compare` function would compare each field of the `struct` to the corresponding attribute of the object being tested.

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 attributes of each element or key. You might additionally want to use the hashing

attributes to identify elements of the collection so that you can look up the element or key with particular value(s) in its identifying attributes.

Hashing Elements of an Unordered Collection

An unordered collection computes hash values for its elements by calling the `hash` member function of its comparator. The first variant of that member function computes the hash value for a specified persistent object; the second variant computes the hash value for the persistent object with the specified identifying data.

When you define a comparator class to be used with unordered collections, you can override the first variant of the `hash` member function to compute hash values for persistent objects using whatever criteria or algorithm you choose.

NOTE The parameter to the `hash` member function is a `const` handle. Because any operation that accesses the referenced object through a handle changes its internal state, you must cast the parameter to a non-`const` handle and use the non-`const` handle to access the objects being hashed.

Your `hash` member function 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.

You can override the second variant of `hash` to call the first variant, assuming its parameter is a pointer to a persistent object. Alternatively, you could implement the second variant to support content-based lookup as described in “Supporting Content-Based Lookup in an Unordered Collection” on page 266.

You typically also override the `compare` member function to compare two persistent objects for equality based on the same criteria that are used to hash them.

EXAMPLE The comparator class `CompSSN` hashes objects of the class `Employee` and its derived classes. The class `Employee` represents people employed by a particular company in the United States; the `SSN` attribute of this class is a string representation of employee’s Social Security Number (SSN). The `hash` member function of `CompSSN` verifies that its parameter is a handle to an `Employee` object and, if so, converts the SSN string to a 32-bit integer to be used as the hash value.

The `compare` member function of `CompSSN` compares the social security numbers of two objects.

A comparator of class `CompSSN` could be used by an unordered set of `Employee` objects or by an unordered object map whose keys are `Employee` objects.

```
// DDL file company.ddl
#include <ooCollection.h>
…
class Employee : public ooObj {
public:
   ooVString SSN;
   …;
};
class CompSSN : public ooCompare {
public:
   virtual int hash (const ooHandle(ooObj) &objH) const;
   virtual int hash (const void *&lookupVal) const;
   virtual int compare (const ooHandle(ooObj) &ob11jH,
                        const ooHandle(ooObj) &obj2H) const;
   virtual int compare (const ooHandle(ooObj) &obj1H,
                        const void *&lookupVal) const;
};
```

```
// Application code file
#include "company.h"
…
// Utility function to convert a string SSN to integer
int ssnStringToInt(ooVString ssn) {
   int s1, s2, s2;
   const char *str = ssn;
   sscanf(str, "%3d-%2d-%4d", &s1, &s2, &s3);
   return s1 * 1000000 + s2 * 10000 + s3;
}

// Compute a hash value from Employee's SSN
int CompSSN::hash(const ooHandle(ooObj) &objH) const {
   ooHandle(Employee) &empH;
   // Cast objH to a non-const handle before accessing the
   // referenced object
   ooHandle(ooObj) &objH = const_cast<ooHandle(ooObj)&>(objH);
   if (objH.open() == oocError)
      return 0;
   if (objH->ooIsKindOf(ooTypeN(Employee))) {
      empH = static_cast<ooHandle(Employee)&>(objH);
      return ssnStringToInt(empH->SSN);
   }
```

```
   else
      return 0;
} // End hash function - variant 1

// Compute a hash value from Employee's identifying data;
// assume that lookupVal points to an Employee object
int CompSSN::hash(const void *&lookupVal) const {
   // Cast the pointer to a handle
   const ooHandle(ooObj) &objH =
      static_cast<const ooHandle(ooObj) &>(lookupVal);
   // Hash the object
   return hash(objH);
} // End hash function - variant 2

// Compare two Employees by SSN
int CompSSN::compare (const ooHandle(ooObj) &obj1H,
                      const ooHandle(ooObj) &obj2H) const {
   ooHandle(Employee) &empH;
   // Cast objH to a non-const handle before accessing the
   // referenced object
   ooHandle(ooObj) &objH = const_cast<ooHandle(ooObj)&>(obj1H);
   if (objH.open() == oocError)
      return -999;
   if (objH->ooIsKindOf(ooTypeN(Employee))) {
      empH = static_cast<ooHandle(Employee)&>(objH);
      const char *v1 = empH->accountNo;
   }
   else {   // Can't compare
      return -999;

   // Cast objH to a non-const handle before accessing the
   // referenced object
   objH = const_cast<ooHandle(ooObj)&>(obj2H);
   if (objH.open() == oocError)
      return -999;
   if (objH->ooIsKindOf(ooTypeN(Employee))) {
      empH = static_cast<ooHandle(Employee)&>(objH);
      const char *v2 = empH->accountNo;
   }
   else {   // Can't compare
      return -999;
   return strcmp(v1, v2);
} // End compare function - variant 1
```

```
// Compare an Employee object to identifying data; assume
// that lookupVal points to an Employee object
int CompSSN::compare (const ooHandle(ooObj) &obj1H,
                          const void *&lookupVal) const {
   // Cast the pointer to a handle
   const ooHandle(ooObj) &obj2H =
      static_cast<const ooHandle(ooObj) &>(lookupVal);
   // Compare the two objects
   return compare(obj1H, obj2H);
} // End compare function - variant 2
```

Supporting Content-Based Lookup in an Unordered Collection

A comparator class for unordered collections can optionally provide the ability to identify an element or key based on the attribute values 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 and test whether a collection contains that element.

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:

- Implement the second variant of the `hash` member function to compute a hash value from data that identifies the persistent object.
- Implement the second variant of the `compare` member function to compare an element or key of the unordered collection with data that identifies a persistent object.

EXAMPLE In this example, the comparator class `CompSSN` has been modified to identify an object of the class `Employee` based on its `SSN` attribute. The example on page 343 illustrates its use with an unordered set.

```
// DDL file company.ddl
#include <ooCollection.h>
...
class CompSSN : public ooCompare {
public:
   virtual int hash (const ooHandle(ooObj) &objH) const;
   virtual int hash (const void *&lookupVal) const;
   virtual int compare (const ooHandle(ooObj) &ob11jH,
                        const ooHandle(ooObj) &obj2H) const;
   virtual int compare (const ooHandle(ooObj) &obj1H,
                        const void *&lookupVal) const;
};
```

```
// Application code file
#include "company.h"
…
// Utility function to convert a string SSN to integer
int ssnStringToInt(ooVString ssn) {
   … // See page 264
}
// Compute a hash value from Employee's SSN
int CompSSN::hash (const ooHandle(ooObj) &objH) const {
   … // See page 264
} // End hash function - variant 1

// Compute a hash value from string containing SSN
int CompSSN::hash (const void *&lookupVal) const {
   return ssnStringToInt(*lookupVal);
} // End hash function - variant 2

// Compare two Clients by SSN
int CompSSN::compare (const ooHandle(ooObj) &obj1H,
                      const ooHandle(ooObj) &obj2H) const {

   … // See page 265
} // End compare function - variant 1

// Compare a Client to the specified SSN string
int CompSSN::compare (const ooHandle(ooObj) &obj1H,
                      const void *&lookupVal) const {
   ooHandle(Employee) &empH;
   // Cast obj1H to a non-const handle before accessing the
   // referenced object
   ooHandle(ooObj) &objH = const_cast<ooHandle(ooObj)&>(obj1H);
   if (objH.open() == oocError)
      return -999;
   if (objH->ooIsKindOf(ooTypeN(Employee))) {
      empH = static_cast<ooHandle(Employee)&>(objH);
      const char *v1 = empH->accountNo;
      const char *v2 = lookupVal;
      return strcmp(v1, v2);
   }
   else     // Can't compare
      return -999;
} // End compare function - variant 2
```

For simplicity, the preceding example uses a single string attribute to identify an `Employee` object. In general, however, a comparator can use any number of attributes of any data types to identify the objects.

If the comparator uses a combination of attributes to identify an object, the application must pack the desired values for those attributes together in some way to form the identifying data; it must pass a pointer to that data to the collection member functions that identify an element or a key by its component data. The `compare` or `hash` functions must unpack the attribute values appropriately. For example, the application might create a transient `Employee` object with its identifying attributes set to the desired lookup values; the `compare` function would compare the identifying attributes of this transient object with the identifying attributes of the object being tested; the `hash` function would compute a hash value from the identifying attributes of the transient object. Alternatively, the application might create a `struct` whose fields contain the values for the various identifying attributes. The `compare` function would compare each field of the `struct` to the corresponding attribute of the object being compared; the `hash` function would compute a hash value from the fields as if they were values of the corresponding attributes.

Using a Comparator

To use a comparator of an application-defined comparator class, you create it and assign it to one or more scalable collections. Special care may be required when modifying objects in the collection; see “Modifying Objects in the Collection” on page 269.

Creating a Comparator

To create a comparator, instantiate your comparator class. As is the case for any basic object, you specify whether a comparator is to be transient or persistent when you create it; comparators *must be persistent*. You create a comparator with a call to the `new` operator; the clustering directive in that call specifies where in the federated database to store the new comparator. You assign the pointer returned by `new` to a handle through which you work with the comparator. See “Creating a Basic Object” on page 184.

A comparator is locked whenever you access the collection that uses it. 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 persistent collection or reference a comparator in an attribute of a persistent object. Typically, an application uses only comparators that it creates dynamically; it does not explicitly look up a particular comparator in the database.

Assigning a Comparator to a Collection

After creating the comparator, you can assign it to any collections that need to use the comparator's particular comparison and hashing algorithms. You assign a comparator to a collection by passing a handle to 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.

EXAMPLE This example creates an unordered set that uses a comparator of the `CompSSN` class, which is defined on page 263.

```
// Application code file
#include "company.h"
...
ooTrans trans;
ooHandle(ooContObj) compContH;
ooHandle(ooContObj) setContH;
ooHandle(ooCompare) compH;
ooHandle(ooHashSet) setH;
...
trans.start();
...     // Open the federated database for update
...     // Set compContH to reference the container where the
        // comparator will be stored
// Create the comparator
compH = new(compContH) CompSSN();
...     // Set setContH to reference the container where the
        // unordered set will be stored
// Create the set, assigning the comparator to it
setH = new(setContH) ooHashSet(compH);
...
trans.commit();
```

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 data member 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 can 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 data member 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 can add the element back to the collection, which will assign it the (new) correct hash value.

Comparators and Interoperability

If a persistent collection uses a comparator of an application-defined class, the data for the collection in the federated database includes an object reference to the comparator. Any application that finds the collection will also find its comparator. As a consequence, any application that finds the comparator must include a comparator class with the same name as the comparator's class.

Objectivity / DB provides persistent storage for data only, not for member functions, so the federated database does not store `compare` and `hash` member functions of the comparator. The comparator class in the retrieving application must include implementations for those member functions; furthermore, those member functions 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` member functions for the collection's comparator.

If the applications that use a given persistent collection are all implemented in the same language (for example, C++), they can all share the definition of the comparator class. If the applications are written in different languages (for example, some in C++ and some in Java), their comparator classes must use equivalent comparison and hashing algorithms.

12

Variable-Size Arrays

A *variable-size array* (VArray) is similar to a C++ array, except that its size can be changed dynamically. The *size* of a VArray is the number of elements it contains. Persistent objects can contain VArrays as attributes. In addition, an Objectivity/C++ application can work with temporary VArrays that are never stored persistently.

This chapter describes:

- General information about VArrays
- Creating VArrays
- Working with VArrays: getting and setting VArray elements, assigning a VArray, and managing the size of a VArray
- Java-compatibility classes for variable-size arrays

Understanding VArrays

VArrays are instances of classes created from either of the following template classes:

- `ooVArrayT<`*`element_type`*`>`
- `ooTVArrayT<`*`element_type`*`>`

The *`element_type`* parameter specifies the type of elements in the VArray. For example, `ooVArrayT<Point>` is a class representing VArrays whose elements are instances of the non-persistence-capable class `Point`.

NOTE For backward compatibility, you can use the macro-style names `ooVArray(`*`element_type`*`)` and `ooTVArray(`*`element_type`*`)` instead of the corresponding template names `ooVArrayT<`*`element_type`*`>` and `ooTVArrayT<`*`element_type`*`>`.

Standard and Temporary VArrays

Objectivity/C++ supports two kinds of VArrays:

- *Standard VArrays*, which are instances of the `ooVArrayT<element_type>` classes.
- *Temporary VArrays*, which are instances of the `ooTVArrayT<element_type>` classes.

Standard and temporary VArrays define the same member functions. However:

- Only a standard VArray can be saved persistently—for example, as an embedded data member of a persistent object. See "VArrays and Persistence" on page 273.
- Only a temporary VArray can have handle or iterator elements, or, more generally, elements that contain memory pointers to other elements; see "A Closer Look at Resizing" on page 279.

VArray Elements

Elements of VArrays can be of most types, including non-persistence-capable class types. However:

- VArrays cannot contain other VArrays, either directly or indirectly.
- Elements of a standard VArray may not contain memory pointers to other elements. You must use a temporary VArray for such elements, including handles or iterators; see "A Closer Look at Resizing" on page 279.
- Elements of a standard VArray must be a primitive type, an object-reference type, or an embedded-class type. See "Defining Data Members" in Chapter 2 in the Objectivity/C++ Data Definition Language book.

The *element_type* of every VArray must have a default constructor (a constructor that can take no parameters).

When one VArray is set from another VArray using either the assignment operator (=) or the copy constructor, an element-by-element copy is performed. Each element is set using *element_type* assignment. If the default compiler-generated assignment operator for *element_type* is not sufficient for copying elements of an embedded-class type, you should define the appropriate assignment operator for that class.

VArray Structure

A VArray is a compound object consisting of a reference to a vector of elements. The reference portion of the VArray occupies a fixed amount of space; the vector portion occupies a variable amount of space and may be relocated by certain operations. Elements in the vector are guaranteed contiguous within virtual memory.

A VArray with 0 elements has no vector allocated for it. A vector is allocated when elements are added; resizing the VArray dynamically grows or truncates the vector. Resizing a VArray to 0 elements deallocates the vector.

You access each element by its position in the VArray. Elements are numbered starting with 0; the position number is the element's *index* or *subscript*. Because of the way VArrays are represented, you must use access member functions to get the first element of an array (the element whose index is 0); a dereferencing expression such as `*myArray` does not access the first element of `myArray`.

VArrays and Persistence

A standard VArray is transient unless you incorporate it in a persistent object, typically by embedding it as a data member of the persistent object or as a data member of one of its base classes. A VArray can be saved persistently through multiple levels of inheritance and embedding (a member of an embedded-class member of a base class, and so on). A temporary VArray cannot be incorporated in a persistent object; consequently, all temporary VArrays are transient.

When you incorporate a standard VArray in a persistent object, storage for the reference portion of the VArray is embedded in the object, and storage for the vector portion is allocated outside the object but within the same container.

Opening a persistent object allows you to get the reference portion of any member VArray, but does not automatically open the vector of elements. The vector of a persistent VArray is opened when you call a member function on the VArray.

Opening a persistent object implicitly locks that object and any member VArrays, because they are stored in the same container. If a persistent object is open (and locked) for read, operations that modify a member VArray will implicitly attempt to promote the read lock on the container to update.

Creating a VArray

To create a standard or temporary VArray, you can use one of three constructors on the appropriate template class:

- The default constructor, which creates a VArray of size 0. No element vector is allocated until you explicitly add elements; see "Resizing a VArray" on page 278.
- A constructor whose parameter is the initial (integer) number of elements.
- The copy constructor, which creates a VArray of the same size as the specified VArray and then performs an *element-by-element* copy into the new VArray. The element type's default constructor creates the new elements and

then assigns each element of the specified VArray to a corresponding element of the new VArray.

EXAMPLE Although `Point` is not persistence-capable, it is declared in a DDL file because it is used as the type of embedded-class attributes of persistence-capable classes (such as `Polygon` and `Rectangle`).

```
// DDL file geometry.ddl
typedef int32 Coord;
class Point {
   public:
      Coord x, y;
      Point() { x = y = 0; }   // Default constructor
      Point(Coord newX, Coord newY) { x = newX; y = newY; }
};
…
```

This example creates a transient instance of a standard VArray that has three `Point` elements. Note that `Point` provides a default constructor, as required.

```
// Application code
#include "geometry.h"
…
// Create a transient Point VArray
Point p1(0, 0), p2(10, 10), p3(0,20);
ooVArrayT<Point> pts(3);              // VArray constructor
pts[0] = p1;
pts[1] = p2;
pts[2] = p3;
```

EXAMPLE This example creates a temporary VArray of `Person` handles. Because `Person` is a persistence-capable class, you cannot use `Person` as the element type of a temporary or standard VArray. In addition, you cannot use a standard VArray (transient or persistent) for handle elements.

```
// Application code
#include "company.h"
…
typedef ooHandle(Employee) EmployeeH;
ooTVArrayT<EmployeeH> salesReps(); // Default VArray constructor
```

Getting Elements

You can get a VArray element by specifying its index to the subscript operator (operator[ ]) or the elem member function on the VArray.

The subscript operator verifies that the specified index falls within the VArray's current bounds. The elem member function bypasses subscript bounds checking for performance.

EXAMPLE This code fragment prints the first 10 elements of a VArray in the values data member of the persistent object referenced by the handle countH. Because the subscript operator is used, an error is signaled if values contains fewer than 10 elements.

```
// Application code file
#include "counter.h"
ooHandle(Counter) countH;
...     // Start transaction; set countH to reference a Counter
for (i=0;i < 10;i++)              // Print values
   printf("%d\n", countH->values[i]);
```

You can also use a VArray iterator to get each element of a VArray in turn. You can call a VArray's create_iterator member function to initialize a VArray iterator to get the elements of the VArray. You can then step through the elements of the VArray by calling the VArray iterator's next member function. For additional information, see "VArray Iterators" on page 309.

EXAMPLE This example computes the sum of the elements of a VArray.

```
// Application code
#include "myClasses.h"
...
ooVArrayT<uint32> nums;
...     // Set the VArray nums;
// Initialize the VArray iterator
d_Iterator<uint32> vItr = nums.create_iterator();
uint32 sum = 0;
uint32 curVal;
while (vItr.next(curVal)) {
   sum += curVal;
}
```

Setting Elements

You can set a VArray element to a value by specifying the element's index and the desired value as parameters to the VArray's set member function.

EXAMPLE This code fragment uses the set member function to set the first 10 elements of a VArray in the values data member of the persistent object referenced by the handle countH.

```
// DDL file counter.ddl
class Counter : public ooObj{
   public:
      ...
      ooVArrayT<uint32> values;
};
```

```
// Application code file
#include "counter.h"
ooHandle(Counter) countH;
...    // Start transaction; set countH to reference a Counter
int i;
for (i=0; i < 10; i++)
   countH->values.set(i, i*2);    // Set the VArray values
```

Alternatively, you can use either operator [] or the elem member function on the left side of an assignment.

When you set an element of a persistent VArray, the vector portion of the VArray must be opened in update mode, and the lock on the container upgraded to update, if necessary. These actions are performed automatically if you use the set member function. However, if you use the subscript operator or elem to set an element value, you must explicitly open the vector for update first. To do this, you call the VArray's update member function before assigning the new value.

EXAMPLE This code fragment explicitly opens a VArray in the values data member of the persistent object referenced by the handle countH, and then uses the subscript operator to set the first 10 elements of the VArray.

```
// Application code file
#include "counter.h"
ooHandle(Counter) countH;
...    // Start transaction; set countH to reference a Counter
countH->values.update();    // Open the VArray for update
```

```
for (int i=0; i < 10; i++)
   countH->values[i] = i*10;       // Set the VArray values
```

Assigning a VArray

You can assign a standard VArray to a standard VArray, or a temporary VArray to a temporary VArray. Both class templates overload the assignment operator (operator=).

The VArray on the left side of the assignment is adjusted in size to match the VArray on the right, and an element-by-element copy is performed to populate the adjusted VArray. The *element_type* default constructor creates new elements in the adjusted VArray and then each element of the VArray on the right is assigned to a corresponding element of the adjusted VArray.

EXAMPLE A Polygon object contains a standard VArray of Point objects. The Point class is shown on page 274.

```
// DDL file geometry.ddl
...
class Polygon : public ooObj {
public:
   ooVArrayT<Point> vertices;

   // Constructors
   // Default constructor; creates empty polygon
   Polygon() { }
   // Sets the number of vertices with the VArray constructor
   Polygon(uint32 size) : vertices(size) { }
   // Initializes the polygon with a VArray of points
   Polygon(ooVArrayT<Point> &varray) : vertices(varray) { }
};
```

This example uses the default Polygon constructor to create a persistent Polygon object with an empty VArray. It assigns a transient VArray to the polygon's vertices attribute. The application then uses the third Polygon constructor to create another polygon with an initialized VArray.

```
// Application code file
#include "geometry.h"
...
// Create a transient VArray for initializing persistent VArrays
Point p1(0, 0), p2(10, 10), p3(0,20);
```

```
ooVArrayT<Point> pts(3);
pts[0] = p1;
pts[1] = p2;
pts[2] = p3;

ooHandle(ooDBObj) dbH;
ooHandle(Polygon) polyH;
polyH = new(dbH) Polygon;    // Create the first polygon

// Assign the transient VArray to the empty persistent VArray
polyH->vertices = pts;

// Create the second polygon with an initialized VArray
ooHandle(Polygon) pH = new(dbH) Polygon(pts);
```

Managing VArray Size

The VArray classes provide member functions for finding and changing the size of a VArray.

Finding the Current VArray Size

You can find out the current number of elements in a VArray by calling its `size` member function.

Resizing a VArray

You can change the size of a VArray by calling its `resize` member function. You specify the VArray's new size as the parameter to the member function. The array is resized to contain the specified number of elements:

- If the new size is larger than the current size, `resize` allocates storage for the additional elements and invokes the *`element_type`* default constructor to create new, empty elements.
- If the new size is smaller than the current size, `resize` invokes the *`element_type`* destructor for the elements to be truncated (the elements from index *`newSize`* + 1 to the end) and then truncates the VArray to the new size.

If the VArray to be resized is persistent, it is implicitly opened for update, and the lock on the container is upgraded, if necessary.

A Closer Look at Resizing

A resizing operation may relocate the vector portion of a VArray to keep the elements contiguous in virtual memory. This relocation is performed differently by each kind of VArray, which determines whether the VArray elements can contain pointers to other elements:

- When a standard VArray is relocated, its elements are *bit-wise* copied. This preserves the element data exactly, but invalidates any element data that consists of memory pointers to other (now relocated) elements. Consequently, elements of a standard VArray may not contain pointers to other elements.
- When a temporary VArray is relocated, its elements are copied *element-by-element*, which invokes the `element_type` default constructor to create new, empty elements and then uses `element_type` assignment to assign each original element value to a corresponding new element. This preserves the validity of any elements that point to other elements, provided that the default constructor and destructor for `element_type` manage the pointer linkage appropriately.

Note that resizing a standard VArray is faster than resizing a temporary VArray.

Extending a VArray

You can add a single element to the end of a VArray by calling its `extend` member function. You specify the value of the new element as the parameter to the member function. If the VArray is persistent, its vector is automatically opened for update, and the lock on the container is upgraded, if necessary.

Extending a VArray implicitly resizes it, which is a potentially expensive operation. You should therefore use `extend` as a convenient way to add only a single element to a VArray. If you need to add multiple elements in a single transaction, you should consider using `resize` to allocate all the elements in one operation.

EXAMPLE This example creates a persistent `Counter` object, which contains a standard VArray of integers. It fills the VArray with 21 integers and uses the `set` member function to set each integer to twice the value of its index in the VArray. The `set` member function automatically opens the vector of the persistent VArray.

```
// Application code file
#include "counter.h"
...     // Start transaction; set containerH
// Create a Counter object with an empty VArray
ooHandle(Counter) countH;
countH = new(containerH) Counter();
```

```
countH->values.resize(10);         // Set the VArray size to 10

int i;
for (i=0; i < 10; i++)
   countH->values.set(i, i*2);     // Set the VArray values

countH->values.resize(20);         // Resize the VArray to 20.
for (i=10; i < 20; i++)
   countH->values.set(i, i*2);     // Add more values

countH->values.extend(20*2);       // Extend the VArray by one

uint32 size = countH->values.size();  // Obtain the VArray size
for (i=0;i < size;i++)                // Get and print values
   printf("%d\n", countH->values[i]);
…
// Commit transaction
```

Java-Compatibility Arrays

If your application interoperates with a Java application that added class descriptions to the schema, you may have persistence-capable classes with attributes of the type `ooRef(oojArrayOf`*Type*`)`. This attribute corresponds to a Java attribute containing an array of elements of the type *Type*. For example, a C++ object of type `oojArrayOfInt32` corresponds to a Java array of elements that are 32-bit integers. Because the Java-compatibility classes for variable-size arrays are persistence capable, they cannot be embedded in a persistence-capable class the way a VArray can. Instead, they must be linked with reference attributes.

Each Java-compatibility array class is a wrapper for a VArray class; it has a member function for obtaining the VArray.

Java-Compatibility Array of Class	Member Function	Gets VArray of Class
oojArrayOfBoolean	getBooleanArray	ooVArrayt<uint8>
oojArrayOfCharacter	getCharacterArray	ooVArrayt<uint16>
oojArrayOfInt8	getInt8Array	ooVArrayt<int8>
oojArrayOfInt16	getInt16Array	ooVArrayt<int16>
oojArrayOfInt32	getInt32Array	ooVArrayt<int32>

Java-Compatibility Array of Class	Member Function	Gets VArray of Class
oojArrayOfInt64	getInt64Array	ooVArrayt<int64>
oojArrayOfFloat	getFloatArray	ooVArrayt<float32>
oojArrayOfDouble	getDoubleArray	ooVArrayt<float64>
oojArrayOfObject	getObjectArray	ooVArrayt<ooRef(ooObj)>

The Java-compatibility array classes are defined in the `javaBuiltins.h` header file.

To work with a Java-compatibility array, you call the appropriate member function to obtain the VArray, then you work with the VArray as described in this chapter.

EXAMPLE The Java persistence-capable class `Polygon` has a `sides` attribute containing an array of floating-point numbers.

```
// Java source file Polygon.java
package Geometry;
public class Polygon extends ooObj {
   public Float[] sides;
   ...
}
```

The Objectivity for Java application gives the `Polygon` class a customized schema class name of `Polygon` (in place of the default schema class name of `Geometry_Polygon`); it then adds the description of this class to the schema of a federated database. The customized schema class name allows an interoperating Objectivity / C++ application to represent persistent objects of the `Polygon` class as instances of a C++ class named `Polygon`; otherwise, the C++ class would need to be named `Geometry_Polygon`.

In the DDL file, the `sides` attribute of the `Polygon` class is declared to be an object reference to the `oojArrayOfFloat` class.

```
// DDL file geometry.ddl
class Polygon : public ooObj {
public
   ooRef(oojArrayOfFloat) sides;
   ...
};
```

The Objectivity/C++ application gets the lengths of a polygon's sides from the VArray wrapped by the Java compatibility array that is referenced by the `Polygon` object's `sides` attribute.

```
// Application code file
#include "geometry.h"
...
ooHandle(Polygon) polyH;
...     // Set polyH to reference a Polygon
// Get the VArray from the java-compatibility array
ooVArray<float32> vary = polyH->sides->getFloatArray();
uint32 nSides = vary.size();
for (uint32 i=0; i < nSides; i++) {
   cout << "Length of side " << i+1 << ": " << vary[i] << endl;
}
```

13

Objectivity/C++ Strings

Objectivity/C++ provides string classes that support the persistent storage of C++ strings. This chapter describes:

- General information about using strings as persistent data.
- Variable-size strings
- Optimized string classes
- Java-compatibility string classes

Strings as Persistent Data

A persistent object can store a fixed-size string of type `char[]` as an attribute. Because the string size is known, space for the string attribute can be allocated within the persistent object, and when the string is assigned to the attribute, its characters are embedded directly in the object.

Persistent objects can also have strings of unknown length as attributes; however, such strings cannot be of type `char *` because C++ pointers cannot be stored as persistent data. Instead, a persistence-capable class must use one of the Objectivity/C++ string classes described in this chapter.

You can use Objectivity/C++ strings anywhere in an application; furthermore, you can convert transparently between an Objectivity/C++ string and a `const char *` string, enabling Objectivity/C++ strings to be passed to functions as parameters of type `const char *` and vice versa.

NOTE If your application needs to interoperate with Java or Smalltalk applications, the attributes of your persistence-capable classes should not contain fixed-size character arrays or optimized strings; neither Java nor Smalltalk can access objects of classes that use those data types.

Variable-Size Strings

A *variable-size string* is a sequence of any number of 8-bit characters. This kind of string is appropriate for an attribute that will contain strings whose lengths are not known or are known to vary widely.

Structure of Variable-Size Strings

A variable-size string is an instance of the class `ooVString`. This class is a VArray of character elements with member functions for performing common string operations. Consequently, a variable-size string is a compound object consisting of a reference to a vector of elements. The reference portion of the variable-size string occupies a fixed amount of space, which is embedded in the containing persistent object. The vector portion is external to the persistent object; the vector occupies a variable amount of space and may be relocated by certain operations. Elements in the vector are guaranteed contiguous within virtual memory.

Working With Variable-Size Strings

You use `ooVString` constructors to create variable-size strings with 0 or more elements. An empty variable-size string has no vector allocated for it until you assign another variable-size string to it or grow it using the `resize` member function. You use this member function to grow or truncate the vector dynamically. Resizing a variable-size string to 0 elements deallocates the vector.

You get and set an individual character by its position in the variable-size string. Characters are numbered starting with 0; the position number is the character's *index* or *subscript*. Operations on a variable-size string verify that any specified indexes are valid based on the string's current size. The `length` of a variable size string is the number of characters in the VArray, not including the null terminating character that is automatically added.

Operators are provided for assignment, concatenation, access to characters, and various comparisons between two variable-length strings or between a variable-length string and a string of type `const char *`.

Because a variable-size string contains a reference to a vector of characters, you cannot access the first character by dereferencing the string; that is, the expression *`*myVString`* does not access the first element of *`myVString`*. Instead, you can specify the index 0 to the subscript operator (`operator[ ]`) to get the first character; alternatively, you can call the `head` member function to get a pointer to the first character.

EXAMPLE The DDL file `person.ddl` defines a persistence-capable class `Person` that contains data members `name` and `address` of class `ooVString`. The application code manipulates the `name` and `address` strings in various ways.

```
// DDL file person.ddl
class Person : public ooObj {
public:
   Person() { }
   Person(char * _name, char * _address, uint32 _id) :
      name(_name), address(_address), id(_id) { }
   ooVString name;
   ooVString address;
   uint32 id;
};

void search(const char *);
```

```
// Application code file
#include <stdio.h>
#include "person.h"

// Create a Person object, initializing the name data member
ooHandle(Person) personH = new Person("John", 0, 231876549);

// Assign a string to the address data member
personH->address = "124 Park Ave., Palo Alto, CA 95444";

// Concatenate a string to the name data member
personH->name += " Smith";

// Access the second character in the name
char c = personH->name[2];

// Get the length of the name
uint32 length = personH->name.length();

// Pass name as a C++ string. A VString behaves like char *
search(personH->name);

// Compare the name
if (personH->name == "Larry Johnson")
   printf("Larry Johnson is found\n");

// Create another person
ooHandle(Person) manH;
manH = new(personH) Person("Ken Smith", 0, 227549990);
```

```
// Ken Smith has the same address as John Smith
manH->address = personH->address;

// Print Ken Smith's address
// Sometimes an explicit cast of ooVString to char * is needed
printf("Ken Smith: %s\n", (const char *) manH->address);

// Compare the name
if (personH->name != manH->name)
   printf("This person is not John Smith\n");

// Resize a string
char * abc = "abcdefghijklmnopqrstuvwxyz";
uint32 length = strlen(abc);
personH->name.resize(length);
memcpy(personH->name.head(), abc, length + 1);

// But the same thing could be accomplished via
// personH->name = abc; check if the name is null or not
if (! personH->name)
   printf("Error: name is null\n");
```

Optimized Strings

An *optimized string* is a sequence of any number of ASCII characters, optimized to contain less than a specified number of characters. This kind of string is appropriate for an attribute that will contain strings that are known to be generally less than a certain length.

Structure of Optimized Strings

An optimized string is an instance of the parameterized class `ooString(`*N*`)`; the parameter *N* is a positive integer. An optimized string can have any number of ASCII characters, although it provides very efficient storage and improved performance when the number of characters is less than *N*. An optimized string contains a VArray of characters and a fixed-size character array whose length is the integer *N*, where $N > 0$. If an optimized string contains fewer than *N* characters, these characters are stored in the fixed-size array, and the vector portion of the VArray is not allocated. On the other hand, if the number of characters is greater than or equal to *N*, the vector is allocated and all of the characters are stored in it.

An optimized string always contains space for the fixed-size array (whether or not it is used) and for the VArray's reference to its vector (whether or not the vector is actually allocated). When the number of characters is *N* or greater, space for the vector is added.

The fixed portion of the optimized string is embedded in the containing persistent object; the vector, if any, is external to the object and may be relocated by certain operations. Elements in the vector are guaranteed contiguous within virtual memory.

Efficient Use of Optimized Strings

An optimized string allows you to avoid the overhead of VArrays when operating on strings whose size you can predict, and still have the flexibility to use VArrays if an occasional large string occurs. For example, if you are defining a class that contains mostly strings of fewer than 8 characters, you might want to use the `ooString(8)` class. This class provides maximum efficiency for most of your strings (avoiding VArray overhead when the VArray is not needed) and uses a VArray for the occasional occurrence of strings of length greater than 7. Furthermore, performance is better for the shorter strings whose characters are directly embedded in the containing persistent object; when a VArray is used, a dereference operation is performed to find the vector containing the characters.

For a particular attribute, you should choose a value for *N* (the length of the fixed-size character array in the class) so that a high percentage (for example, 90%) of the attribute values will be strings whose length is less than *N*. It is preferable that *N* be an even number. Note that *N* must take into account the terminating null needed by C++ strings.

An optimized string allocates the fixed-size character array whether or not it is used. If the number *N* is not properly chosen, then the fixed part of the optimized string could be too big to be fully utilized or be too small to store the string in most cases. In either case, significant storage space may be wasted. You should perform an analysis of usage patterns before selecting *N*.

Working With Optimized Strings

You work with an optimized string as you would a variable-size string (see "Working With Variable-Size Strings" on page 284). Objectivity / DB automatically manages the underlying storage mechanisms.

EXAMPLE The DDL file `person.ddl` defines a persistence-capable class `Person` whose `name` and `address` data members are optimized strings of different sizes. The application code manipulates the `name` and `address` strings in various ways.

```
// DDL file person.ddl
class Person : public ooObj {
public:
   Person() { }
   Person(const char * _name, const char * _address,
      uint32 _id) :
         name(_name), address(_address), idNumber(_id) { }
   ooString(8) name;
   ooString(24) address;
   uint32 idNumber;
};
```

```
// Application code file
#include <stdio.h>
#include "person.h"
…
// Create a Person object, initializing the name data member
ooHandle(Person) personH = new Person("John", 0, 231876549);

// Assign a string to the address data member
personH->address = "124 Park Ave., Palo Alto, CA 95444";

// Concatenate a string to the name data member
personH->name += " Smith";

// Get the length of the name
uint32 length = personH->name.length();

// Compare the name to a constant string
if (personH->name == "Larry Johnson")
   printf("Larry Johnson is found\n");

// Compare the name to a variable-sized string
ooVString who("Larry Johnson");
if (personH->name != who)
   printf("This person is not Larry Johnson\n");

// Resize a string
char * abc = "abcdefghijklmnopqrstuvwxyz";
uint32 length = strlen(abc);
personH->name.resize(length);
memcpy(personH->name.head(), abc, length + 1);
```

```
// But the same thing would be accomplished via
// person->name = abc; check whether the name is null
if (!personH->name)
   printf("Error: name is null\n");
```

Java-Compatibility Strings

If your application interoperates with a Java application that added class descriptions to the schema, you may have persistence-capable classes with attributes corresponding to a Java string or a Java string array.

- A Java string attribute (of the Java class `java.lang.String`) is stored in an Objectivity / DB federated database as an embedded object of the class `ooUtf8String`. See "Unicode Strings" below.
- A Java string array (of the Java type `java.lang.String[]`) is described in the schema as an object reference to a Java-compatibility array of class `oojArrayOfObject`. See "Java-Compatibility Arrays" on page 280. The object references in that Java-compatibility array are references to the persistence-capable class `oojString`. See "String Elements" on page 291.

The classes `ooUtf8String` and `oojString` are defined in the `javaBuiltins.h` header file.

Unicode Strings

The `ooUtf8String` class represents a *Unicode string*—a sequence of Unicode characters in UTF-8 encoding. Like `ooVString`, this class represents a Unicode string as a VArray whose elements are the component bytes of the string. The class simply enables a C++ application to store and retrieve the binary representation of a Java string; it is the application's responsibility to parse the sequence of bytes into Unicode characters.

NOTE If your application renders a Unicode string, it is responsible for selecting the appropriate glyph for any non-ASCII character in the string.

Because this class is derived from `ooVArrayT<ooChar>`, you can work with a Unicode string just as you would work with a character VArray. If you prefer, you can take advantage of the string operations provided by the class `ooVString`. To do so, you cast the `ooUtf8String` object to a `const char *` and then pass the `const char *` to the `ooVString` constructor.

EXAMPLE In this example, the persistence-capable class `Person` has a `name` attribute of type `ooUtf8String`.

```
// DDL file person.ddl
class Person : public ooObj {
public:
   …
   ooUtf8String name ;
   int32 age ;
};
```

The application performs string manipulation on a person's `name` using `ooVString` member functions; it then sets the `name` attribute to the modified string.

```
// Application code file
#include "person.h"
…
ooHandle(Person) personH;
ooUtf8String utfs_name;
ooVString vs_name
…     // Set personH to reference a Person object

// Access the name as a Unicode string
utfs_name = personH->name;

// Create an ooVString from the name
vs_name = ooVString(static_cast<const char *>(utfs_name));

// For test purposes, only first names were used;
// append last name
if (vs_name == "Robert")
   vs_name += " Smith";
…
// Cast the ooVString to const char * and assign the
// result to the Unicode string
personH->name = static_cast<const char *>(vs_name)
```

String Elements

The persistence-capable class `oojString` represents an element of a Java string array. You obtain an instance of this class from a Java-compatibility array of the class `oojArrayOfObject`. Extract the VArray of object references from the Java-compatibility array; each element of the VArray is an object reference to a string element of class `oojString`. See “Java-Compatibility Arrays” on page 280.

The `oojString` class is a wrapper for a Unicode string of the `ooUtf8String` class; its `getStringValue` member function returns the Unicode string.

To work with a string element, you call its `getStringValue` member function to obtain the corresponding Unicode string. You then work with the Unicode string as described in “Unicode Strings” on page 289.

EXAMPLE The Java persistence-capable class `Felon` has an `aliases` attribute containing an array string—namely, the aliases by which the felon is known.

```
// Java source file Felon.java
package Crime;
public class Felon extends ooObj {
   public String[] sides;
   …
}
```

The Objectivity for Java application adds the description of this class to the schema of a federated database without giving the class a customized schema class name. By default, the schema class name of `Crime_Felon` is derived from the package-qualified class name of `Crime.Felon`. An interoperating Objectivity / C++ application can represent persistent objects of the `Felon` class as instances of a C++ class named `Crime_Felon`.

In the DDL file, the `aliases` attribute of the `Felon` class is declared to be an object reference to the `oojArrayOfObject` class.

```
// DDL file crime.ddl
class Crime_Felon: public ooObj {
public
   ooRef(oojArrayOfObject) aliases;
   …
};
```

The Objectivity / C++ application gets the felon’s aliases from the VArray wrapped by the Java compatibility array that is referenced by the `Crime_Felon` object’s `aliases` attribute. It then gets the `Utf8String` wrapped by each string element of the array.

```
// Application code file
#include "crime.h"
...
ooHandle(Crime_Felon) felonH;
ooVArray<ooRef(ooObj)> vary;
ooRef(oojString) stringEltR;
Utf8String uString;
...     // Set felonH to reference a Crime_Felon object
// Get the VArray from the java-compatibility array
vary = felonH->aliases->getObjectArray();
uint32 nAliases = vary.size();
cout << "Aliases:" << endl;
for (uint32 i=0; i < nSides; i++) {
   // Get object reference to string element
   stringEltR = static_cast<ooRef(oojString)>(vary[i]);
   // Get Utf8String from the string element
   uString = stringEltR->getStringValue();
   // Print the alias
   cout << static_cast<const char *>(utfs_name) << endl;
}
```

14

Iterators

An *iterator* is an object that provides a mechanism for iterating through a group of items, called the iterator's *iteration set*.

This chapter describes the four kinds of Objectivity / C++ iterator:

- Object iterators, which step through the group of objects found from a storage object, a name scope, or a to-many association.
- Name-map iterators, which step through the key-value pairs in a name map.
- Scalable-collection iterators, which step through the objects in a scalable persistent collection.
- VArray iterators, which step through the elements of a VArray.

Object Iterators

Objectivity / C++ provides *object iterators* for finding groups of Objectivity / DB objects in a federated database. For example, you use an object iterator to find all objects in a particular storage object or all the destination objects linked to a given persistent object through a to-many association.

Understanding Object Iterators

During a transaction, you can create and initialize an object iterator to find a specified group of Objectivity / DB objects. The group of objects to be found is the object iterator's iteration set. Object iterators can be initialized to find objects in any of the following kinds of iteration sets:

- Objects contained in a particular storage object (for example, all the containers in a database).
- Objects of a given class at any level of the storage hierarchy below a particular storage object.
- Destination objects linked to a particular source object by a particular association.

- Persistent objects in a particular name scope.
- Scope objects that name a particular persistent object.

In some cases, the search for persistent objects can be restricted based on the values of particular data members.

Object-Iterator Classes

Object iterators are instances of the parameterized classes `ooItr(`*`className`*`)`, where *`className`* is a class of Objectivity / DB objects. An object iterator of the class `ooItr(`*`className`*`)` can be initialized to find objects of class *`className`* or its derived classes.

Objectivity / C++ provides parameterized object-iterator classes corresponding to the classes of basic objects, containers, databases, and autonomous partitions in the programming interface. For example, `ooItr(ooObj)` is a general-purpose object iterator for finding Objectivity / DB objects of any kind; `ooItr(ooDBObj)` is an object iterator for finding databases.

The DDL processor generates an object-iterator class for every persistence-capable class defined by an application. For example, if a DDL file contains the definition of a basic-object class `Library`, the DDL processor generates the definition of the corresponding object-iterator class `ooItr(Library)`.

Object Iterators as Handles

Every object-iterator class is a subclass of a handle class—for example, the object-iterator class `ooItr(Library)` is a subclass of the handle class `ooHandle(Library)`. An object iterator, therefore, is a special kind of handle and can invoke any of the member functions defined on the parent handle class.

When created, an object iterator is null; like any null handle, it does not reference any object.

Iteration Set

An object iterator's *iteration set* is the group of objects that the iterator can find. When created, an object iterator has no iteration set. When you *initialize* an object iterator, you specify the objects to be found; doing so creates a description of the iteration set. Once initialized, the object iterator maintains its *position* in the iteration set. The position starts out just before the first object in the iteration set.

If the iteration set is nonnull, you can *advance* the object iterator through the iteration set. Each time you advance the object iterator, you move its position forward by one object, setting it to reference the object at the current position. The first advance moves the position to the first object in the iteration set and sets

the object iterator to reference that object. Successive advances step through the iteration set, so that the object iterator references each object in turn. When the end of the iteration set is reached, the object iterator is positioned after the last object. At this point the object iterator is null once again—that is, it does not reference any object.

An object iterator makes a single pass through the iteration set, finding the objects in an undefined order. Because an object iterator works from a *description* of an iteration set (instead of producing an intermediate collection in memory), persistent objects can be added, moved, or deleted while the object iterator is active, and such changes may affect the set of objects found by the object iterator. Depending on the iteration set to be found, however, an object iterator is not guaranteed to notice such changes—for example, if an object iterator is scanning a database for `Library` objects and a `Library` object is added to the database during iteration, the new object will not be found if the object iterator has already searched the container in which the new object is clustered.

You can guarantee a stable iteration set by explicitly locking all the relevant containers before you initialize and advance the object iterator. Locking in advance also guarantees read or update access to all objects found during iteration. If guaranteed access is not required, you can increase concurrency by opening objects as you need them instead of locking the relevant containers beforehand.

Open Mode

Initializing an object iterator sets its *open mode*—that is, the intended level of access to each found object. The open mode indicates how to open each found object. As the object iterator is advanced to reference each object in the iteration set, the found object is opened as specified by the object iterator's open mode:

- The default open mode (`oocNoOpen`) indicates that the object iterator should reference the found object without opening it.
- The `oocRead` open mode indicates that the object iterator should open each found object for read.
- The `oocUpdate` open mode indicates that the object iterator should open each found object for update.

NOTE Advancing the object iterator will fail in the middle of an iteration if existing locks on a found object prohibit the object iterator from opening it as specified by the open mode.

Memory Management

Like any handle, an object iterator pins an object that it opens; see “Reference Counting With Handles” on page 212. When the object iterator is advanced from one object to the next, it is set to reference the second object and its pin on the first object is removed.

When you scan the federated database or a database for basic objects of some class, each container within the federated database or the scanned database is opened when its contents are searched. If you specify the open mode `oocUpdate`, each of these containers is opened for update and remains open until the end of the transaction—even if it does not contain basic objects of the desired class. In contrast, if you specify the open mode `oocNoOpen` or `oocRead`, each container is opened for read and then closed during the iteration (unless it remains pinned by some other open handle, as described below).

To avoid keeping containers pinned unnecessarily, you should use the open mode `oocUpdate` in such scan operations *only if* you are sure that at least one object in each container will actually be found and modified. If you need to open each found object for update, you can do so explicitly; after advancing the iterator, simply call its inherited `update` member function to open the referenced object for update.

During iteration, you may set handles other than the object iterator to reference the current object. If you do so, you should close those handles explicitly before advancing the iterator to the next object. Whereas the object iterator is closed automatically when you have advanced through the entire iteration set, any other handles you set during the iteration are left open until the end of the transaction unless you explicitly close them. If you fail to close the handles, their referenced objects may be pinned in memory until the end of the transaction. In addition, if you are scanning basic objects in the federated database or in a database, the referenced objects’ containers will remain open until the end of the transaction.

The conditions for closing (and therefore unpinning) containers during iteration are the same as those for releasing locks during iteration; see “Strategies for Avoiding Lock Conflicts” on page 119.

Working With an Object Iterator

You work with an object iterator by:

- Obtaining the definition of an appropriate object-iterator class.
- Creating an object iterator of the chosen class.
- Initializing the object iterator to find the desired objects.
- Advancing the object iterator to reference each object in the iteration set in turn.

Obtaining an Object-Iterator Class Definition

If you are creating an object iterator of a predefined Objectivity/C++ class, such as `ooItr(ooObj)` or `ooItr(ooTreeList)`, your source file must include the appropriate Objectivity/C++ header file(s) to obtain the required class definitions. See Appendix A, "Objectivity/C++ Include Files".

If you are creating object iterators to find objects of an application-defined persistence-capable class *`appClass`*, your source file must include a generated header file to obtain the definition of `ooHandle(`*`appClass`*`)`. You normally include the primary header file that is generated from the DDL file containing *`appClass`*. If, however, you are simply creating and using object iterators of class `ooItr(`*`appClass`*`)`, without actually accessing any instances of *`appClass`* itself, your source file can include just the references header file. For more information on including Objectivity/C++ header files and generated header files, see "Developing Application Source Code" on page 57.

Creating an Object Iterator

An application prepares to use an object iterator by creating a null object iterator of the appropriate object-iterator class. The following definition creates a null object iterator called `libI` with the potential to find `Library` objects as well as objects of any classes derived from `Library`:

```
ooItr(Library) libI;
```

Initializing an Object Iterator

You *initialize* a null object iterator to provide it with a description of its iteration set. For example, the null object iterator `libI`, an instance of the `ooItr(Library)` class, must be initialized so it can find a particular group of `Library` objects. After you initialize an object iterator, it is prepared to find the first object in the iteration set. However, until you call the object iterator's `next` member function, the object iterator does not yet reference any object. See "Advancing an Object Iterator" on page 298.

An initialization operation returns `oocSuccess` whenever it successfully provides the object iterator with an iteration-set description—even if the described set contains no objects. You should therefore use the return code simply to determine whether initialization completed, and not to control a loop that advances the object iterator through the iteration set.

Depending on the desired iteration set, you initialize an object iterator either by calling a member function on the object iterator or by passing the object iterator as a parameter to a member function of another object. The following table contains a summary of the various ways to initialize an object iterator.

Initialize Object Iterator to	See
Find databases in the federated database	“Finding a Database” on page 165
Find persistent objects in a particular storage object	“Finding Contained Objects” on page 357
Scan a storage object for objects of a particular class	“Scanning a Storage Object” on page 360
Find destination objects linked to a particular source object by a particular to-many association	“Following To-Many Association Links” on page 325
Find scope objects that name a particular persistent object	“Finding Scope Objects” on page 372
Find persistent objects that are named in the scope of a particular persistent object	“Finding Named Objects” on page 370
Find all autonomous partitions in the federated database	“Finding an Autonomous Partition” on page 545
Find all databases in a particular autonomous partition	“Finding Databases in a Partition” on page 550
Find all containers controlled by a particular autonomous partition	“Finding Containers Controlled by a Partition” on page 552
Find all autonomous partitions that contain an image of a particular database	“Finding Partitions That Contain an Image” on page 564

Advancing an Object Iterator

After initializing an object iterator, you advance it through the iteration set by calling its `next` member function. The first time you call `next`, the object iterator is set to reference the first found object in the iteration set. Each successive invocation of `next` sets the object iterator to reference the next found object in the set. Depending on the open mode with which the object iterator was initialized, the found object may also be opened for read or update.

You normally advance an object iterator through an iteration set by calling `next` from a `while` or `for` statement, using the return status of `next` to control the loop. The loop continues when an object is found, because `next` returns `oocTrue`, which evaluates to a nonzero value. The loop exits when the end of the set is reached, because `next` returns `oocFalse`, which evaluates to 0.

EXAMPLE This example scans a database for `Library` objects. The `next` member function both advances the object iterator and controls the `while` loop.

```
// Application code file
#include "library.h"   // Include object-iterator class
...
ooHandle(ooDBObj) dbH;
...        // Set dbH to reference database to be scanned
ooItr(Library) libI;         // Create a null Library iterator
if (libI.scan(dbH)) {        // Initialize iterator
   while(libI.next()) {         // Advance to a library
      ...                       // Process current library
   }
}
```

In this alternative, the `scan` operation that initializes the object iterator also initializes the `for` loop; the `next` member function serves as the test for continuing or exiting the loop.

```
// Application code file
#include "library.h"   // Include object-iterator class
...
ooHandle(ooDBObj) dbH;
...        // Set dbH to reference database to be scanned
ooItr(Library) libI;         // Create a null Library iterator
for (libI.scan(dbH); libI.next();) {  // Initialize iterator
      ...                       // Process current library
}
```

Accessing the Current Object

Because an object iterator is a special kind of handle, you can use it as you would use any handle. While the object iterator references a found object, you can operate on that object as follows:

- Use the direct member-access operator (`.`) to call object-iterator member functions or inherited handle member functions:

  `iterator.handleMemberFunction``(...)`

- Use the indirect member-access operator (`->`) to access the found object's data members:

  `iterator->foundObjectDataMember`

- Use the indirect member-access operator (->) to call the found object's member functions:

  iterator->foundObjectMemberFunction(...)

EXAMPLE This example iterates through all patrons linked to a library by the `members` to-many association. It tests each patron's `hasMoved` attribute (of type `ooBoolean`). If the patron has moved, it calls the object iterator's inherited `update` member function, then the patron's `setNewAddress` member function.

```
// Application code file
#include "library.h"
...
ooHandle (Library) libraryH;
...        // Set libraryH to reference the desired library
ooItr(Patron) patronI;          // Create a null Patron iterator
if (library->members(patronI)) {   // Initialize iterator
   while(patronI.next()) {     // Advance to a patron
      if (patronI->asMoved) {         // Test Patron attribute
         patronI.update();            // Call handle function
         patronI->setNewAddress(...);  // Call Patron function
      }
   }
}
```

Deleting Found Objects

You can delete the found object that is currently referenced by an object iterator without affecting the remainder of the iteration.

EXAMPLE This example scans a container for rectangles, deleting those rectangles whose area is less than 10.

```
// Application code file
#include "geometry.h"
...
ooHandle(ooContObj) contH;
ooItr(Rectangle) rectI;   // Create a null Rectangle iterator
...        // Set contH to reference the container to be scanned.

rectI.scan(contH));
```

```
while (rectI.next()) {
   if ((rectI->area) < 10){
      ooDelete(rectI);   // Delete the object
   }
)
```

Casting an Object Iterator to a Handle

Many Objectivity / C++ operations to look up groups of objects initialize a general-purpose object iterator of class `ooItr(ooObj)`. Most of these operations initialize the object iterator to find *persistent objects* of any class; a few operations initialize the object iterator to find Objectivity / DB objects of *any kind* (including the federated database, databases, and autonomous partitions). In either case, as the object iterator is advanced through the iteration set, it may reference objects of different classes.

Because the class `ooItr(ooObj)` is derived from the general-purpose handle class `ooHandle(ooObj)`, you can treat a general-purpose object iterator as if it were a general-purpose handle. In addition to accessing the referenced object as you would with a handle, you can cast the general-purpose object iterator to a type-specific handle—just as you would cast a general-purpose handle. See “Class Compatibility and Casting” on page 228.

Figure 14-1 shows the inheritance hierarchies for an application that manages fleets of rental vehicles. Suppose an object iterator of class `ooItr(ooObj)` references an object of class `Fleet`. You could cast the object iterator to a handle of class `ooHandle(Fleet)` and use that handle to call member functions defined by the class `Fleet`.

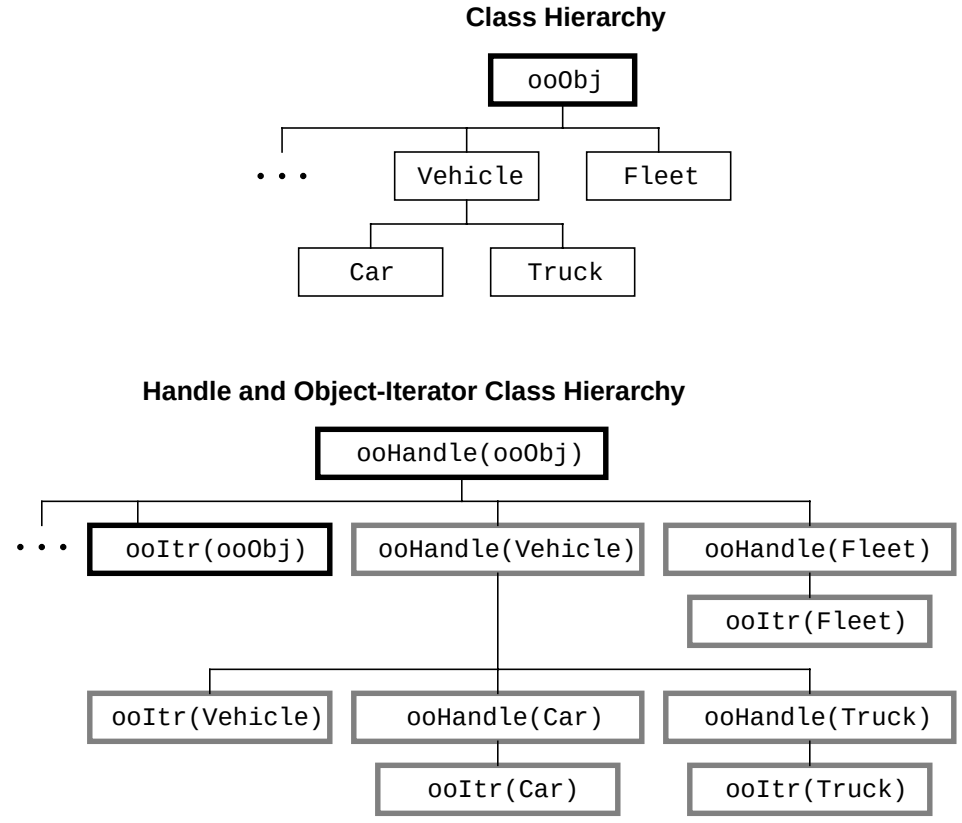


Figure 14-1 Hierarchy of Handle and Object-Iterator Classes

The same principle applies when you initialize an object iterator for persistent objects of any base class. For example, you might initialize an object iterator of class `ooItr(Vehicle)`. As you advance the object iterator through the iteration set, it may reference an object of a derived class, for example `Car`. If you need to access members defined by the derived class, you can cast the object iterator to a handle of class `ooHandle(Car)`.

As always, when you set handles from the object iterator, you must be sure to close them explicitly to prevent pinning the referenced object until the end of the transaction.

EXAMPLE This example makes a single pass through the basic objects in a container, calling the `printSummary` member function of each `Fleet` object, calling the `printStatus` member function of each `Vehicle` object, and ignoring objects of other classes.

```
// Application code file
#include "vehicle.h"
...
ooHandle(ooContObj) contH;
ooHandle(Fleet) fleetH;
ooHandle(Vehicle) vehicleH;
ooTypeNumber typeNum;
...        // Set contH to reference container to be searched
ooItr(ooObj) objI;              // Create null object iterator
contH.contains(objI);           // Initialize object iterator
while (objI.next()) {
   // Set typeNum to the type number of the current object
   typeNum = objI.typeN();
   if (typeNum == ooTypeN(Fleet)) {
      // Cast general-purpose object iterator to Fleet handle
      fleetH = static_cast<ooHandle(Fleet)>(objI);
      fleetH->printSummary;     // Call Fleet member function
      fleetH.close();           // Close Fleet handle
   }
   else if (objI->ooIsKindOf(ooTypeN(Vehicle))) {
      // Cast general-purpose object iterator to Vehicle handle
      vehicleH = static_cast<ooHandle(Vehicle)>(objI);
      vehicleH->printStatus;    // Call Vehicle member function
      vehicleH.close();         // Close Vehicle handle
   }
}
```

Terminating the Iteration

An object iterator's iteration can be terminated automatically or explicitly:

- Iteration is terminated automatically after the object iterator has found all objects in an iteration set.
- Like any handle, an object iterator is valid only during the transaction in which it was initialized. Committing, aborting, or checkpointing the transaction terminates the iteration automatically, even if the iteration set has not yet been exhausted.
- If you finish using a particular object iterator without advancing through the entire iteration set, you can terminate the iteration explicitly by calling the

object iterator's end member function. Doing so signals that you will not use the current iteration set again so the object iterator's data structures can be deleted.

Terminating the iteration makes the object iterator a null iterator, which has no iteration set. After iteration has terminated, you should not attempt to use the object iterator without reinitializing it. If you do so, an error occurs.

Object Iterators as Parameters

A common practice is to define functions that accept object iterators for various operations. For example, an application could define a function that accepts a null object iterator, initializes it, and returns it. Similarly, an application-defined function could accept an initialized object iterator, advance it, and perform a series of operations on each found object.

When defining a function that accepts an object iterator, remember that object iterators, like all handles, should be passed by reference—for example:

```
ooStatus Inspect(ooItr(ooObj) &objI);       // By reference
```

Passing object iterators by reference saves any unnecessary copying and housekeeping that would be triggered if they were passed by value.

Name-Map Iterators

A *name-map iterator* is an instance of the class `ooMapItr`; it can be initialized to step through the key-value pairs in a name map. The elements of a name map are implemented as instances of the class `ooMapElem`; a name-map iterator is actually a special kind of object iterator for finding name-map elements.

Your application must include the `ooMap.h` header file to use name maps, name-map iterators, and name-map elements.

Initializing a Name-Map Iterator

You initialize a name-map iterator to find the elements of a particular name map. The elements of the name map constitute the name-map iterator's iteration set. You can initialize a name-map iterator in either of two ways:

- Construct an initialized name-map iterator by passing a handle to the name map as the parameter to the constructor.
- Initialize a name-map iterator with the assignment operator (`operator=`), specifying a handle to the name map as the right-hand operand.

EXAMPLE This example constructs a name-map iterator initialized to find the elements of a name map.

```
// Application code file
#include "myClasses.h"   // DDL file myClasses.ddl
                         // includes <ooMap.h>
...
ooHandle(ooMap) mapH;
...     // Set mapH to reference the name map
ooMapItr mapI(mapH);  // Construct initialized mapI
```

An alternative approach is to construct a null name-map iterator and then initialize it by assignment.

```
ooMapItr mapI;     // Construct null mapI
mapI = mapH;       // Initialize mapI
```

Working With a Name-Map Iterator

Because `ooMapItr` is derived from `ooItr(ooMapElem)`, you use a name-map iterator just as you would use any object iterator:

- Advance the name-map iterator through its iteration set by calling its `next` member function.
- Use the name-map iterator as a handle to access the current element of the name map.
- If desired, call the name-map iterator's `end` member function to terminate the iteration.

Like any object iterator, a name-map iterator makes a single pass through its iteration set, finding the name-map elements in an undefined order. You cannot add or delete name-map elements while iterating over a name map. However, you can modify the objects that are referenced by the found elements.

For additional information about name-map elements, see "Finding the Values of a Name Map" on page 369.

Scalable-Collection Iterators

A *scalable-collection iterator* steps through the elements, keys, or values of a particular scalable persistent collection. These objects constitute the iterator's iteration set.

A scalable-collection iterator has a *current index*, which gives its zero-based position within the iteration set; the *current object* is the element of the iteration set at the scalable-collection iterator's current index. When the iterator is created, it is positioned before the first element of the iteration set. That is, its index is -1 and it has no current object.

A scalable-collection iterator is an instance of a class derived from class `ooCollectionIterator`. Because `ooCollectionIterator` defines the interface shared by all scalable-collection iterators, other scalable-collection iterators are not documented.

Your application must include the `ooCollections.h` header file to use scalable-collection iterators.

Initializing a Scalable-Collection Iterator

During a transaction, you obtain a pointer to an initialized scalable-collection iterator by calling a member function of a collection.

To Obtain a Pointer to an Iterator Initialized to Find	Call	See
All elements of a list or set	The `iterator` member function of a list or set	"Finding the Elements of a List or Set" on page 365
All keys of an object map	The `keyIterator` member function of an object map	"Finding the Keys and Values of an Object Map" on page 366
All values of an object map	The `valueIterator` member function of an object map	"Finding the Keys and Values of an Object Map" on page 366

You are responsible for deleting the scalable-collection iterator when you are finished using it.

Working With a Scalable-Collection Iterator

After obtaining an initialized scalable-collection iterator, you can use it in a loop that processes each element of the iteration set in turn.

To advance through the iteration set from beginning to end, call the `hasNext` member function for loop control to test whether additional elements remain in the iteration set. Within the loop, you make successive calls to the iterator's `next`

member function to get each element. As you advance through the iteration set from beginning to end, the current index increases until the scalable-collection iterator is positioned after the last element of the iteration set. At that point, the current index is the size of the iteration set and the scalable-collection iterator has no current object.

You can reposition the iterator within the iteration set. To position the iterator at a particular index, you call `goToIndex`; to position the iterator at a particular object, you call `goTo`. (The latter is useful when iterating through a sorted collection.)

After iterating forward (or repositioning the iterator), you can reverse the direction of iteration. To iterate backward from the current index, you call the `hasPrevious` member function for loop control to test whether additional elements remain in the iteration set. Within the loop, you make successive calls to the iterator's `previous` member function to get each element. As you step backward through the iteration set, the current index decreases until the scalable-collection iterator is positioned before the first element of the iteration set. At that point, the current index is back to -1 and the iterator has no current object.

Various member functions, such as `next` and `previous`, find persistent objects in the iterator's corresponding collection. These functions return general-purpose object references.

EXAMPLE This example iterates through a sorted set.

```
// Application code file
#include "myClasses.h"   // DDL file myClasses.ddl
                         // includes <ooCollection.h>
...
ooHandle(ooTreeSet) setH;
ooRef(ooObj) objR;
...    // Set setH to reference the sorted set

// Create and initialize a scalable-collecion iterator
ooCollectionIterator *setIptr = setH->iterator();

// Step through the elements of the set
while (setIptr->hasNext()) {
   objR = setIptr->next();
   ...           // Do something with this element of the set
}
delete setIptr;  // Delete the scalable-collection iterator
```

Modifying the Collection

Methods of the scalable-collection iterator allow you to modify the corresponding persistent collection. If the iterator is currently positioned at an element of the iteration set (that is, not before the first element or after the last element), the `remove` member function removes the current object from the corresponding collection; the `set` member function replaces the current object with a specified object.

EXAMPLE This example iterates over a list of `Vehicle` objects, printing the license number of each and then deleting it from the list.

```
// Application code file
#include "vehicle.h"     // DDL file vehicle.ddl
                         // includes <ooCollection.h>
...
ooHandle(ooTreeList) tlistH;
ooRef(Vehicle) vR;
...     // Set tlistH to reference the list

// Create and initialize a scalable-collecion iterator
ooCollectionIterator *listIptr = tlistH->iterator();

// Step through the elements of the set
while (listIptr->hasNext()) {
   vR = static_cast<ooRef(Vehicle)>(listIptr->next());
   cout << "Vehicle: " << vR->getLicense() << endl;
   // Remove the current element from the list
   listIptr->remove();
}
delete listIptr;  // Delete the scalable-collection iterator
```

While you are iterating through a scalable collection, the *only* way you should modify the collection is using the scalable collection iterator's `remove` and `set` member functions. In particular:

- You should not call the collection's `remove` member function to remove an element.
- You should not continue to use a scalable-collection iterator after you add elements to the collection from which you obtained the iterator. Instead, you should delete the old iterator and get a new scalable-collection iterator after you add elements to the collection.

VArray Iterators

A *VArray iterator* steps through the elements of a VArray. The elements of the VArray constitute the iterator's iteration set. A VArray iterator is an instance of a class created from the class template `d_Iterator<element_type>`, as described in the ODMG standard. The *`element_type`* parameter specifies the type of elements in the VArray. For example, a VArray iterator of the class `d_Iterator<int32>` iterates over the elements of a VArray of the class `ooVArrayT<int32>`, which are 32-bit signed integers.

Initializing a VArray Iterator

You initialize a VArray iterator to step through the elements of a particular VArray—either a standard VArray or a temporary VArray. To do so, you call the a VArray's `create_iterator` member function. When you call this function on an *`element_type`* VArray, you obtain a VArray iterator of class `d_Iterator<element_type>`.

Advancing a VArray Iterator

A newly created VArray iterator is initialized to point to the first element of that VArray. A VArray iterator makes a single pass through its iteration set, getting elements of the VArray by increasing index order.

An initialized VArray iterator supports two alternative iteration styles for advancing through the VArray's elements:

- The `next` member function tests for a next element, sets a parameter to the current element of the VArray, and then advances the iterator, all as a single operation.

  You typically use `next` to control a `while` loop that executes the same statements once for each element in the VArray.

- The `not_done`, `get_element`, and `advance` member functions perform the iteration actions as separate operations.

  You typically use `not_done` and `advance` as the expressions that control a `for` loop; within the loop, you can use `get_element` to get the current element of the VArray.

EXAMPLE This example uses a `while` loop to compute the sum of the integers that are elements of a VArray.

```
// Application code file
#include "myClasses.h"
...
ooVArrayT<uint32> values;
...     // Set values VArray;
// Initialize the VArray iterator
d_Iterator<uint32> vItr = values.create_iterator();
uint32 sum = 0;
uint32 curVal;
while (vItr.next(curVal)) {
   sum += curVal;
}
```

This alternative example uses a `for` loop to compute the sum of the elements of a VArray.

```
ooVArrayT<uint32> values;
...     // Set values VArray;
// Initialize the VArray iterator
d_Iterator<uint32> vItr = values.create_iterator();
uint32 sum = 0;
for (; vItr.not_done(); vItr.advance()) {
   sum += vItr.get_element();
}
```

Part 4 FINDING PERSISTENT OBJECTS

This part explains how to organize persistent objects to minimize search and how to use the organization to find the persistent objects when they are needed.

15

Creating and Following Links

Many applications link persistent objects together to form *object graphs,* which are directed graph data structures that consist of objects linked to other objects.

This chapter describes:

- General information about links between persistent objects
- Mechanisms for linking persistent objects into object graphs: reference attributes and associations on persistent objects, and persistent collections

Understanding Links Between Persistent Objects

An object graph consists of a number of persistent objects linked together. Each link in the graph can be thought of as an arrow from a source object to a destination object.

An application can link persistent objects together in the following ways:

- Set a reference attribute of a source object to contain object references to one or more destination objects.
- Create association links from a source object to one or more destination objects.
- Create a persistent collection (the source object of links) and add other persistent objects to the collection (making them the destination objects of the links).

Linking objects into a graph not only models the relationships among the various objects, it also facilitates finding the group of related objects. Regardless of which linking mechanisms are used to create an object graph, you can follow the links to find objects. You typically start by finding a particular object that is relevant to some task or operation. If that object is a source object for links in an object graph, you can follow those links to find the related destination objects.

The procedure for creating a link and following a link depends on the particular linking mechanism.

Linking With Reference Attributes

You can use *reference attributes* to link objects together. A reference attribute is any attribute data member that can contain one or more object references.

- A data member can contain a single object reference if its type is an object-reference class.
- A data member can contain multiple object references if its type is a fixed-size array or a VArray of object references.

The persistence-capable class defining the reference attribute is the source class for the links; the class referenced by the object references is the destination class. An object of the source class, called the *source object*, is linked by the reference attribute to the *destination objects* that the attribute references.

You create and follow links by accessing the reference attribute of a source object.

- If you have a handle to the source object, you access the reference attribute using the indirect member-access operator (`->`) on the handle.
- From within a member function of the source class, you can access the reference attribute directly, as you would in any C++ member function.

See the Objectivity/C++ Data Definition Language book for additional information about defining persistence-capable classes with attributes of object-reference types. See Chapter 12, “Variable-Size Arrays,” for more information about VArrays. See Chapter 10, “Handles and Object References,” for information about working with handles and object references to persistent objects.

Defining a Reference Attribute

You declare a reference attribute as a data member of the source class. You then run the DDL processor to generate the C++ definition of the source class, including the definition of the reference attribute.

An attribute that uses the object-reference class `ooRef(`*`className`*`)` can contain references to instances of *`className`* or its derived classes. If all destination objects are guaranteed to be stored in the same container as their referencing source object, you can use short object references of the class `ooShortRef(`*`className`*`)`.

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

```
// DDL file vehicle.ddl
class Vehicle : public ooObj {
public:
   ...
   // Attribute 'fleet' links the vehicle to its fleet
   ooRef(Fleet) fleet;
};

class Fleet : public ooObj {
public:
   ...
   // Attribute 'vehicles' links the fleet to its
   // 1000 vehicles
   ooRef(Vehicle) vehicles[1000];
};
```

An alternative approach would be to define the `vehicles` data member of the `Fleet` class to contain a VArray instead of a fixed-size array:

```
// DDL file vehicle.ddl
class Fleet : public ooObj {
public:
   ...
   // Attribute 'vehicles' links the fleet to
   // any number of vehicles
   ooVArrayT<ooRef(Vehicle)> vehicles;
};
```

Creating, Replacing, and Deleting Links

You link a source object to a destination object by setting its reference attribute to contain an object reference to the destination object. If you need to replace an existing link with a link to a different object, you modify the reference attribute, replacing the existing object reference with an object reference to the desired object. If you need to delete a link, you set the reference attribute to contain a null object reference.

As is the case for all attributes, you must set the reference attribute within an update transaction and the source object must be open for update. To ensure that

the source object is opened correctly, you may choose to define an accessor member function to set the reference attribute.

EXAMPLE The `Vehicle` class defines an accessor member function `setFleet` to set the `fleet` attribute, creating a link to the specified `Fleet` object. See the `fleet` attribute definition on page 315.

```
// Application code file
#include "vehicle.h"
...
ooStatus Vehicle::setFleet(ooHandle(Fleet) newFleetH) {
   // Open this vehicle for update
   if (ooUpdate()) {
      // Set the 'fleet' attribute to reference the
      // specified fleet
      fleet = newFleetH;
      return oocSuccess;
   }
   else
      return oocFailure;
} // End setFleet
```

Finding a Destination Object

You find a destination object by getting an object reference to it from the source object's reference attribute. If you need to perform multiple operations on the destination object, you can assign the object reference to a handle.

EXAMPLE The `Fleet` class defines the member function `printSummary` to find and print the status of each vehicle in the fleet. That function finds destination objects by accessing the fleet's `vehicles` data member.

```
// DDL file vehicle.ddl
class Vehicle : public ooObj {
public:
   ...
   void printID();       // Print identifying information
   void printStatus();   // Print status of this vehicle
};
```

```
class Fleet : public ooObj {
public:
   …
   ooRef(Vehicle) vehicles[1000];
   void printSummary();
};
```

```
// Application code file
#include "vehicle.h"
…
void Fleet::printSummary() {
   ooHandle(Vehicle) currentVehicleH;
   for (i=0; i<1000; i++) {
      // Set currentVehicleH to reference the current
      // vehicle in the fleet
      currentVehicleH = vehicles[i];
      // Print the current vehicle's identifying information
      currentVehicleH->printID();
      // Print the current vehicle's status
      currentVehicleH->printStatus();
   } // End for
} // End print Summary
```

Linking With Associations

You can use associations to link objects together. The persistence-capable class defining the association data member is the source class for the links; the class referenced by the association is the destination class.

- Use a to-one association to link each source object to a single destination object.
- Use a to-many association to link each source object to one or more destination objects.

If you need to be able to use either of two linked objects as the starting point for finding the other, you can define a pair of bidirectional associations between their classes.

Defining and Accessing Associations

You declare an association as a data member of the source class, using special Objectivity / DDL syntax. You then run the DDL processor to generate the C++ definition of the source class, including the definition of the association. The DDL processor also generates member functions for the source class that allow you to access the association. You call these member functions on a source object to test, create, delete, and follow links.

See the Objectivity / C++ Data Definition Language book for information about defining associations. See “Associations” on page 145 for more information about the various characteristics of associations.

EXAMPLE This example substitutes a pair of bidirectional associations for the `fleet` and `vehicles` attributes in the preceding example. The `Vehicle` class defines a to-one `fleet` association to link a vehicle to its rental fleet. The `Fleet` class defines the to-many `vehicles` association to link a rental fleet to all the vehicles in the fleet. Each of the two associations is declared to be the other’s inverse; that is, if a given vehicle has a `fleet` association to a particular fleet, that fleet has a `vehicles` association to the vehicle.

```
// DDL file vehicle.ddl
class Vehicle : public ooObj {
public:
   …
   // To-one bidirectional association: fleet
   //    destination class:    Fleet
   //    inverse association:  vehicles
   ooRef(Fleet) fleet <-> vehicles[];
};
class Fleet : public ooObj {
public:
   …
   // To-many bidirectional association: vehicles
   //    destination class:    Vehicle
   //    inverse association:  fleet
   ooRef(Vehicle) vehicles[] <-> fleet;
};
```

Generated Member Functions

When an association called *linkName* is defined in a persistence-capable class, the DDL processor generates a set of member functions on that class for managing a source object's associations. The DDL processor generates a slightly different set of member functions for to-one associations and for to-many associations. The following table describes the generated member functions.

Generated Member Function	Meaning for To-One Association	Meaning for To-Many Association
`exist_`*linkName*	Tests whether this source object is associated with the specified destination object	Tests whether this source object is associated with the specified destination object
`set_`*linkName*	Creates an association from this source object to the specified destination object	(Not applicable)
`add_`*linkName*	(Not applicable)	Creates an association from this source object to the specified destination object
linkName	Finds the destination object associated with this source object	Initializes an iterator to find all destination objects associated with this source object
`sub_`*linkName*	(Not applicable)	Deletes the association from this source object to the specified destination object
`del_`*linkName*	Deletes the association from this source object to its destination object	Deletes the associations from this source object to each of its destination objects

The DDL processor places the declarations for these member functions in the generated primary header file that contains the class definition; the function definitions are placed in the corresponding generated C++ implementation file, which you compile and link with your application. See "Files Generated by the DDL Processor" on page 20 of the Objectivity/C++ Data Definition Language book for a discussion of these files.

EXAMPLE The persistence-capable class `Car` defines a one-to-one unidirectional association `madeBy` whose destination class is `Manufacturer`.

```
// DDL file car.ddl
class Car: public ooObj {
public:
   ...
   ooRef(Manufacturer) madeBy : copy(delete);
};
```

The DDL processor defines the following member functions on the `Car` class:

- `exist_madeBy`—Tests whether this car is linked to a manufacturer.
- `set_madeBy`—Creates an association from this car to a particular manufacturer.
- `madeBy`—Finds this car's associated manufacturer.
- `del_madeBy`—Deletes the association from this car to its manufacturer.

The persistence-capable class `Company` defines a one-to-many bidirectional association `employs` to the `Employee` class with the inverse relationship `employedBy`.

```
// DDL file company.ddl
class Company: public ooObj {
public:
   ...
   // One-to-many bidirectional association: employs
   //    destination class:    Employee
   //    inverse association:  employedBy
   ooRef(Employee) employs[] <-> employedBy;
};
class Employee: public ooObj {
public:
   ...
   // Many-to-one bidirectional association: employedBy
   //    destination class:    Company
   //    inverse association:  employs
   ooRef(Company) employedBy <-> employs[];
};
```

The DDL processor defines the following member functions on the `Company` class:

- `exist_employs`—Tests whether this company is linked to an employee.
- `add_employs`—Creates an association from this company to a particular employee.

- employs—Initializes an iterator to find the employees of this company.
- sub_employs—Deletes the association from this company to a particular employee.
- del_employs—Deletes the associations from this company to each of its employees.

All generated member functions for accessing an association must be called within a transaction. Those that create or delete associations must be called in an update transaction.

Testing for the Existence of a Link

You can call a source object's exist_*linkName* member function to test whether the object is linked by a *linkName* association to a destination object. The parameter indicates the destination object of interest. You can pass a handle to a persistent object as the parameter to find out whether the source object is linked by a *linkName* association to that destination object. Alternatively, you can pass 0 as the parameter to find out whether the source object is linked by a *linkName* association to *any* destination object.

Linking Objects by To-One Associations

To create a link for a to-one association named *linkName*, you call the source object's set_*linkName* member function. The parameter is a handle to the destination object.

Because a given source object can be linked by a to-one association to at most one destination object, set_*linkName* signals an error if the source object already has a *linkName* association. In that case, you must delete the source object's existing *linkName* association by calling its del_*linkName* member function. Deleting an association from a source object to a destination object "breaks the link" between the two objects.

EXAMPLE The Vehicle class defines a member function assignToFleet to link a vehicle to the fleet to which it belongs. It can be called for a new vehicle or for a vehicle that is being moved from one fleet to another. See the fleet association definition on page 318.

```
// Application code file
#include "vehicle.h"
...
ooStatus Vehicle::assignToFleet(ooHandle(Fleet) fleetH) {
ooHandle(Vehicle) vH;
```

```
// Set vH to reference this vehicle
ooThis(vH);

// No action is needed if this vehicle is already linked
// to the specified fleet
if (!vH->exists_fleet(fleetH)) {
   if (vH->exists_fleet(0)) {
      // The vehicle is already linked to some other fleet;
      // delete that link
      vH->del_fleet();
   }
   // Link the vehicle to the specified fleet
   vH->set_fleet(fleetH);
}
```

Linking Objects by To-Many Associations

To create a link for a to-many association named *linkName*, you call the source object's add_*linkName* member function. The parameter is a handle to the destination object.

Because a source object can be linked by a to-many association to any number of destination objects, you can call add_*linkName* repeatedly to link the source object to any number of destination objects. Each call creates a new *linkName* association from the source object to the specified destination object. In fact, this member function allows you to create duplicate associations between the same source and destination objects (even though it could be semantically meaningless to do so).

EXAMPLE The Patron class has a many-to-many association canUse to the Library class, and the Library has an inverse many-to-many association members to the Patron class.

```
// DDL file library.ddl
class Patron: public ooObj {
public:
   ...
   // Many-to-many bidirectional association: canUse
   //    destination class:    Library
   //    inverse association:  members
   ooRef(Library) canUse[] <-> members[];
};
```

```
class Library: public ooObj {
public:
   ...
   // Many-to-many bidirectional association: members
   //    destination class:    Patron
   //    inverse association:  canUse
   ooRef(Patron) members[] <-> canUse[];
};
```

The application links a patron to the Main library, avoiding a duplicate link if the patron can already use the Main library.

```
// Application code file
#include "library.h"
...
trans.start();
... // Open the federated database for update

// Set mainLibH to reference the Main library
ooHandle(Library) mainLibH = ... ;

// Set patH to reference a patron specified by the user
ooHandle(Patron) patH = ... ;

// Test whether the patron can use the Main library;
// if not, link the patron to the library
if (!patH->exists_canUse(mainLibH)) {
   patH->add_canUse(mainLibH);
}
trans.commit();
```

A source object can be linked by a to-many association to any number of destination objects; you can break one or all of these links:

- Call a source object's `del_`*`linkName`* member function to delete all *`linkName`* associations to destination objects.
- Call a source object's `sub_`*`linkName`* member function to delete its *`linkName`* association to a particular destination object. You specify the destination object by passing its handle as a parameter to `sub_`*`linkName`*. If the source object has *`linkName`* associations to other destination objects, those associations remain unchanged.

  If *`linkName`* is a many-to many association or a one-to-many bidirectional association, multiple links may exist from the source object to a given destination object. For those associations, the generated `sub_`*`linkName`* member function takes a second parameter that you can use to delete

multiple links. You can specify 0 to delete all links between the two objects; you can specify a different number to delete that number of links between the objects. If you omit the second parameter, a single link between the two objects is deleted.

EXAMPLE This example deletes all links from a patron to a library. See the `canUse` association definition on page 322.

```
// Application code file
#include "library.h"
...
trans.start();
... // Open the federated database for update

// Set libH to reference a library specified by the user
ooHandle(Library) libH = ... ;

// Set patH to reference a patron specified by the user
ooHandle(Patron) patH = ... ;

// Remove any links from the patron to the library
patH->sub_canUse(libH, 0);
trans.commit();
```

Following To-One Association Links

To find the destination object linked to a source object by a to-one association named *linkName*, you call the source object's *linkName* member function.

If you want to set a particular object reference or handle to reference the destination object, you can pass it as a parameter to *linkName*. The parameter should be a *className* object reference or handle, where *className* is the destination class of the *linkName* association. If you do not pass such a parameter, the *linkName* member function allocates a new handle in which to return the destination object. In all cases, the function returns the object reference or handle that it sets. If the source object does not have a *linkName* association; the function returns a null object reference or handle.

EXAMPLE This example follows the `employedBy` link from an employee to find the company where an employee works. See the `employedBy` association definition on page 320.

```
// Application code file
#include "company.h"
...
trans.start();
... // Open the federated database
// Set empH to reference the desired employee
ooHandle(Employee) empH = ... ;

// Set compH to reference the company where the employee works
ooHandle(Company) compH;
if (!empH->employedBy(compH)) {
   // Something went wrong; the employee has no company
   trans.abort()
}
trans.commit();
```

If you want to open the destination object for read or update, you can specify the desired open mode as a parameter to the *`linkName`* member function.

Following To-Many Association Links

You can initialize an object iterator to find the destination objects that are linked to a particular source object by a to-many association. To do so, you call the source object's *`linkName`* member function, passing as a parameter the object iterator to be initialized. This function returns a success code indicating whether initialization succeeded. If the source object has no *`linkName`* associations, the object iterator is initialized to a null iterator.

After initializing the object iterator, you advance it through the iteration set by calling the iterator's `next` member function. See "Object Iterators" on page 293 for information about working with an object iterator.

By default, *`linkName`* initializes the object iterator to reference, but not open, each destination object when it advances through the iteration set. If you want the iterator to open each destination object for read or update, you can specify the desired open mode as a parameter to the *`linkName`* member function.

Finding All Destination Objects

By default, the *linkName* member function initializes an object iterator to find all destination objects.

EXAMPLE This example initializes an iterator to find all patrons who are members of a library. See the `members` association definition on page 323. After initializing the iterator, the application steps through the patrons, printing the name of each. The `Patron` class defines the `printName` member function (not shown).

```
// Application code file
#include "library.h"
...
trans.start();
... // Open the federated database
ooItr(Patron) patronI; // Create a null Patron iterator

// Set libraryH to reference the desired library
ooHandle(Library) libraryH = ... ;

// Initialize the iterator to find all members of the library
if (library->members(patronI)) { // Initialization succeeded
   while(patronI.next()) {
      patronI->printName(); // Print the current patron's name
   }
   trans.commit();
}
else
   trans.abort();
```

Finding Destination Objects that Satisfy a Condition

You can initialize an object iterator to find only those destination objects that satisfy a condition. To do this, you pass a predicate string as a parameter to the *linkName* member function. For additional information about predicate strings, see "Predicate Queries" on page 375. Note that, unlike a predicate scan, a predicate query over destination objects is not optimized through the use of indexes.

EXAMPLE This example demonstrates how to find those members of a library whose last names begin with the letter M. See the members association definition on page 323.

```
// Application code file
#include "library.h"
...
ooStatus rc;
ooTrans trans;
...
trans.start();
... // Open the federated database
ooItr(Patron) patronI; // Create a null Patron iterator

// Set libraryH to reference the desired library.
ooHandle(Library) libraryH = ... ;

// Initialize the iterator to find those members of the library
// whose last names begin with the letter M
rc = library->members(
                patronI,             // Iterator to initialize
                "lastName =~ M.*");  // Predicate to test
if (rc) {
   while(patronI.next()) {
      ... // Process the current patron
   }
   trans.commit();
}
else
   trans.abort();
```

When finding destination objects that satisfy a condition, you may specify an open mode for the found objects and the access level of the data members to be tested by the predicate.

EXAMPLE The call to members in the previous example could be modified as follows to open each found destination object for read; because it tests only a public data member, it specifies oocPublic as the access level.

```
rc = library->members(
          patronI,             // Iterator to initialize
          oocRead,             // Open mode for found objects
          oocPublic            // Access level for data members
          "lastName =~ M.*");  // Predicate to test
```

Associations and Attributes

In some respects, data members for associations—especially one-to-one unidirectional associations—are similar to attribute data members containing object references. For example, both kinds of data member provide storage for an object reference to the destination object. However, associations save you work because:

- The member functions generated for each association provide a complete interface for accessing destination objects. In contrast, you must define your own accessor member functions for getting and setting object references in attribute data members.

  Note that the destination objects referenced by an association data member are accessible *only* through the generated interface. A member function of the source class cannot access association data members directly to get and set the destination objects (as it can for attribute data members).

  In addition, the generated member functions provide access to *all* associations. In contrast, you can choose to make attribute data members private and not implement accessor member functions for them.
- The member functions generated for bidirectional associations automatically operate on both associations in the pair—that is, adding or removing an association in one direction simultaneously adds or removes the inverse association. In contrast, you must implement referential integrity yourself when you link objects with attribute data members.
- The member functions generated for to-many associations automatically allow you to add, remove, and iterate over destination objects. In contrast, if you use a fixed-size array or VArray of object references in attribute data members to manage multiple associations, you must implement your own mechanism for adding, removing, and iterating over the object references to the destination objects in the array.

Linking With Persistent Collections

A persistent collection maintains links to the objects it contains. A persistent object itself represents a simple object graph. It can also be embedded within a larger object graph.

Because a persistent collection is an aggregate of persistent objects, a source object can use a link to a persistent collection as an alternative to a to-many association or an attribute containing an array of object references. In this case, the persistent collection acts as an intermediate object linking one source object to a group of destination objects. Although this approach introduces an additional level of indirection, there are circumstances where it may be worth while. For

example, if you need to link a source object to a very large number of destination objects, you may want to use an intermediate scalable collection.

To use a persistent collection as an intermediate object between a source object and its destination objects:

- Create a persistent collection containing the desired destination objects. See "Building a Persistent Collection" on page 242.
- Link the source object to the persistent collection, using either an attribute or a to-one association.

EXAMPLE In this example, the `vehicles` attribute of the `Fleet` class references a set of the `ooTreeSet` class. The set maintains object references to the various vehicles in the fleet.

```
// DDL file vehicle.ddl
#include <ooCollection.h>
...
class Vehicle : public ooObj {
public:
   ...
   // Attribute 'fleet' links the vehicle to its fleet
   ooRef(Fleet) fleet;
};
class Fleet : public ooObj {
public:
   ...
   // Attribute 'vehicles' links the fleet to
   // a set of its vehicles
   ooRef(ooTreeSet) vehicles;
};
```

To find a destination object from the source object, you must follow two links:

- You find the persistent collection by following the attribute or association link from the source object, as described in this chapter.
- You follow links from the collection to find the destination objects it contains. Chapter 16, "Individual Lookup of Persistent Objects," explains how to look up individual objects in a persistent collection; Chapter 17, "Group Lookup of Persistent Objects," explains how to iterate over the objects in a persistent collection.

16

Individual Lookup of Persistent Objects

Objectivity/C++ provides a number of mechanisms for assigning a unique key to each persistent object in a group. The key can be a name, a numeric index, or another persistent object. A given persistent object can belong to different groups and can have a different key in each group.

To support individual lookup, you must organize objects into a group, assigning a unique key to each. This chapter describes:

- General information about individual lookup
- Organization and individual lookup within name scopes, name maps, lists, sets, and object maps
- Using unique indexes for individual lookup

You can use these mechanisms to organize and find containers and basic objects, including basic objects of Objectivity/C++ classes, such as persistent collections.

An additional mechanism is available for naming and looking up a particular container within the group of containers in a given database. Unlike other basic objects, a container can have a system name because it is also a storage object. See “Creating a Container” on page 172 for information on setting a container’s system name; see “Finding a Container” on page 176 for information on looking up a container by system name. Note, however, that looking up a system name is relatively slow compared to other lookup techniques, such as looking up a name in a name scope or in a name map.

Understanding Individual Lookup

Individual lookup is appropriate within a group of objects that are relevant to a given task if the application typically performs that task on one particular object selected from the group. For example, a human-resources application might perform a sequence of operations when an employee is transferred from one department to another. Although the department-transfer operations could reasonably be performed on any `Employee` object in the federated database, the

application never actually iterates through all employees performing these operations for each. Instead, it executes the department-transfer task for a particular employee in response to a user request. It is therefore useful to be able to find an individual employee without having to search through all employees in the federated database.

Regardless of the kind of individual lookup mechanism to be used, you organize the objects as follows:

- Select a "focus object" that will identify the group of objects to be searched. The focus object can be either a scope object that defines a name scope to be searched, or a persistent collection to be searched.
- If the focus object is itself a persistent object, provide a way to find it with minimum search (for example, give it a scope name).
- Give each persistent object in the group a unique key that is meaningful to you or to the users of your application.

When you need to find a particular object, you:

- Find the focus object for the group of relevant objects.
- Get the key of the desired object.
- Look up the object's key within the group defined by the focus object.

Individual Lookup in Name Scopes

Objectivity / DB allows you to name persistent objects within the scope of a particular *scope object*. A scope object defines a *name scope*—a group of persistent objects for which the scope object maintains unique names. Each of these names is called a *scope name* and is the key for finding a particular persistent object within the name scope. You can think of a name scope as a local name space defined by the scope object.

When you want to find a particular named object, you first find the scope object that defines the name scope. The scope object is the focus object that identifies the group of objects to be searched. You then find the desired object within the name scope by looking up its name.

A persistent object can have no more than one scope name within a particular name scope, and all scope names within a name scope must be unique. However, a given persistent object can have different names in a number of different name scopes.

Scope Objects

A scope object can be any kind of Objectivity / DB object: the federated database, a database, a persistent container, a persistent basic object, or an autonomous partition. All name scopes support individual lookup of named objects. In addition, the name scopes of persistent objects (containers and basic objects) also support group lookup—that is, they allow you to find all named objects in the name scope. See “Group Lookup in Name Scopes” on page 370.

A scope object uses the hashing mechanism of a hashed container to pair each scope name with the appropriate object. Therefore, a container used as a scope object must be a hashed container, and a basic object used as a scope object must be stored in a hashed container. See “Hashed and Nonhashed Containers” on page 170. Table 16-1 identifies the hashed container used by each kind of scope object.

Table 16-1: Hashed Container Used for Scope Objects

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 system database of the federated database
Autonomous partition	The default container of that partition’s system database

When setting or removing scope names, the application must be able to obtain an update lock on the hashed container used by the scope object. When looking up a scope name, the application must be able to obtain a read lock on this container.

Building a Name Scope

To build a name scope, you first select its scope object. You then add the desired objects to the name scope by naming it in the scope of the scope object. If you need to remove an object from the name scope, you unname it.

NOTE You can perform group lookup in a name scope only if its scope object is a persistent object. See “Group Lookup in Name Scopes” on page 370. Therefore, if you plan to perform both individual and group lookup in a name scope, be sure to use a container or basic object as the scope object, and not the federated database, a database, or an autonomous partition.

You name and unname a persistent object by calling member functions on a handle to the object:

- Call the handle's nameObj member function to add the persistent object to a name scope. The parameters are a handle to the scope object and the desired scope name. The scope name is a null-terminated string of up to 487 nonnull characters.
- Call the handle's unnameObj member function to remove the persistent object from the name scope. The parameter is a handle to the scope object.
- If you need to change the scope name of a persistent object, you must first call unnameObj to delete the existing name, then call nameObj to set the new name.

EXAMPLE This example uses the database named "Sales" as the scope object for naming Employee objects who are the company's sales representatives. Each sales representative manages the accounts for a single client company. Within the name scope, a sales representative has a scope name composed of the account number of the client company.

```
// DDL file company.ddl
class Employee : public ooObj {
   …
};
class Client : public ooObj {
public:
   uint32 accountNo;
   …
};
```

The newSalesRep function is called during a transaction to assign an employee to be the sales representative for a particular client.

```
// Application code file
#include "company.h"
…
ooStatus newSalesRep(
         ooHandle(Employee) &repH,       // New sales rep
         ooHandle(Client) &clientH) {    // Client company
   char repName[16];

   // Get a handle to the federated database
   ooHandle(ooContObj) contH = repH.contantedIn();
   ooHandle(ooDBObj) dbH = contH.containedIn();
   ooHandle(ooFDObj) fdH = dbH.containedIn();
```

```
   // Make this an update transaction
   if (!fdh.update())
      return oocError;

   // Find the scope object
   if (!dbH.open(fdH, "Sales", oocUpdate))
      return oocError;

   // Set repName to the rep's scope name
   sprintf(repName, "%d", clientH->accountNo);

   // Add the rep to the database's name scope
   return repH.nameObj(dbH, repName);
}
```

Finding an Object by Scope Name

To look up a persistent object by its scope name, call the `lookupObj` member function on a handle of the appropriate class; this function sets the handle to reference a named object. The parameters are a handle to the scope object, the scope name, and an optional open mode. The found object is opened in the specified mode (or for read if no mode is specified).

If you know that the named object is an instance of *`className`* or a class derived from *`className`*, you can call the `lookupObj` member function on a handle of type `ooHandle(`*`className`*`)`. If you don't know the class of the named object, you can call the `lookupObj` member function on a handle of type `ooHandle(ooObj)` and use runtime type identification (page 189) to determine the class of the found object.

EXAMPLE The `getSalesRep` function is called during a transaction to find the sales representative for a specified client. This function converts the client's account number to a scope name, and looks up that name in the name scope of the `Sales` database. It sets the specified handle to reference the found `Employee` object. The found object is not opened.

```
// Application code file
#include "company.h"
...
ooStatus getSalesRep(
           uint32 accountNo,          // Client's account number
           ooHandle(Employee) &repH) { // Handle to set
   ooHandle(ooFDObj) fdH;
   ooHandle(ooDBObj) dbH;
   char repName[16];
```

```
    // Get a handle to the federated database
    if (!fdh.open("Corporate", oocRead))
        return oocError;

    // Find the scope object--the focus object for the search
    if (!dbH.open(fdH, "Sales"))
        return oocError;

    // Set repName to the rep's scope name
    sprintf(repName, "%d", accountNo);

    // Look up sales rep by its scope name; set repH to
    // reference the found object
    return repH.lookupObj(dbH,            // Scope object
                          repName,        // Scope name
                          oocNoOpen);     // Open mode
}
```

Individual Lookup in Name Maps

An alternative to using a name scope is to create an application-specific dictionary that assigns a unique name to each object in a group. You implement an application-specific dictionary as a *name map*—a nonscalable, unordered persistent collection of key-value pairs in which each key is a string and each value is a persistent object. Each key string can be thought of as an application-defined name for the corresponding value object.

A name map is a persistent object of the class `ooMap`. Your application must include the `ooMap.h` header file to use name maps. For general information about name maps and other persistent collections, see Chapter 11, “Persistent Collections”.

You can create as many application-specific 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-specific dictionary instead of a name scope is that you have control over the clustering and growth performance of the name map that implements the dictionary.

When you want to find a particular named object, you first find the name map that implements the dictionary. The name map is the focus object that identifies the group of objects to be searched. You then find the desired object in the name map by looking up its name.

Naming an Object

You name a persistent object in a name map by adding an element to the map. The new element's key is the name; its value is the persistent object. For example, if you know that a given name is not already used in a particular name map, you can call the map's add member function to add an element with that name. See "Building a Name Map" on page 245.

EXAMPLE This example modifies the one on page 334 to use a name map instead of a name scope to group the sales representatives for a company. The name map for the sales representatives is named "SalesReps" in the name scope of the database named "Sales".

```
// Application code file
#include <ooMap.h>
#include "company.h"
...
ooStatus newSalesRep(
          ooHandle(Employee) &repH,       // New sales rep
          ooHandle(Client) &clientH) {    // Client company
   ooHandle(ooMap) nameMapH;
   char repName[16];

   // Get a handle to the federated database
   ooHandle(ooContObj) contH = repH.contantedIn();
   ooHandle(ooDBObj) dbH = contH.containedIn();
   ooHandle(ooFDObj) fdH = dbH.containedIn();

   // Make this an update transaction
   if (!fdh.update())
      return oocError;

   // Find the scope object for the name map
   if (!dbH.open(fdH, "Sales", oocUpdate))
      return oocError;

   // Find the name map
   if (!nameMapH.lookupObj(dbH, "SalesReps", oocUpdate))
      return oocError;

   // Add the representative to the name map, using
   // client's account number as the key
   sprintf(repName, "%d", clientH->accountNo);
   return nameMapH->add(repName, repH);
} // End newSalesRep
```

Finding an Object by Name

To look up a persistent object by its name in a name map, call the map's lookup member function. You pass the name as a parameter to lookup; you may optionally pass:

- A handle to be set to the found object.

  The function expects a persistent-object handle of class ooHandle(ooObj). If you know the class of the object to be found, you can use a type-specific handle and cast it to the expected parameter type.

- An open mode.

  If you omit this parameter, the found object is opened for read by default.

EXAMPLE The getSalesRep function is called during a transaction to find the sales representative for a specified client. It looks up the sales representative by its name in the SalesReps name map and sets the specified handle to reference the found Employee object. Note that the handle is cast to ooHandle(ooObj) before being passed to lookup. The found object is not opened.

```
// Application code file
#include <ooMap.h>
#include "company.h"
...
ooStatus getSalesRep(
          uint32 accountNo,          // Client's account number
          ooHandle(Employee) &repH) { // Handle to set
   ooHandle(ooFDObj) fdH;
   ooHandle(ooDBObj) dbH;
   ooHandle(ooMap) nameMapH;
   char repName[16];

   // Get a handle to the federated database
   if (!fdh.open("Corporate", oocRead))
      return oocError;

   // Find the scope object for the name map
   if (!dbH.open(fdH, "Sales"))
      return oocError;

   // Find the name map--the focus object for the search
   if (!nameMapH.lookupObj(dbH, "SalesReps"))
      return oocError;
```

```
    // Look up the sales rep in the name map,
    // using the account number as the lookup key
    sprintf(repName, "%d", accountNo);
    return nameMapH->lookup(repName, repH, oocNoOpen);
}
```

Individual Lookup in Lists

If sequential integer indexes provide sufficient information for finding the persistent objects in a group, you can create a list containing those objects. A *list* is a persistent collection whose elements can be located by position, given as a zero-based index. A list can contain duplicate elements and null elements.

A list is a persistent object of the class `ooTreeList`. Your application must include the `ooCollections.h` header file to use lists. For general information about lists and other persistent collections, see Chapter 11, “Persistent Collections”.

When you want to find a particular object by its index, you first find the list. The list is the focus object that identifies the group of objects to be searched. You then find the desired object in the list by looking up its index.

Assigning an Index

You assign an index to a persistent object in a list by adding the object to the list at the desired index. See “Building a List” on page 244. For example, you could add the objects to the list in order by calling the list’s `add` (or `addLast`) member function repeatedly. To change the object at a particular index, you can call the list’s `set` member function.

If you intend to use indexes within the list as lookup keys, you should avoid inserting objects into the list. When you insert an object at a particular index, you increment the indexes of all current elements at and after that position in the list.

EXAMPLE At year end, a company ranks its sales representatives by their total sales for the year. The representative who sold the most is ranked first, and so on. The sales representatives are grouped in a list, ordered by their ranking. After the sales representatives have been ordered by their total sales, each one in turn is added to the end of the list, which is initially empty.

```
// Application code file
#include <ooCollection.h>
#include "company.h"
...
ooTrans trans;
ooHandle(ooTreeLisp) listH;
ooHandle(Employee) repH;

trans.start();
...     // Open the federated database for update
...     // Calculate total sales for each sales rep and order
        // the sales reps by decreasing total sales

...     // Set listH to reference the list

// Delete last year's sales reps from the list
listH->clear();

// Iterate over the sales reps ordered by ranking
while (...) {
   ...     // Set repH to reference the next sales representative

   // Add this sales rep to the end of the list
   listH->add(repH);
}
trans.commit()
```

Finding an Object by Index

To look up a persistent object by its index in a list, call the list's `get` member function, passing the index of the desired object. This function returns an object reference to the object at the specified position. If you need to perform multiple operations on the object, you can assign the object reference to a handle.

EXAMPLE A company calculates year-end bonuses for a sales representative based on the company's profit and the sales rep's ranking in total sales for the year. One year, for example, the company awarded $10,000 to the top performing sales rep, $8000 to the second, $5000 to the third, $3000 to the fourth through tenth, $2000 to the eleventh through twentieth, and so on.

The `awardBonus` function finds the sales representative with the specified rank and awards that employee a bonus of the specified amount. The list that orders the sales representatives by their current ranking is named `"SalesPerformance"` in the name scope of the database named `"Sales"`.

```
// Application code file
#include <ooCollection.h>
#include "company.h"
…
ooStatus awardBonus(
            int32 rank,           // Index to look up
            int32 bonusAmt) {     // Bonus to award
   ooTrans trans;
   ooHandle(ooFDObj) fdH;
   ooHandle(ooDBObj) dbH;
   ooHandle(ooTreeList) listH;
   ooHandle(ooObj) objH;
   ooHandle(Employee) repH;
   ooStatus rc = oocSuccess;

   trans.start();
   // Make this an update transaction
   if (!fdh.open("Corporate", oocUpdate)) {
      trans.abort();
      return oocError;
   }

   // Find the list--the focus object for the search
   if (!dbH.open(fdH, "Sales", oocUpdate)) {
      trans.abort();
      return oocError;
   }
   if (!listH.lookupObj(dbH, "SalesPerformance")) {
      trans.abort();
      return oocError;
   }

   // Look up sales rep by rank in the list;
   // assign the returned object reference to a handle
   objH = listH->get(rank);
   if (objH.isNull()) {
      trans.abort();
      return oocError;
   }
   repH = static_cast<ooHandle(Employee)>(objH);
   repH.update();           // Open sales rep for update
   repH->bonus = bonusAmt;  // Set bonus attribute
   trans.commit()
   return rc;
}
```

Individual Lookup in Sets

If the objects in a group have unique values for a particular identifying attribute, those objects can be collected into a set. A *set* is a persistent collection with no duplicate elements and no null elements. A *sorted set* is an instance of `ooTreeSet`; an *unordered set* is an instance of `ooHashSet`. Your application must include the `ooCollections.h` header file to use sets. For general information about sets and other persistent collections, see Chapter 11, “Persistent Collections”.

A set’s comparator can enable you to look up individual elements by values of an identifying attribute. The comparator must be an instance of an application-defined class that supports content-based lookup. See “Application-Defined Comparator Classes” on page 256.

When you want to find a particular object by its identifying attribute, you first find the set. The set is the focus object that identifies the group of objects to be searched. You then find the desired object in the set by looking up the value of its identifying attribute.

Providing an Identifying Attribute for Elements

An application-defined comparator class can provide the ability to identify persistent objects based on their persistent data. The comparator can use any component data to identify an object—that is, the values of any attributes of the object.

A comparator can use any number of attributes to identify the objects, and the attributes can be of any data types. For simplicity, the remaining discussion assumes that the set’s comparator uses a single identifying attribute instead of a combination of attributes.

The comparator class implicitly defines the identifying attribute by using that attribute to order or hash elements of the set:

- The comparator for a sorted set orders the objects in the set based on their values for the identifying attribute. See “Supporting Content-Based Lookup in a Sorted Collection” on page 260.
- The comparator for an unordered set computes an object’s hash value based on its value for the identifying attribute. See “Supporting Content-Based Lookup in an Unordered Collection” on page 266.

If you plan to perform individual lookup on the persistent objects in a set, you must:

- Define a comparator class that uses the identifying attribute to sort or hash objects.
- Create an instance of your comparator class and assign it to the set when you create the set.

EXAMPLE A human-resources application considers an employee eligible for promotion after being at a given job level for a certain period of time. It groups employees who are eligible for promotion into an unordered set of `Employee` objects.

A comparator of the class `CompSSN` hashes `Employee` objects based on their social security numbers. It can also identify an `Employee` object based on the social security number in the object's `SSN` attribute.

```
// DDL file company.ddl
#include <ooCollection.h>
...
class Employee : public ooObj {
public:
   ...
   ooVString SSN;
   oouint16 jobLevel;
   oocSuccess eligibleForPromotion();
   oocSuccess computeNewSalary();
};

// See example on page 266 for member function definitions
class CompSSN : public ooCompare {
public:
   virtual int hash (const ooHandle(ooObj) &objH) const;
   virtual int hash (const void *&lookupVal) const;
   virtual int compare (const ooHandle(ooObj) &ob11jH,
                        const ooHandle(ooObj) &obj2H) const;
   virtual int compare (const ooHandle(ooObj) &obj1H,
                        const void *&lookupVal) const;
};
```

The function `initEligible` creates the set for eligible employees, using a comparator of the class `CompSSN`. For best concurrency, a comparator should be clustered in a different container from any collection that uses it. The function `initEligible` clusters the comparator in the container `Comparator` and clusters the set in the container `Personnel`, both in the `HR` database. It names the set `Eligible` in the name scope of the `HR` database.

```
// Application code file
#include <ooCollection.h>
#include "company.h"
…
ooStatus initEligible() {
   ooTrans trans;
   ooHandle(ooFDObj) fdH;
   ooHandle(ooDBObj) dbH;
   ooHandle(ooContObj) contH;
   ooHandle(CompSSN) comparatorH;
   ooHandle(ooHashSet) setH;

   trans.start();
   // Make this an update transaction
   if (!fdh.open("Corporate", oocUpdate)) {
      trans.abort();
      return oocError;
   }
   if (!dbH.open(fdH, "HR", oocUpdate)) {
      trans.abort();
      return oocError;
   }
   if (!contH.open(dbH, "Comparator", oocUpdate)) {
      trans.abort();
      return oocError;
   }

   // Create the comparator
   comparatorH = new(contH) CompSSN();
   if (!contH.open(dbH, "Personnel", oocUpdate)) {
      trans.abort();
      return oocError;
   }

   // Create the set, assigning the comparator to it
   setH = new(contH) ooHashSet(comparatorH);

   // Name the set in the database's name scope
   if (!setH.nameObj(dbH, "Eligible")) {
      trans.abort();
      return oocError;
   }
   trans.commit();
   return oocSuccess;
}
```

Assigning an Identifying Value

Before an object is added to the set, it must be assigned a unique value for the identifying attribute. When you add the object to the set, it will be sorted or hashed according to the value for its identifying attribute. That value serves as the key with which the object can be found.

You add persistent objects to the set as described in "Building a Set" on page 243. For example, you can call the set's `add` member function to add the element.

EXAMPLE The `eligibleForPromotion` member function of the `Employee` class is called during a transaction when a human-resources application has determined that the employee is eligible for promotion; it adds the employee to the set `Eligible`. Because that unordered set uses a comparator of the `CompSSN` class, the employee is hashed by its social security number. An employee's social security number is stored in its `SSN` attribute when the object is created.

```
// Application code file
#include <ooCollection.h>
#include "company.h"
...
ooStatus Employee::eligibleForPromotion() {
   ooHandle(ooHashSet) setH;
   ooHandle(Employee) empH;

   // Get a handle to the federated database
   ooThis(empH);  // Set empH to reference this employee
   ooHandle(ooContObj) contH = empH.contantedIn();
   ooHandle(ooDBObj) dbH = contH.containedIn();
   ooHandle(ooFDObj) fdH = dbH.containedIn();

   // Make this an update transaction
   if (!fdh.update())
      return oocError;
   if (!dbH.open(fdH, "HR", oocUpdate))
      return oocError;

   // Find the set of eligible employees
   if (!setH.lookupObj(dbH, "Eligible"))
      return oocError;

   // Add this employee to the set of eligible employees
   setH->add(empH);
   return oocSuccess;
}
```

Finding an Element by Identifying Value

If a set's comparator identifies elements by some attribute, you can call the set's `get` member function to find the element with a particular value for its identifying attribute. The parameter to `get` is a pointer of type `const void *&` to the value that identifies the desired element. This function returns an object reference to the element (or a null object reference if the set does not contain such an element). If you need to perform multiple operations on the element, you can assign the object reference to a handle.

EXAMPLE The `promotion` function is called during a transaction when a manager interacting with a human-resources application indicates the desire to promote an employee. If the employee is eligible for promotion, the `jobLevel` attribute is incremented and the `computeNewSalary` member function (not shown) is called.

The parameter to the `promotion` function is a string containing the social security number of the employee to be promoted. The function uses that string to look up the Employee object in the `Eligible` set.

```
// Application code file
#include <ooCollection.h>
#include "company.h"
...
ooStatus promotion(const char *SSN) {
   ooHandle(ooFDObj) fdH;
   ooHandle(ooDBObj) dbH;
   ooHandle(ooHashSet) setH;
   ooHandle(Employee) empH;

   // Make this an update transaction
   if (!fdh.open("Corporate", oocUpdate))
      return oocError;

   // Find the Eligible set--the focus object for the search
   if (!dbH.open(fdH, "HR", oocUpdate))
      return oocError;
   if (!setH.lookupObj(dbH, "Eligible"))
      return oocError;

   // Look up the employee by social security number;
   // assign the returned object reference to a handle
   ooHandle(ooObj) objH = setH->get(SSN);
   if (objH == 0) {   // Not found
      cout << "Employee not eligible for promotion" << endl;
      return oocSuccess;
   }
```

```
    // Cast the general-purpose handle to an Employee handle
    // then modify Employee attributes
    empH = static_cast<ooHandle(Employee)>(objH);
    empH.update();
    empH->jobLevel += 1;
    empH->computeNewSalary();
    return oocSuccess;
}
```

Individual Lookup in Object Maps

If each object in a group can be identified by pairing it with another persistent object as its unique key, the objects can be collected into an object map. An *object map* is a collection of key-value pairs in which the key is a persistent object and the value is a persistent object or null. The objects to be grouped together are values in the object map. Each is paired with a unique key object that can be used to find the corresponding value object.

A *sorted object map* is an instance of `ooTreeMap`; an *unordered object map* is an instance of `ooHashMap`. Your application must include the `ooCollections.h` header file to use object maps. For general information about object maps and other persistent collections, see Chapter 11, “Persistent Collections”.

When you want to find a particular value object, you first find the object map. The object map is the focus object that identifies the group of objects to be searched. You then find the desired value object in the set by looking up its key.

Assigning a Key

When you add elements to an object map, you specify the key of the element. Typically, you add an element with the `put` member function, which specifies the key and the value of the new element. See “Building an Object Map” on page 246.

EXAMPLE This example is a variation of the ones shown on page 334 and page 337. Instead of grouping sales representatives in a name scope or in a name map, it groups them in an object map. The key for each sales representative is its client object. The object map is named `"SalesReps"` in the scope of the `Sales` database.

The `newSalesRep` function is called during a transaction to assign an employee to be the sales representative for a particular client.

```
// Application code file
#include <ooCollection.h>
#include "company.h"
...
ooStatus newSalesRep(
          ooHandle(Employee) &repH,       // New sales rep
          ooHandle(Client) &clientH) {    // Client company
   ooHandle(ooTreeMap) objMapH;

   // Get a handle to the federated database
   ooHandle(ooContObj) contH = repH.contantedIn();
   ooHandle(ooDBObj) dbH = contH.containedIn();
   ooHandle(ooFDObj) fdH = dbH.containedIn();

   // Make this an update transaction
   if (!fdh.update())
      return oocError;

   // Find the scope object for the object map
   if (!dbH.open(fdH, "Sales", oocUpdate))
      return oocError;

   // Find the object map
   if (!objMapH.lookupObj(dbH, "SalesReps", oocUpdate))
      return oocError;

   // Add the rep to the object map with the
   // client as its key
   return objMapH->put(clientH, // key object
                       repH);    // value object
}
```

Finding an Object by Key

To look up a persistent object by its key in an object map, call the object map's `get` member function, passing a handle to the key object. This function returns an object reference to the object paired with the specified key (or a null object reference if the object map does not contain such that key). If you need to perform multiple operations on the found object, you can assign the object reference to a handle.

EXAMPLE The `getSalesRep` function finds a sales representative for a given client in the `SalesReps` object map, setting the specified handle to reference the found `Employee` object. The found object is not opened.

```
// Application code file
#include <ooCollection.h>
#include "company.h"
...
ooStatus getSalesRep(
            ooHandle(Client) &clientH,   // Client
            ooHandle(Employee) &repH) { // Handle to set
   ooHandle(ooTreeMap) objMapH;

   // Get a handle to the federated database
   ooHandle(ooContObj) contH = clientH.contantedIn();
   ooHandle(ooDBObj) dbH = contH.containedIn();
   ooHandle(ooFDObj) fdH = dbH.containedIn();

   // Find the scope object for the object map
   if (!dbH.open(fdH, "Sales"))
      return oocError;

   // Find the object map--the focus object for the search
   if (!objMapH.lookupObj(dbH, "SalesReps"))
      return oocError;

   // Look up the sales rep by key in the object map;
   // assign the returned object reference to a handle
   ooHandle(ooObj) objH = objMapH->get(clientH);
   if (objH == 0)        // Not found
      return oocError;
   repH = static_cast<ooHandle(Employee)>(objH);
   return oocSuccess;
}
```

Providing an Identifying Attribute for Keys

If the objects to be used as keys in an object map have unique values for a particular identifying attribute, the object map can be created with an application-defined comparator that identifies keys by their values for the identifying attribute.

An application-defined comparator class can provide the ability to identify persistent objects based on their persistent data. The comparator can use any component data to identify an object—that is, the values of any number of

attributes of any data types. For additional information, see "Application-Defined Comparator Classes" on page 256.

For simplicity, the remaining discussion assumes that the object map's comparator uses a single identifying attribute instead of a combination of attributes.

The comparator class implicitly defines the identifying attribute by using that attribute to order or hash the keys of the object map:

- The comparator for a sorted object must sort the keys based on their values for the identifying attribute. See "Supporting Content-Based Lookup in a Sorted Collection" on page 260.
- The comparator for an unordered object map computes a key's hash value based on its value for the identifying attribute. See "Supporting Content-Based Lookup in an Unordered Collection" on page 266.

If you use this feature, you must:

- Define a comparator class that uses the identifying attribute to sort or hash keys.
- Create an instance of your comparator class and assign it to the object map when you create the object map.

EXAMPLE In the preceding example, each client has a unique account number. The object map of sales representatives could use a comparator that identifies a key by its account number.

A comparator of the class `CompAccount` sorts `Client` objects based on their account numbers. It can also identify a `Client` object based on the account numbers in the object's `accountNo` attribute. See the example on page 260 for more information about the class `CompAccount`.

```
// DDL file company.ddl
#include <ooCollection.h>
…
class Client : public ooObj {
public:
   uint32 accountNo;
   …;
};
class CompAccount : public ooCompare {
   …
};
```

The function `initSalesReps` creates the object map for sales representatives using a comparator of the class `CompAccount`.

For best concurrency, a comparator should be clustered in a different container from any collection that uses it. The function `initSalesReps` clusters the comparator in the container `Comparator` and clusters the object map in the container `Clients`, both in the `Sales` database. It names the object map `SalesReps` in the name scope of the `Sales` database.

```
// Application code file
#include <ooCollection.h>
#include "company.h"
...
ooStatus initSalesReps() {
   ooTrans trans;
   ooHandle(ooFDObj) fdH;
   ooHandle(ooDBObj) dbH;
   ooHandle(ooContObj) contH;
   ooHandle(CompAccount) comparatorH;
   ooHandle(ooTreeMap) objMapH;

   trans.start();
   // Make this an update transaction
   if (!fdh.open("Corporate", oocUpdate)) {
      trans.abort();
      return oocError;
   }
   if (!dbH.open(fdH, "Sales", oocUpdate)) {
      trans.abort();
      return oocError;
   }

   // Create the comparator, clustered in the
   // container named Comparator
   if (!contH.open(dbH, "Comparator", oocUpdate)) {
      trans.abort();
      return oocError;
   }
   comparatorH = new(contH) CompAccount();

   if (!contH.open(dbH, "Clients", oocUpdate)) {
      trans.abort();
      return oocError;
   }

   // Create the object map, assigning the comparator to it
   objMapH = new(contH) ooTreeMap(comparatorH);
```

```
   // Name the object map in the database's name scope
   if (!objMapH.nameObj(dbH, "SalesReps")) {
      trans.abort();
      return oocError;
   }
   trans.commit();
   return oocSuccess;
}
```

Finding an Object by Key's Identifying Value

If an object map's comparator identifies keys by some attribute, you can call the object map's `get` member function, passing a pointer of type `const void *&` to the data that identifies the desired key. This function returns an object reference to the value object whose key has the specified value in its identifying attribute (or a null object reference if the object map does not contain such a key). If you need to perform multiple operations on the found object, you can assign the object reference to a handle.

EXAMPLE This variant of the `getSalesRep` function takes advantage of the object map's ability to identify a `Client` key object by the client's account number. Whereas the variant on page 349 finds a sales representative by looking up a given client *object*, this variant finds the sales representative by looking up the client's *account number*. As before, the specified handle is set to reference the found `Employee` object and that object is not opened.

Note that this variant is equivalent in functionality to the one on page 335, which uses an account number as the name in a name scope, and the one on page 338, which uses an account number as the name in a name map.

```
// Application code file
#include <ooCollection.h>
#include "company.h"
...
ooStatus getSalesRep(
          uint32 accountNo,          // Client's account number
          ooHandle(Employee) &repH) { // Handle to set
   ooHandle(ooFDObj) fdH;
   ooHandle(ooDBObj) dbH;
   ooHandle(ooTreeMap) objMapH;

   // Get a handle to the federated database
   if (!fdh.open("Corporate", oocRead))
      return oocError;
```

```
    // Find the scope object for the object map
    if (!dbH.open(fdH, "Sales"))
        return oocError;

    // Find the object map--the focus object for the search
    if (!objMapH.lookupObj(dbH, "SalesReps"))
        return oocError;

    // Look up the sales rep by client's account number;
    // assign the returned object reference to a handle
    ooHandle(ooObj) objH = objMapH->get(&accountNo);
    if (objH == 0)        // not found
        return oocError;

    // Cast the general-purpose handle to an Employee handle
    // and set the result parameter repH
    repH = static_cast<ooHandle(Employee)>(objH);
    return oocSuccess;
}
```

Unique Indexes

If each object of a class has a unique value for a particular attribute or a unique combination of values for a number of attributes, you could create a *unique index* for objects of the class, using those identifying attributes as the key fields of the index. Doing so would allow you to perform a predicate scan to find the individual object with a given combination of values for the identifying attributes. The predicate scan initializes an object iterator to find all objects that satisfy the predicate; because only one object has a particular combination of values for its identifying attributes, however, the object iterator will only find the single object with the specified combination of values. Predicate scans are described in “Scanning for Objects That Satisfy a Condition” on page 362; unique indexes are described in “Indexes” on page 390.

Using a unique index and predicate scans is an alternative to creating a sorted set of the objects of the class and defining a comparator for the set to test the values of its identifying attribute(s); see “Individual Lookup in Sets” on page 342. In choosing between these two approaches, you should consider the following characteristics:

- Both approaches create data structures in which object references to the relevant objects are sorted by the values of their identifying attributes.

- An index sorts all objects of a given class and all its derived classes that are stored in a given storage object.

  A set is more flexible. For example, it could include objects of some, but not all, derived classes of a given class; it could include a selection of objects that are located in different storage objects or a subset of the objects of a given class that are located in a particular storage object.
- Indexes can be updated automatically when new objects of an indexed class are created or modified. See “Understanding Indexes” on page 390. In contrast, you must explicitly add a newly created object to a set. If you modify the values of an object’s identifying attributes, you need to remove it from the set, then add it back so that it will be sorted in the correct position.

17

Group Lookup of Persistent Objects

Objectivity/C++ provides a number of mechanisms for grouping persistent objects so that applications can find the group of objects without searching the entire federated database. The various grouping mechanisms all support group lookup through iteration; to look up the objects in a group, the application initializes an iterator to find those objects.

This chapter describes:

- General information about group lookup
- Organization and group lookup in the storage hierarchy, persistent collections, and name scopes

Lookup by these means can be used for any persistent object: a basic object of an application-defined class, a container, or a basic object of an Objectivity/C++ class such as a persistent collection.

Understanding Group Lookup

Group lookup is appropriate for objects that are relevant to a particular task if the application typically performs that task on each object in a group without distinguishing one of these objects from another. For example, a payroll application might calculate an employee's pay based on information in the corresponding `Employee` object: exempt or nonexempt status, salary or hourly rate, number of hours worked in a given period, and so on. To prepare pay checks, the application would initialize an iterator to find the `Employee` objects, and then iterate through those objects, calculating the pay for each one.

The grouping mechanism chosen for a particular group of objects determines what kind of iterator you initialize to find the objects.

To Find Objects Grouped by	You Initialize
Storage hierarchy	Object iterator
List	Scalable-collection iterator
Set	Scalable-collection iterator
Object map	Scalable-collection iterator
Name map	Name-map iterator
Name scope	Object iterator

Chapter 14, “Iterators,” explains how to work with the various kinds of iterators.

Group Lookup in the Storage Hierarchy

The hierarchy of storage objects in the federated database can be used to group persistent objects. If you store the objects relevant to a particular task in a particular storage object, you can find the objects by searching just that storage object instead of the entire federated database.

After deciding how to use the storage hierarchy to group your persistent objects, you create the necessary storage objects. You then create your application’s basic objects, storing each in the appropriate storage object.

When you want to find the basic objects relevant to a particular task, you first find the storage object that contains them, then you search that storage object to find the basic objects.

Creating the Storage Hierarchy

If you use the storage hierarchy to organize basic objects, you need to create the databases and containers where you will group the basic objects that are relevant to particular tasks. See “Creating a Database” on page 163 and “Creating a Container” on page 172.

After you have created the storage objects, you assign basic objects to them as appropriate. When you create a basic object, you pass a clustering directive to `operator new` to indicate the container where it should be stored. See “Creating a Basic Object” on page 184.

Finding a Storage Object

If you want to search a database for containers or basic objects, you find the database as described in "Finding a Database" on page 165. For example, you could look up the database by its system name.

If you want to search a container for basic objects, you find the container as described in "Finding a Container" on page 176. Because a container is a persistent object, you can additionally use any of the following organization and lookup strategies:

- Create a link to the container that you follow to find it, as described in Chapter 15, "Creating and Following Links".
- Identify the container with a key by which you can look it up, as described in Chapter 16, "Individual Lookup of Persistent Objects".
- Include the container in a group of persistent objects that you can search, as described in this chapter.

Finding Contained Objects

Once you have a handle to a storage object, you can initialize an object iterator to find the objects that are one level below the storage object in the hierarchy. To do this, you call the `contains` member function on the handle to the storage object, specifying an object iterator of the appropriate type as the parameter:

- To find the containers in a database, you pass an object iterator of class `ooItr(ooContObj)` to the `contains` member function on a database handle.
- To find all the basic objects in a container, you pass an object iterator of class `ooItr(ooObj)` to the `contains` member function on a container handle.

By default, the `contains` member function initializes the iterator so it will simply reference each found object when it advances through the iteration set. If you want the iterator to open each found object for read or update, you specify the desired open mode as a parameter to the `contains` member function.

After initializing the object iterator, you advance it through the iteration set by calling the iterator's `next` member function. See "Object Iterators" on page 293 for information about working with an object iterator.

Finding All Contained Objects

The `contains` member function finds all objects in a storage container, independent of their classes. You can find all containers (of any container class) in a database or all basic objects (of any basic-object class) in a container.

EXAMPLE This example finds all the basic objects in the container referenced by `contH`.

```
// Application code file
#include "myClasses.h"
...
ooHandle(ooContObj) contH;
...        // Set contH to reference the desired container.
ooItr(ooObj) objectI;           // Create a null object iterator
if (contH.contains(objectI)) {  // Initialize the iterator
...        // Initialization succeeded, process each object.
}
```

You can combine calls to the `contains` member function to traverse the entire storage hierarchy of the federated database or any part of it.

EXAMPLE This example illustrates how to traverse the entire storage hierarchy of the federated database whose boot file name is the value of the `OO_FD_BOOT` environment variable.

```
// Application code file
#include "myClasses.h"
...
ooTrans trans;
ooHandle(ooFDObj) fdH;
ooStatus rc;
ooItr(ooDBObj) dbI;
ooItr(ooContObj) contI;
ooItr(ooObj) objI;

transaction.begin();
if (rc = fdH.open()) {
   // Find all databases
   if (rc = fdH.contains(dbI, oocRead)) {
      while (dbI.next()) {
         // Find all containers in current database
         if (rc = dbI.contains(contI, oocRead)) {
            while (contI.next()) {
               // Find all basic objects in current container
               if (rc = contI.contains(objI, oocRead)) {
                  while (objI.next()) {
                     ...    // Process current object
                  } // End while more basic objects in container
               } // End if objI was initialized
            } // End while more containers in database
```

```
         } // End if contI was initialized
      } // End while more databases in federated database
   } // End if dbI was initialized
} // End if federated database was opened
if (rc)
   trans.commit();
else
   trans.abort();
```

Filtering Objects by Class

When you iterate over the containers in a database or the basic objects in a container, you may want to test the class of each found object. For example, you might want to exclude objects of one or more classes from an operation or perform different operations on objects of different classes. (If you want to select only those objects of a particular class, you could scan the storage object for objects of the desired class; see “Scanning a Storage Object” on page 360.)

You can identify the class of each persistent object in the iteration set as described in “Runtime Type Identification” on page 189.

If you need to perform different operations on found objects of different classes, you may also need to reference the objects through handles of the corresponding handle classes. When you know the type of the current object, you can cast the general-purpose object iterator to a handle of the appropriate class. See “Casting an Object Iterator to a Handle” on page 301.

If you set any handle to reference the current object in the iteration set, you should explicitly close that handle before advancing the iterator. Whereas the object iterator is closed automatically when you have advanced through the entire iteration set, any other handles you set during the iteration are left open until the end of the transaction unless you explicitly close them.

EXAMPLE This example finds all objects in a container that is used to store only fruit objects. It performs one operation on apples, a different operation on oranges, and a still different operation on berries. An apple is an instance of the `Apple` class; an orange is an instance of the `Orange` class; a berry is an instance of any class derived from the class `Berry`.

Each pass through the iteration loop consists of the following steps:

- Test the type of the found object.
- Cast the general-purpose object iterator to the appropriate handle class to reference the found object.

- Use the type-specific handle to perform the necessary operations on the found object.
- Close the type-specific handle.

```
ooHandle(ooContObj) contH;
ooTypeNumber typeNum;
ooHandle(Apple) appleH;
ooHandle(Orange) orangeH;
ooHandle(Berry) berryH;
...        // Set contH to reference the container of interest
ooItr(ooObj) objI;       // Create a null object iterator
contH.contains(objI);    // Initialize the iterator
while (objI.next()) {
   // Set typeNum to the type number of the current fruit
   typeNum = objI.typeN();
   if (typeNum == ooTypeN(Apple)) {
      appleH = static_cast<ooHandle(Apple)>(objI);
      ...        // Use appleH to perform operation for Apple
      appleH.close();       // Close Apple handle
   }
   else if (typeNum == ooTypeN(Orange)) {
      orangeH = static_cast<ooHandle(Orange)>(objI);
      ...        // Use orangeH to perform operation for Orange
      orangeH.close();      // Close Orange handle
   }
   else if (objI->ooIsKindOf(ooTypeN(Berry))) {
      berryH = static_cast<ooHandle(Berry)>(objI);
      ...        // Use berryH to perform operation for Berry
      berryH.close();       // Close Berry handle
   }
}
```

Scanning a Storage Object

You can scan any storage object to find objects of a particular persistence-capable class at any lower level in the hierarchy. In particular, you can:

- Scan a container for basic objects of a specified class.
- 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` member function, which performs this operation more quickly than the `scan` member function. See “Finding Contained Objects” on page 357.

- 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.

Once you have a handle to a storage object, you can initialize an object iterator to scan that storage object. To do this, you call the `scan` member function on an object iterator for the class of object you want to find. You pass the storage-object handle as a parameter to the `scan` function. When you call the `scan` member function of an object iterator of class `ooItr(`*`className`*`)`, you search the storage object and its descendants in the storage hierarchy for all objects of the class *`className`* and its derived classes. You can use an iterator of class `ooItr(ooObj)` to find all persistent objects (containers or basic objects).

Different variants of `scan` allow you to scan for all objects of the specified class or to scan for those objects of the class that satisfy a condition.

By default, the `scan` member function initializes the object iterator so it will simply reference each found object when it advances through the iteration set. If you want the iterator to open each found object for read or update, you specify the desired open mode as a parameter to the `scan` member function.

After initializing the object iterator, you advance it through the iteration set by calling the iterator's `next` member function. See "Object Iterators" on page 293 for information about working with an object iterator.

Scanning for All Objects of a Class

By default, the `scan` member function initializes the iterator to find all objects of the relevant class at any level below the specified storage object in the storage hierarchy.

EXAMPLE This example finds all objects of class `Rectangle` and derived classes in the database referenced by the handle `dbH`. The scan operation's return code is used to test whether initialization succeeded (but does not indicate whether any objects have been found).

```
// DDL file geometry.ddl
…
class Rectangle : public ooObj{
public:
   int32 length, width, area;
   …
};
```

```
// Application code file
#include "geometry.h"
...
ooHandle(ooDBObj) dbH;
...           // Set dbH to reference the database to be scanned
ooItr(Rectagle) rectI;    // Create a null Rectangle iterator
if (rectI.scan(dbH)) {   // Initialize the Rectangle iterator
   ...        // If initialization succeeds, advance the iterator
}
```

If you call `scan` on an object iterator of class `ooItr(ooObj)`, you initialize the iterator to find *all* persistent objects in the specified storage container. Even if you limit your scan to a particular class, the scan operation must still search the entire storage object. Each page of that storage object, in turn, is brought into the cache; each object on each page is examined to determine whether it is an instance of the desired class (or a derived class).

Because every object on every page in the storage object must be examined, a scan operation on the federated database is a very expensive operation. A scan operation of a database is also a relatively expensive operation. As a consequence, you should try to organize the persistent objects in your federated database to minimize the need for applications to scan the federated database or individual databases.

Scanning for Objects That Satisfy a Condition

When you want to retrieve only those objects of a class that satisfy some condition, you perform a *predicate scan*. To do this, you supply `scan` with a predicate string that describes a condition that the found objects must satisfy. For example, you could use a predicate scan to find all `Rectangle` objects in a particular database whose `area` attribute is greater than some value. See “Predicate Queries” on page 375 for a description of predicate strings.

EXAMPLE This example scans a database for `Rectangle` objects with an area greater than 10.

```
// Application code file
#include "geometry.h"
...
ooStatus rc;
ooTrans trans;
ooHandle(ooDBObj) dbH;
trans.start();
... // Open the federated database
... // Set dbH to reference the database to be scanned
```

```
ooItr(Rectangle) rectI;  // Create a null Rectangle iterator

// Initialize the Rectangle iterator
rc = rectI.scan(
            dbH,         // Database to scan
            "area>10");  // Predicate to test
if (rc) {
   ...       // If initialization succeeds, advance the iterator
   trans.commit();
}
else
   trans.abort();
```

A predicate scan may specify an open mode for the found objects and the access level of the data members to be tested by the predicate.

EXAMPLE The call to `scan` in the previous example could be modified as follows to open each found object for read; because it tests only a public data member, it specifies `oocPublic` as the access level.

```
// Application code file
#include "geometry.h"
...
rc = rectI.scan(
            dbH,         // Database to scan
            oocRead,     // Open mode for found objects
            oocPublic    // Access level for tested members
            "area>10");  // Predicate to test
```

A predicate scan searches the entire storage object being scanned. Each page of that storage object, in turn, is brought into the cache. Each object on each page is examined to determine whether it is an instance of the desired class (or a derived class); if so, it is tested to determine whether it satisfies the specified condition.

You can make a predicate scan more efficient by defining an index whose key fields are the attributes you test in the predicate. When an index exists, the scan operation does not need to search the entire storage object. Instead, only the pages of the index itself and pages that contain the indexed objects need to be brought into the cache. Ideally, the index itself can be used to determine which objects match the predicate; in that case, not all indexed objects are brought into

the cache—only those that satisfy the predicate. For a complete discussion of indexes, see “Indexes” on page 390.

Group Lookup of Containers

Whenever you initialize an object iterator of class `ooItr(ooContObj)` for group lookup in the storage hierarchy, the iterator is initialized to find *all* containers in the indicated group—both default containers and containers created by applications. Similarly, when you initialize an object iterator of class `ooItr(ooObj)` by scanning the federated database or a database, the iterator is initialized to find all persistent objects in the indicated group, which includes basic objects, default containers, and containers created by applications.

If you are interested only in those containers created by applications but not default containers, you should use runtime type identification (RTTI) to check for default containers, which are instances of the class `ooDefaultContObj`. See “Runtime Type Identification” on page 189. When the object iterator is set to reference a default container, you can skip whatever operations apply only to containers created by applications.

EXAMPLE This example finds all containers in the federated database, and then operates on all found containers except default containers.

```
// Application code file
#include <oo.h>
...
ooHandle(ooFDObj) fdH;
...        // Start transaction and set fdH
ooItr(ooContObj) contI;  // Null container iterator
fdH.contains(contI);     // Initialize the iterator
while (contI.next()) {
   // See if the found container is a default container
   if (contI.typeN() != ooTypeN(ooDefaultContObj)) {
      ...                // Operate on nondefault container
   }
}
```

Group Lookup in Persistent Collections

Persistent collections provide another mechanism for grouping objects. You group objects into a persistent collection by adding them to the collection; the objects in a collection may be stored in different storage objects from each other

and from the persistent collection itself. See “Building a Persistent Collection” on page 242.

Although some persistent collections are intended for individual lookup of objects, all provide a mechanism for group lookup of the objects.

- *Lists* and *sets* have persistent objects as elements. Lists are instances of the class `ooTreeList`; sorted sets and unordered sets are instances of `ooTreeSet` and `ooHashSet`, respectively.

  You can initialize a scalable-collection iterator to find all elements in a set or a list.

- *Object maps* have key-value pairs as elements; each key and each value is a persistent object. A sorted object map is an instance of `ooTreeMap`; an unordered object map is an instance of `ooHashMap`.

  You can initialize a scalable-collection iterator to find all keys in an object map or to find all values in an object map.

- *Name map*s have key-value pairs as elements; each key is a string (or name) and each value is a persistent object. A name map is an instance of the class `ooMap`.

  You can initialize a name-map iterator to find all key-value pairs in a name map.

Your application must include the `ooCollections.h` header file to use lists, sets, object maps, and scalable-collection iterators; it must include the `ooMap.h` header file to use name maps and name-map iterators. For general information about persistent collections, see Chapter 11, “Persistent Collections”.

Finding the Elements of a List or Set

You can call the `iterator` member function of a list or set to obtain a pointer to a new scalable-collection iterator, initialized to find the elements of that list or set. You are responsible for deleting the scalable-collection iterator when you have finished using it.

A scalable-collection iterator finds the elements of a list or sorted set as they are ordered within the collection. A scalable-collection iterator finds the elements of an unordered set in an undefined order.

Once you have initialized the scalable-collection iterator, you can use it to find the elements of the list or set. For example, you can call its `hasNext` member function for loop control and call its `next` member function to get each element. See “Working With a Scalable-Collection Iterator” on page 306. Member functions such as `next` return a general-purpose object reference to the found object; if necessary, you can assign it to a handle or cast it to the appropriate type.

EXAMPLE A human-resources application groups employees who are eligible for promotion into an unordered set of `Employee` objects. Before each yearly performance appraisal period, the application iterates through the elements of the set, calling the `notifyManager` member function. That function notifies the employee's manager to assess the employee's qualifications for promotion.

```
// Application code file
#include <ooCollection.h>
#include "company.h"
...
ooHandle(ooHashSet) setH;
ooRef(Employee) empR;
...     // Set setH to reference set of eligible Employees

// Create and initialize a scalable-collection iterator
ooCollectionIterator *eligibleIptr = setH->iterator();

// Iterate through the Employees in the set
while (eligibleIptr->hasNext()) {
   empR = static_cast<ooRef(Employee)>(eligibleIptr->next());
   empR->notifyManager();
}
delete eligibleIptr;  // Delete the scalable-collection iterator
```

Finding the Keys and Values of an Object Map

Object maps are designed primarily for individual lookup; you can find a value object in the object map by looking up its corresponding key object. See "Individual Lookup in Object Maps" on page 347. However, it is sometimes necessary to find all the objects that have been grouped into an object map.

You can call an object map's member functions to obtain a pointer to a new scalable-collection iterator, initialized to find its keys or its values; you are responsible for deleting the scalable-collection iterator when you have finished using it.

- Call the `keyIterator` member function to initialize the iterator to find the keys of the object map.
- Call the `valueIterator` member function to initialize the iterator to find the values of the object map.

A scalable-collection iterator initialized by `keyIterator` finds the keys of a sorted object map in their sorted order; it finds the keys of an unordered object map in an undefined order.

A scalable-collection iterator initialized by `valueIterator` finds the value of a sorted object map in the order in which their keys are sorted; it finds the values of an unordered object map in an undefined order. In either case, the iterator initialized by `valueIterator` advances through the key-value pairs of the object map in the same order as does the iterator returned by `keyIterator`.

Once you have initialized the scalable-collection iterator, you can use it to find the keys or values of the object map. For example, you can call its `hasNext` member function for loop control and call its `next` member function to get each object in the iteration set. See "Working With a Scalable-Collection Iterator" on page 306. Member functions such as `next` return a general-purpose object reference to the found object; if necessary, you can assign it to a handle or cast it to the appropriate type.

EXAMPLE The sales representatives of a company are grouped together in a sorted object map. Each element in the object map has a `Client` object as its key and an `Employee` object as its value; this pairing indicates that the employee (value) is the sales representative for the client (key). This example iterates through all the company's sales representatives, calling the `computeYearlySales` member function for each.

```
// Application code file
#include <ooCollection.h>
#include "company.h"
...
ooHandle(ooTreeMap) objMapH;
ooHandle(ooObj) objH;
ooHandle(Employee) repH;
...    // Set objMapH to reference the object map of sales reps

// Create and initialize a scalable-collection iterator
ooCollectionIterator *repIptr = objMapH->valueIterator();

// Iterate through the sales representatives (value objects)
while (repIptr->hasNext()) {
   objH = repIptr->next();
   repH = static_cast<ooHandle(Employee)>(objH);
   repH.update();
   repH->computeYearlySales();
}
objH.close();   // Close general-purpose handle
repH.close();   // Close Employee handle used during iteration
delete repIptr; // Delete scalable-collection iterator
```

If you want to iterate through the key-value pairs of an object map, you can iterate over the keys, calling scalable collection iterator's `currentValue` member function to get the value corresponding to the current key.

EXAMPLE This example iterates through the keys and values of the object map of sales representatives, printing the name of the sales representative for each client.

```
// DDL file company.ddl
#include <ooCollection.h>
class Employee : public ooObj {
public:
   ooVString name;
   …
};
class Client : public ooObj {
public:
   ooVString companyName;
   …
};
```

```
// Application code file
#include "company.h"
…
ooHandle(ooTreeMap) objMapH;
ooRef(Client) clientR;
ooRef(Employee) repR;
…    // Set objMapH to reference the object map of sales reps

// Create and initialize iterator for clients (key objects)
ooCollectionIterator *clientIptr = objMapH->keyIterator();

// Iterate through clients, getting the sales rep for each
while (clientIptr->hasNext()) {
   clientR = static_cast<ooRef(Client)>(clientIptr->next());
   repR =
     static_cast<ooRef(Employee)>(clientIptr->currentValue());
   cout << clientR->companyName << ": "
        << repR->name << endl;
}
delete clientIptr;    // Delete iterator for clients
```

Finding the Values of a Name Map

Name maps are designed primarily for individual lookup; you can find a value object in the name map by looking up its corresponding key. See “Individual Lookup in Name Maps” on page 336. However, it is sometimes necessary to find all the objects that have been grouped into a name map—that is, the values in the name-map’s key-value pairs.

To find all key-value pairs of a name map, you initialize a name-map iterator (an instance of class `ooMapItr`) to find the elements of the name map. For example, you can use the assignment operator (=) to assign the name map to the name-map iterator. See “Initializing a Name-Map Iterator” on page 304. You then call the name-map iterator’s `next` member function to obtain each successive name-map element. See “Working With a Name-Map Iterator” on page 305.

The elements of a name map are implemented as *name-map elements*. Each name-map element is an instance of the persistence-capable class `ooMapElem`, which represents a key-value pair. Thus, on each step through the iteration set, a name-map iterator references the current name-map element. You can call member functions of a name-map element to get its key and its value.

- Call the name-map element’s `name` member function to get its key, a C++ string of type `const char *`.
- Call the name-map element’s `oid` member function to find the persistent object that is its value. This function returns a general-purpose object reference, which you can assign to a handle or cast to the appropriate type.
- Call the name-map element’s `set_oid` member function to replace its value with a different persistent object. The parameter is an object reference to the new value object.

You cannot add or delete name-map elements while iterating over a name map. However, you can modify the objects that are referenced by the found name-map elements.

EXAMPLE This example iterates through the elements of a name map of `Book` objects. It prints the name (key) of each element followed by the title of the corresponding book (the value in the key-value pair).

```
// Application code file
#include <ooMap.h>
#include "library.h"
...
ooRef(Book) bookR;
ooHandle(ooMap) mapH;
...                          // Set mapH to reference name map
ooMapItr mapI = mapH;        // Initialize name-map iterator
```

```
while(mapI.next()) {     // Get the next name-map element
   // Print the element's name
   cout << mapI->name();
   // Get an object reference to the element's book
   bookR = static_cast<ooRef(Book) &>(mapI->oid());
   cout << ": " << bookR->getTitle() << endl;
}
```

Group Lookup in Name Scopes

A *name scope* is a group of persistent objects that have unique names within the scope of a particular *scope object*. You group objects into a name scope by naming them in the scope of the scope object. The named objects may be stored in different storage objects from each other. If the scope object is a storage object, the named objects need not be located within that scope object. If the scope object is a basic object, the named objects need not be stored in the same container or database as the scope object. See "Building a Name Scope" on page 333.

Name scopes are designed primarily for individual lookup; you can find a named object by looking up its name in the scope of the appropriate scope object. See "Individual Lookup in Name Scopes" on page 332. If the scope object is a persistent object (container or basic object), you can also perform group lookup of named objects—that is, you can find all objects in the name scope of a particular scope object.

Name scopes also support the ability to find scope objects. Given any persistent object that is named in one or more name scopes, you can find all scope objects that name the object.

The ability to find named objects and scope objects enables you to reestablish scope names when you move a basic object that is used as a scope name or that is named in a name scope. See "Preserving Scope Names" on page 203.

Finding Named Objects

If a scope object is a basic object or a container, you can find all persistent objects in its name scope. To do so, call the `getNameObj` member function on a handle to the scope object, specifying an object iterator of class `ooItr(ooObj)` as the parameter. This function initializes the object iterator to find all persistent objects in the scope object's name scope. The object iterator finds the named objects without opening them.

As you advance the object iterator through the objects in the name scope, you can get the scope name of each object by calling the `getObjName` member function on

a handle to the named object. The parameter to this function is a handle to the scope object.

EXAMPLE The function `moveScopeObject` is called during a transaction to move a basic object that is used as a scope object. Its parameters are handles to the scope object to be moved and the object with which to cluster the moved object. This function creates a temporary name map and copies all scope names to the name map. It then moves the scope object and recreates its name scope by copying the names from the name map. It then deletes the name map.

```
// Application code file
#include <ooMap.h>
...
moveScopeObject(
      ooHandle(ooObj) &scopeH,     // Scope object to be moved
      ooHandle(ooObj) &whereH{     // New location for object
   ooHandle(ooMap) mapH;
   ooItr(ooObj) objI;
   ooRef objR;
   // Create the temporary name map
   mapH = new(whereH) ooMap();
   // Initialize object iterator to find all named objects
   scopeH.getNameObj(objI);
   // Add all named objects to the name map
   while(objI.next) {
      // Add the current object to the name map
      mapH->forceAdd(objI.getObjName(scopeH), objI);
   }
   // Move the scope object
   scopeH->move(whereH);
   // Recreate the name scope
   ooMapItr mapI = mapH;    // Initialize name-map iterator
   // Add all named objects to the name scope
   while(mapI.next()) {
      objR = mapI->oid();
      // Add current object to the name scope
      objR->nameObj(scopeH, mapI->name());
   }
   // Delete the name map
   ooDelete(mapH);
} // End movScopeObject
```

Finding Scope Objects

If you know that a persistent object is named in the scope of at least one scope object, you can find all scope objects whose name scopes contain the object. To do so, call the `getNameScope` member function on a handle to the persistent object, passing an object iterator of class `ooItr(ooObj)` as the parameter. This function initializes the object iterator to find all objects in the federated database that define scope names for the object. The object iterator finds the scope objects without opening them.

As you advance the object iterator through the scope objects, you can get the scope name of the named object in each name scope by calling the `getObjName` member function on a handle to the named object. The parameter to this function is a handle to the scope objects.

EXAMPLE The function `moveNamedObject` is called during a transaction to move a basic object that is grouped into one or more name scopes. Its parameters are handles to the named object to be moved and the object with which to cluster the moved object. This function creates a temporary VArray with handles to the scope objects that have names for the object being moved. It then creates a parallel temporary VArray containing the object's scope name in each name scope. After moving the named object, it reestablishes the object's name in each name scope.

```
// Application code file
#include <oo.h>
...
moveNamedObject(
       ooHandle(ooObj) &namedH,     // Named object to be moved
       ooHandle(ooObj) &whereH{     // New location for object
   ooTVarrayT<ooHandle(ooObj)> scopeObjects;
   ooTVarrayT<char *> scopeNames;
   ooItr(ooObj) objI;
   // Initialize object iterator to find all scope objects
   namedH.getNameScope(objI);
   // Add all scope objects to the first VArray
   int count = 0;
   while(objI.next) {
      scopeObjects.extend(objI);
      count++;
   }
   // Create the VArray of scope names
   scopeNames.resize(count);
   for (int i = 0; i < count; i++) {
      scopeNames.set(i, namedH->getObjName(scopeObjects[i]));
   }
```

```
   // Move the named object
   namedH->move(whereH);
   // Reestablish the scope names
   for (int i = 0; i < count; i++) {
      nameH->nameObj(scopeObjects[i], scopeNames[i]);
} // End moveNamedObject
```

When you work with the general-purpose object iterator initialized by `getNameScope`, remember that a scope object may be an Objectivity / DB object of any type—the federated database, a database, a container, a basic object, or an autonomous partition. For this reason, you should use the object iterator to perform only general-purpose persistence operations that are shared by *all* Objectivity / DB objects. For example, you should not attempt to call the referenced object's `ooIsKindOf` member function, because that function is available for persistent objects only.

At each step through the iteration set, you can test what class the object iterator references as if it were a general-purpose handle. See "Getting the Class of the Referenced Object" on page 221. If you determine that the referenced scope object is a persistent object, you can safely perform any operation that is available through a handle of type `ooHandle(ooObj)`.

EXAMPLE This example iterates over the scope objects for a named object and tests each scope object to determine what kind of Objectivity / DB object it is.

```
// Application code file
#include "myClasses.h"
...
ooHandle(ooObj) objH;
ooItr(ooObj) scopeI;
...     //Set objH to reference the named object
// Initialize object iterator to find all scope objects
objH.getNameScope(scopeI);
while(scopeI.next) {
   typeNum = scopeI.typeN();
   if (typeNum == ooTypeN(ooFDObj)) {
      ...        // Scope object is the federated database
   }
   else if (typeNum == ooTypeN(ooDBObj)) {
      ...        // Scope object is a database
   }
   else if (typeNum == ooTypeN(ooAPObj)) {
      ...        // Scope object is a database
   }
```

```
    // Now the scope object is known to be a persistent object
    else if (objH->ooIsKindOf(ooTypeN(ooContObj))) {
       ...        // Scope object is a container
    }
    else
       ...        // Scope object is a basic object
    }
}
```

18

Content-Based Filtering

Content-based filtering allows you to filter found persistent objects by their content—that is, by the values of their attributes. A filtering operation compares persistent objects against a condition and selects only those that satisfy the condition.

This chapter describes:

- Predicate queries, which perform content-based filtering during the search for persistent objects
- Query objects, which can be used to perform content-based filtering after persistent objects have been found
- Indexes, which can optimize certain predicate queries

Predicate Queries

A *predicate query* performs content-based filtering as an integral part of the search for persistent objects. The query tests the objects to be found against a condition and initializes an object iterator to find only those objects that satisfy the condition. The condition is specified as a *predicate string*—a string in the Objectivity / DB predicate query language. If desired, you can extend this language with application-defined relational operators.

Predicate queries can be used in two operations that find persistent objects:

- You can specify a predicate string when you scan a storage object; such an operation is called a *predicate scan*. See “Scanning for Objects That Satisfy a Condition” on page 362.
- You can specify a predicate string when you find destination objects linked by a to-many association. See “Finding Destination Objects that Satisfy a Condition” on page 326.

Predicate Query Language

The predicate query language supports standard operators and literals and has the ability to refer to attributes (data members) of persistent objects in an Objectivity / DB federated database. It can be extended with application-defined relational operators. It does not provide the ability to declare variables or the ability to call member functions of the objects for which the condition is being tested.

The language ignores any white space and new lines that separate standard tokens; it requires white space around the tokens for application-defined operators. This section describes the syntax of the predicate query language.

Supported Data Types

Predicates can test numeric, character, Boolean, and string values only. As a consequence, a predicate may test a data member only if its C++ data type is one of the following:

- A primitive type such as `uint16`, `int16`, `char`, or `ooBoolean`

  See the Objectivity / C++ Data Definition Language book for a complete list of primitive types, or see “Primitive Type Names” on page 25 of the Objectivity / C++ programmer’s reference.
- A string class: `ooString(`*`N`*`)`, `ooVString`, or `ooUtf8String`
- A fixed-size character array of type `char[]` (treated as a null-terminated string)
- A VArray of characters (treated as a null-terminated string)

Attribute Expressions

An attribute expression gets the value of:

- An attribute of the persistent object being tested.
- An attribute of the destination object of a to-one association of the persistent object being tested.
- A data member of an embedded-class attribute of the persistent object being tested.

Attribute of a Tested Object

Within a predicate string, an unquoted sequence of alphanumeric characters (for example, `employeeName`) is interpreted as an attribute name. This expression evaluates to the value of the named attribute for the persistent object being tested.

The following expression evaluates to the value of the attribute *inheritedAttribute*, which is inherited from the base class *baseClassName*.

baseClassName::*inheritedAttribute*

This syntax is needed if the attribute name is ambiguous (for example, the same name is defined in both the base class and the class of the object being tested) or if the member name is not visible to the object being tested due to access control.

The attribute to be tested must be a numeric, character, Boolean, or string attribute. In particular, a predicate string *cannot* test the value of attributes of the following types:

- Embedded-class types other than the string and VArray types listed above

  Although you cannot test an attribute of this type, you can test a numeric, character, or string data member of an embedded-class attribute. See "Data Member of an Embedded Class" on page 377.
- Object-reference types

Attribute of a Destination Object

A predicate string can test a numeric, character, Boolean, or string attribute of the destination object linked by a to-one association to the persistent object being tested. The following expressions evaluate to the value of the attribute *attributeName* of the destination object of the to-one association *associationName*:

associationName.*attributeName*
associationName->*attributeName*

Data Member of an Embedded Class

A predicate string can test a numeric, character, Boolean, or string data member of the object (or struct) in an embedded-class attribute of the persistent object being tested. The following expression evaluates to the value of the data member *fieldName* of the embedded object (or struct) in the embedded-class attribute *attributeName* of the persistent object being tested:

attributeName.*fieldName*

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 sequence of characters (for example, `"John Doe"`).

There is no special support for a Unicode literal; any quoted sequence of characters is treated as an ASCII string, even if it contains Unicode characters.

- An integer literal has the same syntax as in C++ (for example, `123`).
- A floating-point literal has the same syntax as in C++ (for example, `98.765`).

There are no Boolean literals. To compare an attribute of type `ooBoolean` against a constant value, you must specify an integer literal (1 for `oocTrue`, 0 for `oocFalse`).

Operators

Objectivity/DB supports the operators listed in the following sections; their precedence is the same as the precedence of the equivalent programming-language operators. Parentheses `(` and `)` can be used to override the normal precedence.

Arithmetic Operators

Arithmetic operators produce numeric values; their operands can be numeric literals and attributes of a supported numeric type.

Operator	Description
+	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 numeric, character, or string type. The other operators accept an attribute as the left operand and an attribute or literal as the right operand; both operands must be of the same supported numeric, character, or string type.

Operator	Description
=, ==	Equality
<>, !=	Inequality
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

See also “Application-Defined Relational Operators” on page 383.

String-Matching Operators

String-matching operators produce Boolean values. You use string-matching operators to compare a string to a pattern. The left operand is a string attribute and the right operand is a string literal containing a regular expression (page 380). (To compare exact strings for equality, inequality, and so on, you use the relational operators described in the previous section.)

Operator	Description
=~	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 380.

Logical Operators

Logical operators take Boolean operands and return Boolean values. Typical operands are expressions that use relational or string-matching operators.

Operator	Description
`AND, &&`	Conjunction
`OR, ||`	Disjunction
`NOT, !&`	Negation

The words `AND`, `OR`, and `NOT` are reserved words in the language; you cannot refer to a data member 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

Objectivity / DB string-matching operators test whether a string matches a pattern. A pattern is specified as a *regular expression*. Objectivity / DB implements its regular expressions based on the POSIX extended regular expression library.

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 comparison character `A` in a regular expression matches the character `A` in a string; it is a case-insensitive comparison, it also matches the character `a`.

Metacharacter	Description
`.`	Matches any single character. Loses its special meaning when used within `[]`.
`\`	Used as a prefix operator to override any special meaning of the following character. Loses its special meaning when used within `[]`. **Note:** Within a string in your program, 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 `^`, this pattern matches any character *except* the characters in the sequence and the specified ranges. **Note:** Within `[]`, you can use `[` to match the character `[`, but you must use `\]` to match the character `]`.

Metacharacter	Description
-	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 operator to cause the preceding pattern to be matched zero or more times. Loses its special meaning when used within `[]`.
+	Used as a postfix operator 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 entire string from beginning to end. 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; this use is redundant because a regular expression matches the entire string from beginning to end. No special meaning in other locations in a regular expression.
( )	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

The following predicates demonstrate some of the valid expressions you can use. They test the string attributes `name` and `productName`, the numeric attributes `pins`, `cost`, `padFactor`, and `totalBudget`, and the Boolean attributes `inDesign` and `inProduction`.

```
name = "ALU" AND productName = "Pegasus"
name == "ALU" && productName == "Pegasus"
pins > 3
cost <= 3.50 && NOT inDesign
((cost + 3.50 + padFactor) <= totalBudget) AND inProduction
```

The following predicates test the numeric attribute `size` of the destination object of the to-one association `inGroup` for the source object being tested.

```
inGroup.size == 5;
inGroup->size == 5;
```

The following predicates use regular-expression metacharacters in string comparisons that test the `name` attribute.

```
name =~ \".*H.ll.\"          // anystring H anychar ll anychar
name !~~ \"h.*o\"            // h anystring o
name =~ \"a[0-9]+\"          // a then any number of digit(s)
name =~ \"(abc | def\)"      // abc or def
```

Because a regular expression matches the entire string, the following predicates are equivalent ways to test that the `license` attribute begins with the characters `ca`:

```
license =~ \"ca.*\"
license =~ \"^ca.*\"
```

The following expressions demonstrate *invalid* predicate strings (assuming that no application-defined relational operators have been defined).

```
name == ALU     // Error! ALU is not an attribute (use "ALU")
pins >* 3       // Error! Unrecognized operator >*
((cost + 3.50 + padFactor) <= totalBudget AND inProduction
                // Error! Missing close parenthesis
```

Application-Defined Relational Operators

In addition to the standard relational operators of the Objectivity / DB predicate query language, an application can define its own relational operators for use in predicates. The relational operators defined by an application have the same precedence, relative to other kinds of operators, as the relational operators defined by Objectivity / C++. You must use whitespace to separate application-defined operators from their operands.

To make a relational operator available for use in predicates, you must:

- Define an operator function that tests whether two operands are of an appropriate type and whether the desired relation holds between them.
- Register the operator function and the token representing it with the Objectivity / DB predicate-testing mechanism.

When the registered token appears in a predicate string, the predicate-testing mechanism identifies the relevant operands and passes them, along with type information, to the operator function you registered.

Defining an Operator Function

An operator function is an application-defined function that must conform to the calling interface defined by the `ooQueryOperatorPtr` function pointer type. The operator function must accept four parameters—two for the operands to be compared and two for data-type indicators corresponding to these operands. The operator function should return `oocTrue` if the two operands are of the expected type and the desired relation holds between them; otherwise, the function should return `oocFalse`.

The predicate-testing mechanism calls the operator function after parsing and preparing the operands to be compared. Because the predicate-testing mechanism recognizes only a restricted set of data types, it converts operands of other valid data types to one of the recognized data types. Pointers to the operands are then passed to the operator function, along with constants of the global type `ooDataType` that indicate the operands' data types. The following table shows, for each actual operand type, the data type to which it is converted (if conversion is required), and the corresponding data-type indicator.

Actual Operand Types	Data Types Recognized by Predicate Query	Corresponding ooDataType Constant
int8 int16 int32 int64	int64	oocInt64T
uint16 uint32 uint64	uint64	oocUint64T
float32 float64	float64	oocFloat64T
char* char[] ooString(*N*) ooVString ooUtf8String ooVArrayT<char>	char *	oocCharPtrT
ooBoolean	ooBoolean	oocBooleanT
Any other type		oocInvalidTypeT

Your operator function should:

- Use the ooDataType constants to verify that the operands are of a type appropriate to your operator.
- Compare the operands. Because the function accepts void * pointers to the operands, it must cast each pointer to the appropriate recognized data type.

EXAMPLE This example defines an operator function `sameLen` that tests whether two string operands are nonnull and of the same length. The `void *` pointers `a` and `b` are cast to `char *`, even if the operands were originally of another string type.

```
// Application code file
#include <oo.h>
...
ooBoolean sameLen(const void *a,   // Left operand
                  const void *b,   // Right operand
                  ooDataType aT,   // Type of left operand
                  ooDataType bT)   // Type of right operand
{
   // Verify that operands are of the correct types
   if (aT != oocCharPtrT || bT != oocCharPtrT) {
      return oocFalse;
   }
   // Verify that both operands are nonnull
   if (!a || !b) {
      return oocFalse;
   }
   // Compare the lengths of the two operand strings
   if (strlen(static_cast<const char *>(a) ==
       strlen(static_cast<const char *>(b))) {
      return oocTrue;
   }
   return oocFalse;
}
```

Registering an Operator Function

After you create an operator function, you must make it available to the predicate-testing mechanism. You do this by registering it with the *operator set* (an instance of the `ooOperatorSet` class) pointed to by the global variable `ooUserDefinedOperators`.

To register an operator function with an operator set, you call the `registerOperator` member function on the operator set. Typically, you call this function on the default operator set, to which `ooUserDefinedOperators` points when the application starts. However, you can register the operator function with a different operator set and then set `ooUserDefinedOperators` to point to that operator set.

The parameters to `registerOperator` are:

- A token that is to represent the operator in a predicate string. The token is a sequence of characters that may *not* begin or end with any of the following symbols or symbol combinations:

  `) ( && || ! , .`

- A function pointer to the operator function.

If you specify a token that is the same as an existing operator (for example, +), your operator function will override the standard behavior of that operator.

EXAMPLE This example registers the operator function `sameLen` (defined in the previous example) along with the token `@@`. A predicate scan then uses `@@` to find all `myClass` objects whose `name` values are of the same nonzero length as the string specified by `str`. Whitespace (for example spaces or tabs) must be used to separate the token from its operands.

```
// Application code file
#include "myClasses.h"
…
// Register the sameLen function and the @@ token.
ooUserDefinedOperators->registerOperator("@@", sameLen);
…
ooItr(myClass) itrI;
char *str = …;
char pred[64];
sprintf(pred, "name =~ \"He.*\" and name @@ %s", str);
itrI.scan(fdbH, pred);

while (itrI.next()) {
   // Process all myClass objects with a name string
   // beginning with "He" and of the same length as str
   …
}
```

Operator Sets

An operator set is empty at creation, and you can register any number of operator functions with it. At any point in the application, you can call its `clear` member function to remove all the currently registered operators from the set.

When an Objectivity/C++ application starts, the global variable `ooUserDefinedOperators` is initialized to point to a new operator set, called the default operator set.

You can create any number of nondefault operator sets by instantiating the class ooOperatorSet (for example, to register alternative sets of operator functions). However, only one operator set can be in effect at a time; if you create a local operator set, you must assign it to the ooUserDefinedOperators global variable to make its operators available to Objectivity / DB.

The ooUserDefinedOperators global variable is shared by all Objectivity contexts, but it is *not* thread-safe. Your application must ensure that:

- Only one thread updates the operator set at a time.
- If a thread is updating the operator set, no other thread can be making a predicate query at the same time.

To ensure correct use of the ooUserDefinedOperators global variable, you should set this variable and register your operator functions as part of your application's initialization process—that is, after you call ooInit and before you perform any persistence operations.

Query Objects

You use a *query object* to filter an arbitrary group of persistent objects based on their content. A query object is an instance of the non-persistence-capable class ooQuery; it contains a predicate string to be tested against objects of a specified class or its descendant classes. For example, if a particular persistent collection contains persistent objects of the relevant class, you can find the objects in the collection and then use a query object to test its predicate for each found object.

To filter objects that cannot be found by a predicate query:

1. Create a query object by instantiating the class ooQuery.
2. Set up the query object by calling its setup member function. The parameters specify the predicate string and the class of objects to be tested. All attributes in the predicate string must be defined in, or inherited by, the specified persistence-capable class.
3. Use the query object to test whether persistent objects satisfy the predicate. To do this, call the query object's evaluate member function, passing a handle to the persistent object to be tested.

EXAMPLE This example creates a query object to test whether `Rectangle` objects satisfy the predicate string `length > 4 && width = 3`.

```
// DDL file geometry.ddl
...
class Rectangle : public ooObj{
public:
   int32 length, width, area;
   ...
};
```

```
// Application code file
#include "geometry.h"
...
ooQuery *myQuery = new ooQuery;
myQuery->setup("length > 4 and width = 3", ooTypeN(Rectangle));
```

In this example, a given unordered set is used to group `Rectangle` objects. The query object finds elements of the set that match the predicate string.

```
// Application code file
#include "geometry.h"
...
ooHandle(ooHashSet) setH;
ooCollectionIterator *setI;
ooHandle(Rectangle) rectH;
ooRef(Rectangle) recttR;
ooRef(ooObj) objR;

...    // Set setH to reference the unordered set of rectangles
// Initialize a scalable-collection iterator to find
// elements of the set
setI = setH->iterator();
// Filter the elements of the set
while (setI->hasNext()) {
   objR = setI->next();
   if (objR->ooIsKindOf(ooTypeN(Rectangle))) {
      rectR = static_cast<ooRef(Rectangle)>(objR);
      rectH = rectR;
      if (myQuery->evaluate(rectH) {
         ... // Do something with this rectangle
      } // End if condition succeeds
   } // End if element is a Rectangle object
} // End while set has more elements
delete setI;
```

The following code finds the same objects, but without using a query object.

```
ooHandle(ooHashSet) setH;
ooCollectionIterator *setI;
ooHandle(Rectangle) rectH;
ooRef(Rectangle) recttR;
ooRef(ooObj) objR;

...     // Set setH to reference the unordered set of rectangles
// Initialize a scalable-collection iterator to find
// elements of the set
setI = setH->iterator();
// Filter the elements of the set
while (setI->hasNext()) {
   objR = setI->next();
   if (objR->ooIsKindOf(ooTypeN(Rectangle))) {
      rectR = static_cast<ooRef(Rectangle)>(objR);
      rectH = rectR;
      if ((rectH->length > 4) && (rectH->width == 3)) {
         ... // Do something with this rectangle
      } // End if condition succeeds
   } // End if element is a Rectangle object
} // End while set has more elements
delete setI;
```

By default, a query object's predicate string can use any standard relational operators of the Objectivity / DB predicate query language. If the predicate string is to include any application-defined relational operators that have been registered with an operator set, you can pass a pointer to that operator set as a parameter to the query object's `setup` member function. For example, if the predicate string is to use a relational operator that is registered with the default operator set, you would pass the global variable `ooUserDefinedOperators` to the query object's `setup` member function (see "Application-Defined Relational Operators" on page 383).

Indexes

If you expect applications to perform predicate scans that test particular attributes, you can speed those scans by defining indexes that order the objects by the values of those attributes.

Understanding Indexes

An *index* is a data structure that maintains object references to the persistent objects of a particular *indexed class* and its derived classes within a particular storage object; these objects are called its *indexed objects*. An index sorts the indexed objects according to the values in one or more of their attributes, 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.

When an application performs a predicate scan for which there is a relevant index, the scan operation examines only the indexed objects, instead of examining *all* persistent objects of the relevant classes in the storage object being scanned.

An index can be thought of as a sorted collection of object references to its indexed objects. For predicate scans to work correctly, the index must correctly order the correct objects. When an object of the indexed class or a derived class is created in the relevant storage object, the newly created object must be inserted into the index; when an existing indexed object is deleted, it must be removed from the index; when the value of an indexed object's key field is modified, the object must be moved to the appropriate position in the sorting order of the index. An application can control when it updates indexes, relative to when indexed objects are created, deleted, and modified.

Relevant Indexes for an Application

You identify the indexes that are relevant for your application by considering what predicate scans the application will perform.

An index is appropriate only when it can prune the search for objects that satisfy a predicate. If a large number of objects satisfy the predicate, the total time to traverse the index data structure can be longer than the linear search that would be performed without the index. (The exact number of objects at which an index becomes less efficient is application-specific.)

The number of disk reads needed to locate an object through an index is directly related to the number of levels in the index, which is directly related to the number of index objects in the index. You can help minimize the number of disk reads needed to locate an object by indexing at the appropriate level of the inheritance hierarchy. That is, you should define the index on the most specific

class possible. For example, suppose a hospital application includes a class called `Employees`, which has two derived classes called `Doctors` and `Nurses`. If the application will perform a predicate scan to find doctors (but not nurses), it could include an the index on the `Doctors` class. If the application will perform one predicate scan to find doctors and a different predicate scan to find nurses, it could include one index on the `Doctors` classes and another index on the `Nurses` class. If the application will perform a predicate scan that finds both doctors and nurses, then (and only then) it should include an index on the `Employees` class.

In cases where an index is appropriate for the predicate scans that will be performed, 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. See “Updating Indexes” on page 402.

Creation and Use

Indexes are created dynamically by application programs and continue to exist until they are explicitly removed by application programs. See “Creating an Index” on page 396 and “Dropping Indexes” on page 405.

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. An application that performs such a scan must explicitly enable index usage in order to take advantage of the index to optimize its search. See “Enabling and Disabling Indexes” on page 402.

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 may repeatedly scan using different predicates that test the same combination of attributes. An index on those attributes 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.

Multiple transactions can read a given index concurrently, using it to optimize searches for the indexed objects. However, only one transaction at a time can update the index or its objects. If one transaction is updating an index, other clients can read the index in MROW transactions.

Objectivity / DB ensures that any objects that are found using an index are valid objects. This is accomplished by locking the corresponding objects in read mode (by default) or in update mode until the end of the transaction. If your application requires a lower level of consistency but a higher concurrency of operations on a given index, then you can use an MROW transaction to find the indexed object. In an MROW transaction, objects that are found using an index exist in the current application's version of the container. For more information about MROW transactions, see "Multiple Readers, One Writer (MROW) Policy" on page 114.

Key Descriptions

You define an index by creating a *key description*—a persistent object that describes the class of objects to be indexed, the key fields on which to sort the indexed objects, and whether the index is to be unique. The key description is then used to create an index for a particular storage object. The storage object you choose for an index limits the objects referenced by that index—for example, an index created for a container references the objects of the indexed class that reside in that container. When an index is no longer needed, you use its key description to remove or *drop* it from the storage object.

Indexed Class

The class of objects to be indexed is specified when you create a key description. This class is known as the *indexed class*. You can define a key description on any persistence-capable class.

An index created from a key description contains references to objects of the indexed class and objects of classes derived from the indexed class. If the indexed class maintains a version genealogy, the index includes all versions that reside in the storage object on which the index is defined; the index does not distinguish default from nondefault versions.

Key Fields

The key fields of an index are the data members whose values are used for sorting the indexed objects. A key field may be any C++ `private`, `protected`, or `public` data member whose data type can be used in a predicate query—namely, a numeric, character, Boolean, or string data member. See "Supported Data Types" on page 376.

Key fields are represented by the *key-field objects* that you add to the key description. Each key-field object identifies one of the following:

- A particular attribute data member of the indexed class.

- A particular data member of the embedded class of an embedded-class attribute of the indexed class.

Indexes can optimize conditions that compare a key field with a literal numeric or string value.

Sorting Order

Indexed objects are sorted by ascending order of the values in their key fields; the key fields are considered in the order in which the corresponding key-field objects were added to the key description. For example, assume you want to create an index over `Person` objects, sorted by key fields `name` and `age`. You achieve this by creating key-field objects corresponding to the `name` and `age` data members of class `Person`. You add first the `name` key-field object then the `age` key-field object to an appropriate key description. In the resulting index, objects are first sorted by name; if two or more objects have the same name, the one with the lowest age comes first in the indexed order.

Different indexes can sort the same objects in different orders. The key fields and their order within each index determine the order of the indexed objects in each index.

An index determines the order of string values by calling the standard C function `strcoll` to compare them.

You can specify an international collation for string values by using the proper locale table configuration and calling the `setlocale` function with the proper parameters. For more information, refer to your operating system documentation.

NOTE You should use the same locale both for index creation and for lookup, especially if the creation and lookup are performed in different processes or threads.

Unique and Nonunique Indexes

A key description determines whether its indexes are *unique* or *nonunique*. Every object indexed by a unique index must have a unique combination of values in its key fields (that is, no duplicate objects). 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.

An error is reported by a unique index if it attempts to index two versions of an object with the same key-field values.

You can call the `isUnique` member function on an index's key description to find out whether the index ignores or includes duplicate objects.

Optimized Predicate Scans

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 forms shown in the following table.

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 type. *stringConstant* is a string constant that begins with a nonwildcard character.

In each row of the following table, 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 optimize 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; predicate is a conjunction starting with an optimized conditions that tests the first key field.
`(age > 40) && (age != 60)`	`age`	Yes; predicate is a conjunction starting with an optimized conditions that tests the first key field.
`(age > 40) OR (age < 60)`	`age`	No; predicate is a disjunction.
`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`).
`(weight >& 100) && (height > 60)`	`weight, height`	Yes; first condition tests first key field.
`name =~ "Me.er"`	`name`	Yes; pattern begins with nonwildcard 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

Your application must include the ooIndex.h header file to use key descriptions and key fields.

To create one or more indexes for a class, you perform the following steps:

1. Create a key description (an instance of class ooKeyDesc); see "Creating a Key Description" on page 397.
2. Create a key-field object (an instance of class ooKeyField) for each data member whose values are to be used for sorting; see "Creating Key-Field Objects" on page 398.
3. Add each key-field object to the key-description object in the order you want to sort the indexed objects; see "Adding a Key-Field Object to the Key Description" on page 399.
4. Use the key description to create an index for a particular storage object; see "Creating an Index from the Key Description" on page 400.

The examples in the following sections cumulatively create a nonunique index in 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`.

Creating a Key Description

A key description is a persistent object that you use for defining, creating, and dropping indexes for the objects of a particular class and its derived classes. A key description is an instance of the persistence-capable class `ooKeyDesc`. The `ooKeyDesc` constructor specifies the class of objects to be indexed and whether the indexes will be unique or nonunique.

As for any persistent object, you specify a clustering directive when you create a key description, and you work with the key description through a handle. See "Creating a Basic Object" on page 184.

You can improve performance by clustering the key description in an appropriate container based on the storage object containing the objects to be indexed:

- If the index will be created over the objects in a container, you should cluster the key description in that container.
- If the index will be created over the objects in a database, you should cluster the key description in the default container for that database.
- If the index will be created over the objects in the federated database, you can cluster the key description in any container in the federated database.

EXAMPLE This example creates a key description for a nonunique index on objects of class `Client` and its derived classes. Because the index will be created on objects in the container `clientCont`, the key description is clustered in that container.

```
// Application code file
#include "myClasses.h"
...
ooHandle(ooContObj) clientContH;
ooHandle(ooKeyDesc) kDescH;
...     // Set clientContH to reference the container clientCont

// Create the key description clustered in clientCont.
kDescH = new(clientContH) ooKeyDesc(ooTypeN(Client), oocFalse);
```

If you plan to create multiple indexes at different points in your program, you can keep track of the key description for later retrieval by assigning it a scope name or by storing a handle to it somewhere in your program.

Creating Key-Field Objects

A key field is a data member that will serve as a sort key for the indexed objects. You create one key-field object for each desired key field. (You designate key fields as primary, secondary, and so on, when you add the corresponding key-field objects to a key description; see “Adding a Key-Field Object to the Key Description” on page 399.)

A key-field object is an instance of the persistence-capable class `ooKeyField`. The `ooKeyField` constructor specifies the indexed class; this class must match the class you specified when creating the key description. The constructor also specifies the data member on which the indexed objects will be sorted:

- To indicate the data member *`attributeName`* of the indexed class, use a string of the form:

  "*`attributeName`*"

- If the name of an inherited data member is ambiguous (for example, the same name is defined in both the base class and the indexed class) or if the member name is not visible to the indexed class due to access control, use a string of the following form to indicate the data member *`inheritedAttribute`*, which the indexed class inherits from the base class *`baseClassName`*:

  "*`baseClassName`*`::`*`inheritedAttribute`*"

- To indicate the data member *`fieldName`* of the embedded object (or struct) in the embedded-class attribute *`attributeName`* of the indexed class, use a string of the form:

  "*`attributeName`*`.`*`fieldName`*"

As for any persistent object, you specify a clustering directive when you create a key-field object, and you work with the key-field object through a handle. See “Creating a Basic Object” on page 184. For best performance, you should cluster each key-field object with the key description to which it will be added.

EXAMPLE This example creates a key-field object for the `state` data member of class `Client`. The key-field object is clustered with the key description to which it will be added.

```
// Application code file
#include "myClasses.h"
...
ooHandle(ooContObj) clientContH;
ooHandle(ooKeyDesc) kDescH;
ooHandle(ooKeyField) kFieldH;
...    // Set clientContH to reference the container clientCont

// Create the key description clustered in clientCont.
kDescH = new(clientContH) ooKeyDesc(ooTypeN(Client), oocFalse);

// Create the key field clustered with the key description
kFieldH = new(kDescH) ooKeyField(ooTypeN(Client), "state");
```

When you create a key-field object for a string-typed data member, you can optimize the space required to store each string key, as well as the processing time required to access it. See “Optimizing String-Key Storage and Lookup” on page 406.

Adding a Key-Field Object to the Key Description

After creating each desired key-field object, you add it to the key description by calling the `addField` member function on the key description. You can add a key-field object to a key description only if both are defined on the same class. The order in which you add key-field objects to a key description defines the sorting order of the indexed objects—the first key field you add becomes the primary sort key, the second key field is the secondary sort key, and so on.

EXAMPLE This example creates a key description for class `Client`, creates key-field objects for the `state`, `city`, and `zipCode` data members of class `Client`, and adds these key-field objects to the key description. The order in which the key-field objects are added will cause the index to sort `Client` objects first by state, then by city within each state, and then by zip code within each city.

```
// Application code file
#include "myClasses.h"
...
ooHandle(ooContObj) clientContH;
ooHandle(ooKeyDesc) kDescH;
ooHandle(ooKeyField) kFieldH;

... // Set clientContH to reference the container clientCont
// Create the key description clustered in clientCont
kDescH = new(clientContH) ooKeyDesc(ooTypeN(Client), oocFalse);

// Create the key fields and add them to the key description
kFieldH = new(kDescH) ooKeyField(ooTypeN(Client), "state");
kDescH->addField(kFieldH);

kFieldH = new(kDescH) ooKeyField(ooTypeN(Client), "city");
kDescH->addField(kFieldH);

kFieldH = new(kDescH) ooKeyField(ooTypeN(Client), "zipCode");
kDescH->addField(kFieldH);
```

Although it is possible to add the same key-field object to different key descriptions, you should avoid doing so if you ever intend to delete one of the key descriptions. When you delete a key description, all its key-field objects are deleted.

Creating an Index from the Key Description

When the key description is complete, you use it to create an index over all the objects of the indexed class that reside in a particular storage object. To do this, you call the `createIndex` member function on the key description, specifying a handle to a container, a database, or the entire federated database. If the call to `createIndex` fails, a message is displayed and the transaction is aborted.

A storage object can have at most one index created from a given key description, although you can use the same key description to create additional indexes over objects in other storage objects.

EXAMPLE This example uses the key description created in the preceding example to create an index over all the `Client` objects in the `clientCont` container.

```
// Application code file
#include "myClasses.h"
...
ooHandle(ooContObj) clientContH;
ooHandle(ooKeyDesc) kDescH;

// Create the key description and add key fields
kDescH = ...
...     // Set clientContH to reference the container clientCont
kDescH->createIndex(clientContH);
```

The following statements use the same key description to create an index over the `Client` objects in the database referenced by `clientDBH`.

```
ooHandle(ooDBObj) clientDBH;
...     // Set clientDBH to reference a database.
kDescH->createIndex(clientDBH);
```

Keeping Track of Long-Lived Indexes

If your application uses long-lived indexes, you should keep track of how you created them in case you ever need to reconstruct them. Certain schema changes can invalidate existing key fields and key descriptions, causing the corresponding indexes to be dropped. If this occurs, you may want to replace the dropped indexes. See “Reconstructing Indexes After Schema Evolution” on page 408.

A simple approach to keeping track of the indexes is to isolate the necessary information in special application code that you can modify, and rerun as necessary. For example, you might write a separate function for each indexed class that creates all indexes for that class. If the indexed data members of an indexed class are ever modified during schema evolution, you can edit the index-creation function for that class as required to replace or recreate any indexes that were dropped.

An alternative is to keep the information about indexes explicitly in the federated database itself. You could then use this information to determine which indexes need to be reconstructed and to find the key fields that need to be replaced or

modified to create new, consistent key descriptions. For each key description, you would need to save the following information:

- The storage objects over which an index was created with this key description
- The key fields of this key description

Enabling and Disabling Indexes

You can control whether an application uses indexes when performing predicate scans. The use of indexes is disabled by default. You can enable and later disable the use of indexes within a particular Objectivity context; to do so, you call the `ooUseIndex` global function within the thread.

- When indexes are disabled, no predicate scans 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 Predicate Scans” on page 394.

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.

Updating Indexes

An index sorts the indexed objects and uses its ordering to determine which objects satisfy a particular predicate scan. In order for a predicate scan 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 in the relevant storage object, it must be inserted into the index; when an existing indexed object is deleted, it must be removed from the index; when the value of an indexed object’s key field is modified, the object must be moved to the appropriate spot in the sorting order of the index.

Choosing a Policy for Updating Indexes

You can control when indexes are updated, relative to when indexed objects are modified. A transaction's *index mode* specifies the policy for updating indexes in response to changes made during the transaction. You set the index mode for a transaction by specifying the *indexMode* parameter of the `start` member function on the transaction object. Index modes are specified by the following constants of type `ooIndexMode`.

Index Mode	Meaning
`oocInsensitive`	When the transaction is committed, indexes are updated automatically. This is the default index mode.
`oocSensitive`	Indexes are updated automatically when the next predicate scan is performed during the transaction. This allows you to change indexed objects and scan them in the same transaction. If no predicate scan is performed, indexes are updated, as necessary, when the transaction is committed.
`oocExplicitUpdate`	The application must update indexes explicitly by calling the `ooUpdateIndexes` function after a relevant change. Explicit updates are recommended for update-intensive applications that use indexes created over a database or the federated database. **Warning:** If the application modifies indexed objects and never calls `ooUpdateIndexes`, the indexes are left inconsistent at the end of the transaction.

In all index modes, updates to the index are rolled back if the transaction is aborted.

Explicitly Updating Indexes

If you choose the `oocExplicitUpdate` index mode for a transaction, you must update indexes explicitly after creating an object of an indexed class or modifying a key field of an indexed object. To do this, you call the `ooUpdateIndexes` function, specifying a handle to the new or modified object for which indexes are to be updated. `ooUpdateIndexes` finds all the indexes that apply to the specified object. If you fail to call `ooUpdateIndexes` after making a change that affect indexes, the indexes are left in an inconsistent state at the end of the transaction.

EXAMPLE This example creates a new `Client` object and calls `ooUpdateIndexes` to insert the object into all applicable indexes.

```
// Application code file
#include "myClasses.h"
...
ooTrans trans;
ooHandle(Client) clientH;
ooHandle(ooContObj) clientContH;
...
// Start a transaction whose index mode is explicit update
trans.start(oocNoMROW, oocTransNoWait, oocExplicitUpdate);
...    // Set clientContH to reference the container clientCont

// Create a new Client.
clientH = new(clientContH) Client();
... // Initialize the Client object's data members

// Insert the new Client object into all applicable indexes
ooUpdateIndexes(clientH);
...
trans.commit();
```

Using explicit-update mode instead of sensitive mode can improve performance in update-intensive applications that use indexes defined on a database or the federated database. See "Updating Indexes Explicitly" on page 515.

Concurrency and Index Updates

Although multiple transactions can read an index concurrently, only one transaction at a time can update an index:

- When the index is over a container, an update lock is obtained on the container; this lock is retained until the transaction commits. MROW transactions can obtain concurrent read locks while the update lock is held. For more information about MROW transactions, see "Multiple Readers, One Writer (MROW) Policy" on page 114.
- When the index is over a database or the federated database, an update lock is obtained on the database or federated database. This lock is retained until the index has been updated, which may occur before the transaction commits; thus, a transaction can update the index without having to wait for other index-updating transactions to commit. Concurrent read locks are allowed while the update lock is held.

Dropping Indexes

Indexes are stored in the federated database until an application removes, or *drops*, them. For example, you might want to drop an index to recreate it with different characteristics, or you might want to reclaim the space it used.

You can drop a single index created from a particular key description in a particular storage object. To do this, you call the `dropIndex` member function on the key description, specifying a handle to the storage object from which the index is to be dropped.

EXAMPLE This example drops the index for a particular key description from the container referenced by `clientContH`.

```
// Application code file
#include "myClasses.h"
...
ooHandle(ooKeyDesc) kDescH;
...     // Set clientContH to reference the container clientCont
...     // Set kDescH to reference the desired key description
kDescH->dropIndex(clientContH);
```

You can drop all the indexes created from a particular key description. To do this, call the `removeIndexes` member function on the key description. Alternatively, if you no longer need the key description, you can delete it with the `ooDelete` function, which automatically deletes all its indexes.

EXAMPLE This example drops all indexes for a particular key description from all storage objects:

```
// Application code file
#include "myClasses.h"
...
...     // Set kDescH to reference the desired key description
kDescH->removeIndexes();
```

Alternatively, you can delete the key description object, which also deletes any indexes created from the key description, and the key-field objects used by it:

```
// Application code file
#include "myClasses.h"
...
...     // Set kDescH to reference the desired key description
ooDelete(keyDescH);
```

Optimizing String-Key Storage and Lookup

When you create a key-field object for a string-typed data member (a member of type `char[]`, `ooString(N)`, `ooVString`, `ooVArrayT<char>`, or `Utf8String`), you can optimize the space required to store each string key, as well as the processing time required to access it. By default, the index itself allocates 24 bytes to store each string key; longer string keys are stored separately (outside the index). During lookup, these bytes are compared to the first 24 bytes of the comparison value; if they match, the index obtains the entire string value from the indexed object to complete the comparison.

You can adjust the storage characteristics for string-typed key fields with parameters to the `ooKeyField` constructor's parameters.

The *`maxstrlen`* parameter to the `ooKeyField` constructor specifies the number of bytes to be allocated by the index for storing each string key; it is set to 24 by default. You can specify a different length to reduce the size of the index or to improve performance:

- If most of the string keys are smaller than 24 bytes, you can set *`maxstrlen`* to a smaller value to reduce the amount of extra space in the index.
- If most of the string keys are larger than 24 bytes, you can set *`maxstrlen`* to a larger value to reduce the number of times that the index must open an indexed object to get a complete string value.

If the string values of the indexed objects are guaranteed to be of a fixed (or limited) size, you can improve performance by setting the *`fixed`* parameter to `oocTrue`. This prevents an object from being indexed if its string value is greater than *`maxstrlen`*, so every string key in the index is complete; during lookup, comparisons are performed without accessing the actual string value in the indexed object.

WARNING Do not set *`fixed`* to `oocTrue` if any string value might be longer than *`maxstrlen`*. If the string value of an object's key field exceeds *`maxstrlen`*, that object is omitted from the index.

When designing a class that is to be indexed on a string-typed key field, consider using a fixed-size array of characters (`char[]`) for that field, because key fields on variable-length strings with no maximum length may incur an extra page read per indexed object during execution. Remember, however, that applications written in Java and Smalltalk cannot access classes containing fixed-size arrays.

For optimal results, use the following guidelines:

- If the key field is a fixed-size string of length *N*, you should set *fixed* to oocTrue and *maxstrlen* to *N*.
- If the key field is a variable-size string such as an ooVString and you know that most of the string values (for example, 90%) are of length *N* or less, you should set *fixed* to oocFalse and *maxstrlen* to *N*.

EXAMPLE This example creates a key-field object on an optimized string of length 8 (ooString(8)), a type that was chosen because most Person objects have a name whose length is 7 bytes or less. You can save space in the index by setting the *maxstrlen* parameter of the ooKeyField constructor to 8. However, because it is possible for some string values to be longer than 8 bytes, the *fixed* parameter is oocFalse.

```
// DDL file person.ddl
class Person : public ooObj {
   ooString(8) name;     // Most strings are < 8 bytes long
};
```

```
// Application code
#include "person.h"
...
ooHandle(ooKeyField) keyFieldH;
ooHandle(ooKeyDesc) keyDescH;
ooHandle(ooContObj) contH;
...    // Set contH to reference the appropriate container

// Create the key description
keyDescH = new(contH) ooKeyDesc(ooTypeN(Person),oocTrue);
// Create the key field, clustered with the key description
keyFieldH = new(keyDescH)
                ooKeyField (ooTypeN(Person), // Indexed class
                            "name",   // Key attribute
                            oocFalse, // Allow variable length
                            8);       // Optimize for 8 chars
```

Index Scans

As an alternative to performing a predicate scan, an application can scan a storage object with a *lookup key* that represents the condition to be met. Such a scan, called an *index scan*, initializes an object iterator to find objects by searching a compatible index on the specified storage object. An index scan can search an index even if indexes are disabled (page 402); disabling indexes affects only predicate scans, not index scans.

Predicate scans are typically more useful than index scans because:

- A predicate scan can find objects in the scanned storage object even if no compatible index exists. In contrast, if there is no compatible index for an index scan, no objects are found—even if the specified storage object actually contains objects that satisfy the lookup key's condition.
- All objects found by a predicate scan are guaranteed to satisfy the condition specified by the predicate string. In contrast, if a lookup key specifies a conjunction of more than one condition, the found objects may satisfy some, but not all, of those conditions.
- A predicate string (used by a predicate scan) can specify any condition that can be expressed in the predicate query language, including conditions that use application-defined relational operations. In contrast, a lookup key (used by an index scan) cannot perform string matching or test an application-defined relational operator.

Your application must include the `ooIndex.h` header file to use a lookup key. For information on creating a lookup key and using it to perform a predicate scan, see class `ooLookupKey` in the Objectivity/C++ programmer's reference.

Reconstructing Indexes After Schema Evolution

If schema evolution modifies the data members of an indexed class, you may need to recreate the indexes on that class. The following schema changes invalidate existing key fields and key descriptions, causing the corresponding indexes to be dropped:

- Deleting a data member that is a key field in a key description.
- Changing the type of a data member that is a key field in a key description.

If the data member that was deleted or changed is defined by the indexed class, that class itself is modified during schema evolution. If the data member is inherited, however, the indexed class need not be modified. Instead, the base class defining the data member could be deleted, replaced, or modified.

If you have evolved the schema of a federated database that contains indexes, you should write a maintenance application to reconstruct any affected indexes. For example, if you delete one of three key fields from the indexed class, you could create a new index with the other two key fields. If the data type of one of

the key fields is changed, you could recreate the index with the same collection of key fields, but specify the new type for the changed field.

You can determine which classes had indexes that were dropped as follows:

1. Call the `scan` member function on an object iterator of class `ooItr(ooKeyDesc)` to initialize the object iterator to find all key descriptions in the federated database.
2. As you iterate through the key descriptions, call the `isConsistent` member function of each one to test whether it is consistent with the evolved schema.
3. For each inconsistent key description, call its `getTypeName` member function to get the name of the indexed class.

Once you have identified the classes whose indexes were dropped, you need to create new, consistent key descriptions and key fields and recreate the desired indexes. Doing so requires that you know what indexes existed for each class. You need to compare the new data members of the indexed class with the key fields in each original index, and recreate any indexes that were dropped as a result of schema evolution. To save space in the federated database, you may delete any inconsistent key description that you replace or that you no longer need; doing do deletes all its key fields.

Reconstructing indexes is easier if you keep track of information about all long-lived indexes in the federated database. See “Keeping Track of Long-Lived Indexes” on page 401. For example, if you wrote a separate index-creation function for each indexed class, you could edit the function for a particular class as required to replace or recreate any indexes that were dropped as a result of schema evolution. You could then call the updated function in a maintenance application to recreate the indexes.

If you have stored information in the federated database about its long-lived indexes, you can perform further testing on that information to determine which key fields need to be modified or removed when you new create key descriptions for updated indexes. For example, when you identify that a key description is inconsistent with the evolved schema, you could iterate over the key fields to see which of them are inconsistent or which correspond to data members that you know are deleted or changed:

1. As you iterate through the key fields, call the `isConsistent` member function of each to test whether it is consistent with the evolved schema.
2. Call the `getName` member function of a key field to get the name of the data member to which it corresponds or call the `isNamed` member function to test whether it corresponds to a particular data member.

Part 5 SPECIAL TOPICS

This part discusses topics that are of interest to some, but not all, users of the Objectivity/C++ programming interface.

19

Object Conversion

When you evolve a class description in the schema of a federated database, existing objects of that class must be *converted* so that they are consistent with the modified class description. To preserve consistency, Objectivity/DB automatically converts such objects the first time they are accessed by an application. In many cases, automatic object conversion is sufficient; in some cases, however, you must augment automatic object conversion by employing one or more explicit object-conversion mechanisms—for example, to meet your performance and availability requirements, or to complete certain schema-evolution operations.

This chapter describes:

- General information about object conversion
- Automatic object conversion
- Converting objects on demand
- Setting the values of primitive data members during object conversion
- Releasing classes from upgrade protection after certain schema-evolution operations
- Updating indexes affected by schema changes
- Purging schema-evolution history

NOTE You should use the information in this chapter only after reading the schema evolution chapter in the Objectivity/C++ Data Definition Language book.

Understanding Object Conversion

Object conversion is the process of making existing objects in a federated database consistent with changes to class descriptions introduced by schema evolution. Schema evolution occurs when you modify a class declaration in a DDL file and process the file, specifying the `-evolve` option to the DDL processor. The DDL processor generates header files with the updated definition of the classes and it modifies their descriptions in the federated database schema.

Object conversion is required after every *conversion operation* you perform on the schema. A conversion operation is a schema change that affects the *shape* of objects of the modified class—that is, how objects of the class are laid out in storage. For example, adding a data member to a class increases the amount of space that must be allocated for each object of that class.

When you perform conversion operations during schema evolution, any existing *affected objects* are rendered out-of-date until they are converted to their new shapes. At a minimum, the affected objects for a given conversion operation include all objects of the class whose description was changed. In a typical database, other objects are affected, too—namely, objects of classes derived from a changed class, objects that embed objects of a changed class, and so on. Thus, when a data member is added to a base class, additional space must be inserted into the objects of every derived class, too.

Objectivity / DB preserves consistency automatically by identifying affected objects and converting them when they are accessed. You can augment this automatic conversion by converting groups of objects on demand when you need to accommodate performance and availability requirements. You may also need to perform object conversion as a step in certain schema-evolution operations—typically, to set data-member values in converted objects or to release changed classes from upgrade protection.

NOTE When you delete a class from the schema, any existing instances of the class remain in the database files but cannot be accessed by applications. As a consequence, removing those objects is not an object-conversion operation, but rather, an integral part of the schema-evolution process—you should delete all objects of the class *before* you remove the class from the schema.

Chapter 5, “Schema Evolution,” in the Objectivity / C++ Data Definition Language book describes schema evolution. It explains which operations are conversion operations and which objects are affected by each such conversion. It lists the steps that you should follow if you want to delete a class from the schema.

The separately purchased product, Objectivity/C++ Active Schema, provides an alternative mechanism for performing schema evolution and object conversion. An Active Schema application can examine and modify the schema of a federated database; it can also examine and modify the persistent objects in the federated database. A single Active Schema application can perform both schema evolution and the object conversion necessary to update existing objects to the new class descriptions. For additional information, see the Objectivity/C++ Active Schema book.

Conversion to the New Shape

Most schema-evolution operations allow you to choose when and how affected objects are to be converted to their new shapes. For these operations, you can use any combination of:

- *Automatic object conversion*, in which each affected object is converted automatically the first time it is accessed by a deployed application that has been rebuilt after schema evolution.
- *On-demand conversion*, in which one or more *conversion transactions* invoke special functions to access, and therefore convert, all affected objects in particular containers, databases, or the entire federated database.

Other schema-evolution operations require you to run a special application that invokes an upgrade function to convert all affected objects. Because you must run an upgrade application before other applications access certain affected objects, this mechanism is sometimes called *immediate object conversion*.

These three different ways to convert affected objects to their new shapes are sometimes called *modes* of object conversion.

Automatic and On-Demand Object Conversion

Automatic object conversion is also known as *deferred object conversion* because the conversion of an object is deferred until the object is actually used by a rebuilt deployed application. Deferred object conversion tends to distribute the performance overhead of converting each object across many transactions. See “Automatic Object Conversion” on page 420.

In contrast, on-demand conversion explicitly requests the conversion of objects. On-demand conversion can be invoked through rebuilt deployed applications and through *conversion applications* (applications whose only purpose is to trigger object conversion). On-demand conversion tends to concentrate the performance overhead in relatively few transactions, and should be used wherever possible, especially when deployed applications have many short, read-only transactions. See “Converting Objects on Demand” on page 421.

Immediate Object Conversion

A few schema-evolution operations result in internally complex conversion processes. To ensure proper conversion, these operations require that you run the DDL processor with the `-upgrade` option in addition to the usual `-evolve` option. The steps for individual schema-evolution operations specify whether the `-upgrade` option is required (see the schema evolution chapter in the Objectivity/C++ Data Definition Language book).

The `-upgrade` option marks the changed classes (and certain related classes) as *protected* in the schema. When classes are under upgrade protection, their instances are essentially locked until you create and run a special kind of conversion application called an *upgrade application*. An upgrade application invokes a specific function that automatically converts all affected objects in the federated database and then releases the marked classes (and their instances) from upgrade protection. At this point, the affected objects can be accessed by other applications. See “Releasing Classes From Upgrade Protection” on page 434.

Conversion Mechanisms That Set Values

Some schema-evolution operations require that you set data-member values in each affected object as it is converted, usually to preserve existing data in the object’s new representation. For example, when you replace one data member with another, you can use the value of the original member to calculate a value for the new member.

You set values as part of object conversion using one of the following:

- A conversion function (for primitive data members only)
- A conversion application (for nonprimitive data members)

NOTE If the *same* primitive value is to be set for a primitive data member in *every* affected object, you can specify the value using `#pragma oodefault` in the DDL file instead of writing a conversion function. For information about this pragma directive, see the Objectivity/C++ Data Definition Language book.

Objectivity/C++ uses a conversion function or an `oodefault` pragma to set a primitive data member *during* the process of converting the object to its new shape. In contrast, values for nonprimitive data members can be set only *after* Objectivity/C++ has converted the object to its new shape. Typically, a conversion application accesses the affected objects, which causes automatic conversion to the new shape; it then sets the nonprimitive data members as required.

Conversion Function

If you need to set values for primitive data members (values of character, integer, floating point, and enumeration types), you can write a *conversion function* that uses a special interface to get original values from each preconversion shape of the object, perform any necessary calculations, and set the results in the post-conversion shape of the object. You can use a conversion function with deferred, on-demand, or immediate object conversion.

You register a conversion function in any application that is to trigger the conversion of the affected objects; the registered function is invoked automatically during the conversion of each affected object to its new shape. For details about writing and registering a conversion function, see “Setting Primitive Data Members” on page 422.

NOTE The conversion function is not registered persistently. It is used only by the application that registers it. If more than one application needs to use the same conversion function, each such application must register the function.

Conversion Application

If you need to set values for nonprimitive data members (strings, VArrays, object references, associations, or objects of embedded classes), you must build and run a special-purpose *conversion application* that iterates over every affected object, opens the object to trigger conversion to its new shape, and sets the value(s) of the new or changed data member(s). You can scan the federated database to find all instances of an evolved class; see “Scanning a Storage Object” on page 360. If you also need to set primitive data members of the affected objects, you can register the appropriate conversion functions in your conversion application.

Conversion applications are typically run as an intermediate step of a multicycle schema-evolution operation. For example, if you are replacing an obsolete nonprimitive data member with a new data member, you add the new data member in one schema-evolution cycle, build and run a conversion application to set the new data-member value based on the obsolete data-member value, and then delete the obsolete member in a second schema-evolution cycle. The requirements for conversion applications are described in the steps for individual schema-evolution operations (see the schema evolution chapter in the Objectivity / C++ Data Definition Language book).

Impact on Indexes

If schema changes affect the key fields of indexed classes, you should include code in a deployed application, a conversion application, or an upgrade application to reconstruct key description objects and recreate indexes. See “Reconstructing Indexes After Schema Evolution” on page 408.

In addition, if you change a class to make it inherit from an indexed class, existing objects of the changed class must be added to indexes on that class. See “Updating Affected Indexes” on page 439.

When Schema Changes are Distributed

You normally perform schema-evolution operations and test any required object conversion mechanisms on federated databases at your development site. When you are ready to release the evolved schema, you distribute the changes to your end-user sites, where you or your end users reproduce each schema-evolution operation in the deployed federated databases. Alternatively, you can run the `ooschemadump` tool to write the schema changes to a file; you or your end users can use the `ooschemaupgrade` tool to apply the changes to the deployed federated databases. See the Objectivity/DB administration book for a description of these tools.

If conversion or upgrade applications are required by any schema-evolution operation, these applications must be included in the distribution package. See the schema evolution chapter in the Objectivity/C++ Data Definition Language book for information about distributing schema changes.

Object Conversion and Schema-Evolution History

When you perform multiple schema-evolution operations on a class, the schema preserves all of the class descriptions that correspond to distinct shapes for objects of the class. Objectivity/DB uses this schema-evolution history to construct a program for converting each object of the class to the latest shape. Occasionally it is appropriate to purge the history, which you can do during on-demand or immediate object conversion. To help you decide whether to purge history when you convert objects, see “Purging Schema-Evolution History” on page 439.

Summary of Object-Conversion Mechanisms

Table 19-1 lists the mechanisms for converting affected objects.

Table 19-1: Object-Conversion Mechanisms

Conversion Triggered By	Mode	Can Set Values Of	Granularity	DDL Processor Option(s)
Deployed application that accesses individual affected objects during normal operations (see page 420)	Automatic (Deferred)	Primitive-typed data members, if you register a **conversion function**	Persistent object	`-evolve`
Conversion transaction in one or more deployed or conversion applications; invokes the `convertObjects` function for a storage object (see page 421)	On-demand	Primitive-typed data members, if you register a **conversion function**	Container, database, or federated database	`-evolve`
Conversion application that explicitly iterates over every affected object; uses the ordinary Objectivity/C++ interface for iteration and data-member access	(An ordinary program, not a conversion mode)	Data members of any type, accessed in the usual way. Primitive-typed data members, if you register a **conversion function**	Federated database	`-evolve`
Upgrade application, which invokes `upgradeObjects` for the federated database; releases classes from upgrade protection (see page 434)	Immediate	Primitive-typed data members, if you register a **conversion function**	Federated database	`-evolve` `-upgrade`

Note that an application created specifically for triggering the conversion of affected objects can be written to:

- Execute conversion transactions that perform on-demand conversion.
- Iterate over affected objects, open them (causing automatic conversion), and set data-member values.
- Force immediate conversion of all affected objects and release certain evolved classes from upgrade protection (known as an upgrade application).

Automatic Object Conversion

The simplest way to convert affected objects after schema evolution is to run the updated deployed applications. Each affected object is automatically converted to its evolved representation the first time it is accessed by an application. Thus, the conversion of each affected object is *deferred* until the object is actually used.

In deferred object conversion, a converted object is saved persistently only if it is accessed in an update transaction. If an affected object is accessed in a read-only transaction, the transaction will convert the object to read it, but will not save the conversion. The object must be converted again the next time it is accessed.

Converting objects one at a time allows continued access to a federated database. You should consider deferred object conversion if either of the following is true:

- The deployed environment cannot afford the downtime or reduced access that may result from converting all affected objects at one time.
- The federated database contains a large number of affected objects, but only a small number of these objects will ever be accessed.

Deferred object conversion cannot be used following schema modifications that require an upgrade application; see “Releasing Classes From Upgrade Protection” on page 434.

WARNING Do not restart a deployed application unless it has been rebuilt with the new header and implementation files generated by the DDL processor during schema evolution. When an unrebuilt application accesses an affected object, the evolved shape will be inconsistent with the shape expected by the application. At best, the data read from the object may be misinterpreted by the application; at worst, the misinterpreted values may be written to the database and committed, with no error signaled.

You can allow multiple applications to trigger deferred object conversion in the same federated database. If, however, you evolved a class to change a data member from one primitive type to another, you should build these applications on the same architecture (compiler and platform) to ensure that values of the changed data members are converted consistently. Objectivity / C++ does not resolve any differences in conversion semantics across architectures.

Converting Objects on Demand

You can write a *conversion transaction* that converts all affected objects in a container, a database, or a federated database *on demand*. A conversion transaction is an update transaction in which you call the `convertObjects` member function on a handle to the storage object containing the objects to be converted (for details, see "Writing a Conversion Transaction" on page 422). You can modify one or more deployed applications to execute a conversion transaction or you can create a special application for this purpose.

Because on-demand conversion occurs in an update transaction, all converted objects are saved persistently when the transaction commits. In contrast, deferred conversions that occur in a read-only transaction last only for the duration of the transaction.

Converting objects on demand helps you control the performance impact of conversion:

- You can concentrate the performance impact by converting all affected objects in a federated database at once. If the deployment site can afford downtime, you can perform the conversion before restarting deployed applications.
- You can distribute the performance impact by converting some affected objects through deferred object conversion, then converting the affected objects in particular containers or databases on demand, until it is convenient to finish converting the entire federated database.

You should consider on-demand conversion if you want to reduce or eliminate the performance impact of conversion during deployed applications. This is especially important when deployed applications have many short, read-only transactions that repeatedly access the same affected objects. Such transactions experience the performance overhead for an object each time it is opened.

On-demand conversion can be performed instead of, or in combination with, automatic conversion. In all cases, the applications triggering conversion must be built (or rebuilt) with the header and implementation files generated by the DDL processor during schema evolution (see the warning on page 420). If you evolved a class to change a data member from one primitive type to another, all applications triggering conversion should be built on the same architecture (compiler and platform) to ensure that values of the changed data members are converted consistently.

On-demand object conversion cannot be used following schema modifications that require an upgrade application; see "Releasing Classes From Upgrade Protection" on page 434.

Writing a Conversion Transaction

To convert all affected objects in the federated database or in a particular container or database:

1. Start an update transaction.
2. Obtain a handle to the federated database, database, or container.
3. Call the `convertObjects` member function on the handle.
4. Commit the transaction.

Setting Primitive Data Members

Some schema-evolution operations require that you set data-member values in each affected object as it is converted, usually to preserve existing data in the object's new representation. For example, when you replace one data member with another, you can use the value of the original member to calculate a value for the new member.

This section describes how to set data members of primitive types (character, integer, floating point, and enumeration types) in the affected objects of a changed class. To do so, you define a *conversion function* for the class. You then register the conversion function in one or more applications that trigger object conversion; the registered function is invoked automatically each time an affected object of the class is converted. A given conversion function applies to the affected objects of only one changed class; if you want to set primitive values in the affected objects of more than one changed class, you must write a separate conversion function for each class.

NOTE If the data members to be set are of nonprimitive types (strings, VArrays, object references, associations, and objects of embedded classes), you must build and run a conversion application that explicitly iterates over all objects of the changed class, opens each object to trigger conversion to the new shape, and sets the relevant data members appropriately.

Accessing Primitive Data Members

During object conversion, you access the data of the persistent object being converted through two kinds of transient objects:

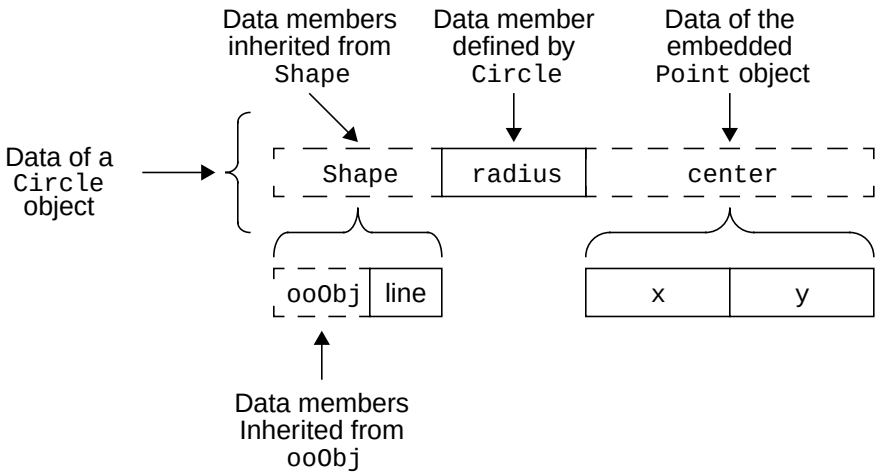
- An instance of the class `ooConvertInObject`, called an *unconverted object*, is your interface to the persistent object *before* it has been converted. It provides read-only access to the primitive data members in the *old shape* of the persistent object, allowing you to get the existing values for those data members.
- An instance of the class `ooConvertInOutObject`, called a *converted object*, is your interface to the persistent object *after* it has been converted. It provides access to the primitive data members in the *new shape* of the persistent object, allowing you to set new values for those data members.

When you need to get values for a data member of the old shape, you use a different unconverted object depending on whether that data member is defined by the class, inherited, or in the data of an embedded class.

EXAMPLE A geometry application includes the following classes before schema evolution.

```
// DDL file geometry.ddl
class Point {
public:
   int32 x, y;     // Rectangular coordinates
   …
};
class Shape : public ooObj {
public:
   int16 line;     // Width of border when drawn
   …
};
class Circle : public Shape {
public:
   float32 radius; // Radius in inches
   Point center;   // Location of center
   …
};
```

The data for an instance of the `Circle` class is illustrated in the following figure.



A conversion function for the `Circle` class would need three different unconverted objects to access all public preconversion data members of a `Circle` object:

- An unconverted object for the `Circle` object itself can access the `radius` data member, which is defined by the `Circle` class.
- An unconverted object for the `Shape` base class can access the `line` data member, which is inherited from the `Shape` class.
- An unconverted object for the `Point` class can access the `x` and `y` data members of the embedded `Point` object in the `center` attribute of the `Circle` object.

A conversion function can call member functions of the unconverted object for the `Circle` class to obtain the unconverted objects for the `Shape` base class and for the embedded `Point` class.

Similarly, when you need to set values for a data member of the new shape, you use a different converted object depending on whether that data member is defined by the class, inherited, or in the data of an embedded class.

EXAMPLE Later in the development of the geometry application, decisions are made to use metric measurements, to record the area of each shape object, and to use a polar coordinate system instead of the rectangular coordinate system that was implemented initially. A developer modifies classes accordingly:

- The `Shape` class is given a new `area` attribute to store the area in square centimeters.

- The x and y data members of the Point class are replaced with rho and theta data members.

Although the attributes defined in the Circle class are not changed, the meaning of the radius attribute is updated for consistency with the new inherited area attribute. Instead of storing a circle's radius in inches, the radius attribute now stores the radius in centimeters.

The new definitions of the classes are as follows:

```
// DDL file geometry.ddl
class Point {
public:
   int32 rho, theta;  // Polar coordinates
   ...
};
class Shape : public ooObj {
public:
   int16 line;        // Width of border when drawn
   float64 area;      // Area
   ...
};
class Circle : public Shape {
public:
   float32 radius; // Radius in centimeters
   Point center;   // Location of center
   ...
};
```

A conversion function for the Circle class would need three different converted objects to access all public post-conversion data members of a Circle object:

- A converted object for the Circle object itself can access the radius data member, which is defined by class Circle.
- A converted object for the Shape base class can access the data members line and area, which are inherited from the Shape class.
- A converted object for the Point class can access the rho and theta data members of the embedded Point object in the center attribute of the Circle object.

A conversion function can call member functions of the converted object for the Circle class to obtain the converted objects for the Shape base class and for the embedded Point class.

Defining a Conversion Function

An conversion function is an application-defined function that must conform to the calling interface defined by the `ooConvertFunction` function pointer type.

To define a conversion function:

1. Create a `void` function that takes two parameters:
   - A `const` C++ reference to an `ooConvertInObject` object. This object is the unconverted object for the persistent object being converted.
   - A C++ reference to an `ooConvertInOutObject` object. This object is the converted object for the persistent object being converted.
2. Call member functions on the unconverted object to get the original values of one or more data members; see "Getting Data-Member Values" on page 426.
3. Perform any desired computation on the original data-member values. For example, you may want to convert those values to a different type or combine them to produce new value(s).
4. Call member functions on the converted object to set the desired data-member value(s); see "Setting Data-Member Values" on page 429.

NOTE A conversion function should access *only* the object being converted; it may not access other persistent objects.

Getting Data-Member Values

Within a conversion function, you call member functions of an unconverted object (an instance of `ooConvertInObject`) to get the original values for data members—that is, the values *before* the persistent object has been converted.

Getting a Primitive Data Member

To get the original value of a primitive data member, you call the appropriate `get`*`Type`* member function of the unconverted object, where *`Type`* indicates the data member's type. For example, you call the `getInt16` member function for a data member of type `int16`.

NOTE The `get`*`Type`* member functions correspond to the Objectivity / DB primitive types that are used in class descriptions in the schema, *not* to Objectivity / C++ types or C++ types that are used in class declarations in DDL files.

If the declared type of the data member is not also an Objectivity / DB primitive type, you call get the `get`*`Type`* function for the corresponding Objectivity / DB type. For example, if the declared type is `int` or `d_Long`, you call `getInt32`. See "Objectivity / DB Primitive Types" on page 188 in the Objectivity / C++ Data Definition Language book. If the declared type has platform-dependent mappings to different Objectivity / DB primitive types, you can use the `ooschemadump` tool to find out the data member's actual type in the schema.

The `get`*`Type`* member functions take as parameters the name of a data member and a C++ reference to the variable in which to return the original value of the specified data member.

EXAMPLE This fragment of a conversion function for the `Circle` class gets the current value of a `Circle` object's `radius` attribute by calling the `getFloat32` member function on the unconverted `Circle` object.

```
// Application code
#include "geometry.h"
...
void ConvertCircle(
         const ooConvertInObject &old,   // Unconverted Circle
         ooConvertInOutObject &new) {    // Converted Circle
   ...
   // Get the original value of the radius attribute
   float32 radiusInInches;
   old.getFloat32("radius", radiusInInches);
   ...
}
```

Getting an Inherited Attribute

If a persistence-capable class inherits data members from a base class, the data corresponding to those inherited data members is embedded in the data for a persistent object of the derived class, as if it were an embedded object of the base class; see the figure on page 423. You access the inherited data members through an unconverted object corresponding to the data for the base class.

To get the original value of a primitive data member of class `A` that is inherited from base class `B`:

1. Call the `getOldBaseClass` data member on the unconverted object for `A`, passing as parameters the name of the base class and another instance of `ooConvertInObject`. This member function sets its second parameter to an unconverted object for the specified base class.

2. Call the appropriate `get`*Type* member function on the unconverted object you obtained in step 1 to get the original value of the inherited data member.

EXAMPLE To get the value of the inherited `line` attribute of a `Circle` object, this fragment of a conversion function for the `Circle` class first gets an unconverted object for the `Shape` base class, then calls that unconverted object's `getInt16` member function.

```
// Application code
#include "geometry.h"
...
void ConvertCircle(
          const ooConvertInObject &old,   // Unconverted Circle
          ooConvertInOutObject &new) {    // Converted Circle
   ...
   // Get an unconverted object for the Shape base class
   ooConvertInObject oldShape;           // Unconverted Shape
   old.getOldBaseClass("Shape", oldShape);

   // Get the original value of the line attribute
   int16 oldLineWidth;
   oldShape.getInt16("line", oldLineWidth);
   ...
}
```

Getting a Primitive Data Member in an Embedded Object

If a persistence-capable class `A` has an embedded-class attribute of class `B`, an object of class `B` is embedded in the data for a persistent object of class `A`; see the figure on page 423. You can access the primitive data members of the embedded object through an unconverted object corresponding to the embedded object.

To get the original value of a primitive data member in an object of class `B` embedded within a persistent object of class `A`:

1. Call the `getOldDataMember` member function on the unconverted object for `A`, passing as parameters the name of the embedded-class attribute and another instance of `ooConvertInObject`. This member function sets its second parameter to an unconverted object for the embedded object in the specified attribute.
2. Call the appropriate `get`*Type* member function on the unconverted object you obtained in step 1 to get the original value of the data member in the embedded object.

EXAMPLE To get the values of the `x` and `y` attributes of the `Point` object embedded in the `center` attribute of a `Circle` object, this fragment of a conversion function for the `Circle` class first gets an unconverted object for the embedded `Point` object, then calls that unconverted object's `getInt32` member function for each of the data members.

```
// Application code
#include "geometry.h"
...
void ConvertCircle(
         const ooConvertInObject &old,   // unconverted Circle
         ooConvertInOutObject &new) {    // converted Circle
   ...
   // Get an unconverted object for the embedded Point
   ooConvertInObject oldPoint;           // unconverted Point
   old.getOldDataMember("center", oldPoint);

   // Get the original rectangualr coordinates
   int32 xCoord, yCoord;
   oldPoint.getInt32("x", xCoord);
   oldPoint.getInt32("y", yCoord);
   ...
}
```

Setting Data-Member Values

Within a conversion function, you call member functions of a converted object (an instance of `ooConvertInOutObject`) to set the new values for data members—that is, the values *after* the object has been converted.

Setting Primitive Data-Member Values

To set the new value of a primitive data member, you call the appropriate `set`*Type* member function of the converted object, where *Type* indicates the data member's type. For example, you call the `setFloat32` member function for a data member of type `setFloat32`.

NOTE The `set`*Type* member functions correspond to the Objectivity / DB primitive types that are used in class descriptions in the schema, *not* to Objectivity / C++ types or C++ types that are used in class declarations in DDL files.

If the declared type of the data member is not also an Objectivity / DB primitive type, you call the `set`*`Type`* function for the corresponding Objectivity / DB type. For example, if the declared type is `double` or `ooFloat64`, you call `setFloat64`. See "Objectivity / DB Primitive Types" in Appendix D of the Objectivity / C++ Data Definition Language book. If the declared type has platform-dependent mappings to different Objectivity / DB primitive types, you can use the `ooschemadump` tool to find out the data member's actual type in the schema.

The `set`*`Type`* member functions take as parameters the name of a data member and the new value for the specified data member.

EXAMPLE This fragment of a conversion function for the `Circle` class gets the old radius (in inches) and converts it to centimeters. The function `convertToCm` (not shown) takes a number of inches and returns the equivalent number of centimeters.

To set the new value for a `Circle` object's `radius` attribute, the conversion function calls the `setFloat32` member function on the converted `Circle` object.

```
// Application code
#include "geometry.h"
...
void ConvertCircle(
         const ooConvertInObject &old,   // Unconverted Circle
         ooConvertInOutObject &new) {    // Converted Circle
   ...
   // Get the original value of the radius attribute
   float32 radiusInInches;
   old.getFloat32("radius", radiusInInches);

   // Convert the original value to centimeters
   float32 radiusInCm = convertToCm(radiusInInches);

   // Set the new value of the radius attribute
   new.setFloat32("radius", radiusInCm);
   ...
}
```

Setting an Inherited Data Member

To set the new value of a primitive data member of class A that is inherited from base class B:

1. Call the `getNewBaseClass` data member on the converted object for A, passing as parameters the name of the base class (`"B"`) and another instance of `ooConvertInOutObject`. This member function sets its second parameter to a converted object for the specified base class.
2. Call the appropriate `set`*Type* member function on the converted object you obtained in step 1 to set the new value of the inherited data member.

EXAMPLE This fragment of a conversion function for the `Circle` class calculates the area from the radius in centimeters. To set the value of the inherited `area` attribute of a `Circle` object, the conversion function first gets a converted object for the `Shape` base class, then calls that converted object's `setFloat64` member function.

```
// Application code
#include "geometry.h"
...
void ConvertCircle(
         const ooConvertInObject &old,   // Unconverted Circle
         ooConvertInOutObject &new) {    // Converted Circle
   ...
   // Calculate the area of the circle
   float64 newArea = radiusInCm * radiusInCm * 3.14159265359;

   // Get a converted object for the Shape base class
   ooConvertInOutObject newShape;        // Converted Shape
   new.getNewBaseClass("Shape", newShape);

   // Set the value for the new area attribute
   newShape.setFloat64("area", newArea);
   ...
}
```

Setting a Primitive Data Member in an Embedded Object

To set the new value of a primitive data member in an embedded object of class B within a persistent object of class A:

1. Call the `getNewDataMember` member function on the converted object for A, passing as parameters the name of the embedded-class attribute and another instance of `ooConvertInOutObject`. This member function sets its second parameter to a converted object for the embedded object in the specified attribute.
2. Call the appropriate `set`*Type* member function on the converted object you obtained in step 1 to set the new value of the data member in the embedded object.

EXAMPLE This fragment of a conversion function for the `Circle` class converts the existing rectangular coordinates to polar coordinates. The function `convertRectToPolar` (not shown) takes rectangular coordinates as its first two parameters and sets its third and fourth parameters to the equivalent polar coordinates.

To set the values of the new attributes of the `Point` object embedded in the `center` attribute of a `Circle` object, the conversion function first gets a converted object for the embedded `Point` object, then calls that converted object's `setInt32` member function for each of the data members.

```
// Application code
#include "geometry.h"
...
void ConvertCircle(
         const ooConvertInObject &old,   // Unconverted Circle
         ooConvertInOutObject &new) {    // Converted Circle
   ...
   // Get a converted object for the embedded Point
   ooConvertInOutObject newPoint;        // Converted Point
   new.getNewDataMember("center", newPoint);

   // Calculate the polar coordinates
   int32 rhoCoord, thetaCoord;
   convertRectToPolar(xCoord, yCoord, rhoCoord, thetaCoord);

   // Set the values for the Point's rho and theta data members
   newPoint.setInt32("rho", rhoCoord);
   newPoint.setInt32("theta", thetaCoord);
   ...
}
```

Registering a Conversion Function

You make a conversion function available to Objectivity / DB by *registering* it for the evolved class to which it applies. You should register a conversion function in every application that will access (and therefore convert) objects of this class. When a conversion function is registered, it is executed automatically during the conversion of each affected object of the specified class.

You can register a conversion function in a deployed application, in a special-purpose conversion application, or in an upgrade application (if you are also making schema changes that require an upgrade application). If you are converting objects as an intermediate step within a schema-evolution operation, or if you intend to perform a subsequent schema-evolution operation on the same class, you must arrange for the conversion function to be invoked for *every* object of the changed class—for example, by registering it in an application that converts all objects on demand. This is the only way to guarantee consistent results, especially if a subsequent schema change introduces a conversion function of its own.

WARNING Do not rely on deferred conversion if you are using a conversion function to preserve the values of deleted data members (for example, by setting these values in new data members). Deferred conversion allows some objects to remain unconverted, so values are not set for these objects. If these objects are then converted after a subsequent schema change is made to the same class, the unset values may be lost.

If an application is to convert the affected objects of multiple changed classes, you register a separate conversion function for each class. In a given application, however, you can register at most one conversion function per class. If you register more than one conversion function for a class, only the last one will be used.

To register a conversion function:

1. Start an update transaction.
2. Call the `setConversion` member function on a handle to the federated database, passing as parameters the name of the class of objects to be converted and a function pointer to the conversion function.

EXAMPLE This example shows how to register the function `ConvertCircle` as the conversion function for the `Circle` class.

```
// Application code
#include "geometry.h"
...
ooTrans trans;
ooStatus status;
ooHandle(ooFDObj) fdH;

// Start an update transaction
trans.start();
if (!fdH.open("myFD", oocUpdate)) {
   cerr << "Cannot open federated database for update" << endl;
   trans.abort();
}

// Register the conversion function for the Circle class.
if (!fdH.setConversion ("Circle", ConvertCircle)) {
   cerr << "Cannot register conversion function" << endl;
   trans.abort();
}
...
trans.commit();
```

After the conversion function has been registered, it will be used to convert objects of the specified class either in the same transaction, or in subsequent transactions in the same process.

Releasing Classes From Upgrade Protection

Certain schema-evolution operations require that you create a special-purpose *upgrade application* to perform object conversion. These operations are:

- Deleting a persistence-capable class with a base class that is the destination class for some associations or reference attributes.
- Moving a persistence-capable class to a higher or lower level in its inheritance graph.

When you perform each of these operations, you must run the DDL processor with both the `-evolve` and `-upgrade` options. The DDL processor marks the changed classes as *protected* in the schema. When classes are under upgrade

protection, their instances are essentially locked until conversion is complete. You run an upgrade application to convert all affected objects in the federated database and release the marked classes (and their instances) from upgrade protection. At this point, the affected objects can be accessed by other applications.

Because an upgrade application converts all affected objects up front, its use is sometimes called *immediate object conversion*. When an upgrade application is required, it takes the place of any other mechanism for triggering conversion of the affected objects to their new shapes.

An error is signaled when a nonupgrade application tries to access an object of a protected class before an upgrade application has run. You can trap this error in other applications to notify users that they need to run the upgrade application on their federated database.

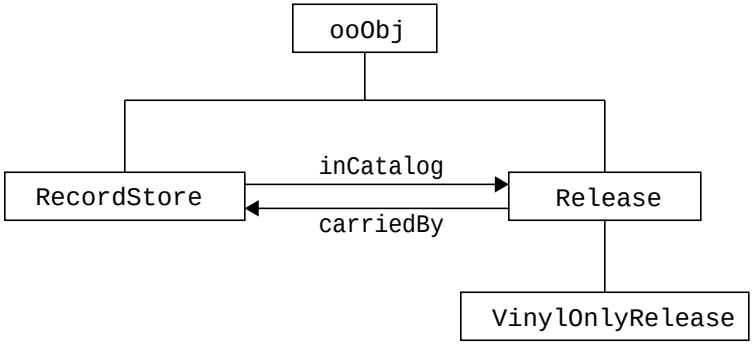
Writing an Upgrade Application

An upgrade application is a single-use application that calls special upgrade-interface member functions. To write an upgrade application:

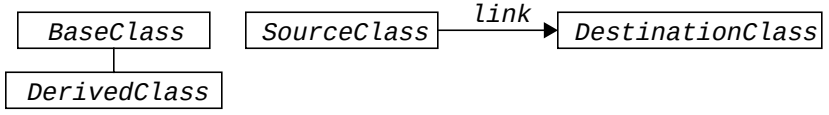
1. Initialize Objectivity / DB and create a transaction object, as in any Objectivity / C++ application.
2. Call the `upgrade` member function on the transaction object to identify this application as an upgrade application. You must call this member function before starting the first (and only) transaction in an upgrade application.
3. Start an update transaction, opening the federated database to be upgraded.
4. If you need to initialize data members for any affected objects, register the desired conversion functions (see “Setting Primitive Data Members” on page 422).
5. Call the `upgradeObjects` member function on a handle to the federated database to perform the necessary upgrade. This function iterates through all objects in the federated database, causing any affected objects to be converted. It also releases any protected objects.
6. Commit the transaction.

EXAMPLE A retail record store carries a number of released recordings for sale. Most releases are available as compact disks (CDs), as vinyl records, or as both CDs and vinyl records.

An application that manages inventory for a chain of record stores uses the following object model:



Key to Symbols



When all releases are available as CDs, the `VinylOnlyRelease` class is deleted.

The application's DDL file is modified as follows. Lines added to the file are shown in boldface; lines deleted from the original file are struck through.

```
// Modified DDL file music.ddl
class RecordStore;
class Release;
class VinylOnlyRelease;

class RecordStore: public ooObj {
private:
   ooVString _name;
   ooBoolean _carryVinyl;
public:
   ooRef(Release) inCatalog[] <-> carriedBy[];
   RecordStore(const char *str, ooBoolean carryVinyl);
   const char *name() const { return (const char *)_name; }
};
```

```
class Release: public ooObj {
private:
   ooVString _title;
   ooVString _artistName;
protected:
   uint32 _CDSales;
   uint32 _vinylSales;
public:
   ooRef(RecordStore) carriedBy[] <-> inCatalog[];
   Release() {;}
   Release(const char *title, const char *artistName);
   uint32 CDSales() const { return _CDSales; }
   uint32 vinylSales() const { return _vinylSales; }
   virtual void set_CDSales(uint32 n) { _CDSales = n; }
   void set_vinylSales(uint32 n) { _vinylSales = n; }
};
// Deleting VinylOnlyRelease class
#pragma oodelete VinylOnlyRelease

class VinylOnlyRelease: public Release {
public:
   VinylOnlyRelease();
   VinylOnlyRelease(const char *title, const char *artistName)
      : Release(title, artistName) { _CDSales = 0; }
   void set_CDSales(uint32 n) {;}
};
```

The DDL processor makes the changes to the federated database schema when it is run with the `-evolve` and `-upgrade` options to process the modified DDL file. The `-upgrade` option is required because the deleted class (`VinylOnlyRelease`) has a base class (`Release`) that is the destination class for an association (`inCatalog`). The upgrade application marks `VinylOnlyRelease` class as protected.

An upgrade application must be run before any application can access objects of the protected classes. The upgrade application deletes any existing objects of class `VinylOnlyRelease`, and ensures that no dangling links exist from `RecordStore` objects to deleted `VinylOnlyRelease` objects. If any application tries to access a `RecordStore` or `Release` object before the upgrade application has finished running, an error is signaled.

The source code for the upgrade application follows.

```
// Application code file
// Upgrade application for musical recording application
#include "music.h"

// Upgrade objects in the federated database
int runUpgrade() {
   ooTrans trans;
   ooHandle(ooFDObj) fdH;

   trans.upgrade();         // Make this an upgrade application

   // Start an update transaction
   trans.start();
   if (!fdH.open("myFD", oocUpdate)) {
      cerr << "Cannot update federated database" << endl;
      trans.abort();
      return 1;
   }

   // Perform the upgrade
   if (fdH.upgradeObjects()) {
      trans.commit();
      return 0;
   }
   else {
      cerr << "Cannot upgrade objects" << endl;
      trans.abort();
      return 1;
   }
} // End runUpgrade

int main() {
   int retval = 0;
   if (ooInit()) {
      retval = runUpgrade();  // Call function to upgrade FD
   }
   else {
      cerr << "Unable to initialize Objectivity/DB" << endl;
      retval = 1;
   }
   return retval;
} // End main
```

Updating Affected Indexes

Some schema changes require that indexes be updated. Typically, any transaction that converts the indexed objects also updates the indexes that contain them. By default, indexes are updated automatically when an update transaction commits, as described in “Updating Indexes” on page 402.

If you change the base classes of a particular class so that it inherits from an indexed class, existing objects of the changed class must be added to the index. Any transaction that converts an object of the changed class also updates the relevant indexes to contain that object. However, any unconverted objects of the changed class are not added to the indexes. For this reason, you may want to create an application that converts *all* objects of the changed class on demand; doing so forces the indexes to be updated to contain all objects of the class.

If you delete a derived class of an indexed class, you must remove all existing objects of the deleted class from the index. This step is performed automatically when you follow the procedure described in “Deleting a Class” on page 126 in the Objectivity/C++ Data Definition Language book. You remove objects of the class *before* you remove the class from the schema; the transaction in which you delete the objects also updates the indexes that contain them.

If schema changes affect the key fields of indexed classes, you must reconstruct key description objects and recreate indexes. See “Reconstructing Indexes After Schema Evolution” on page 408.

Purging Schema-Evolution History

Schema-evolution history is a record of the previous shapes for evolved classes. This history is used during object conversion; it enables deferred conversion to work, even for affected objects that are not accessed until after the class has been evolved several times (through several shapes).

If accumulated schema-evolution history causes the system-database file to occupy too much disk space, you can consider *purging* the history from the schema. You can purge schema-evolution history during on-demand conversion of an entire federated database or during immediate conversion (in an upgrade application):

- During on-demand conversion, call the `convertObjects` member function on a handle to the federated database, passing `oocTrue` as the `purge_schema` parameter.
- In an upgrade application, call the `upgradeObjects` member function on a handle to the federated database, passing `oocTrue` as the `purge_schema` parameter.

You must be careful if you make schema changes in a development federated database and then distribute them to deployed federated databases. In that scenario, you can safely purge schema-evolution history from the development federated database *only after* you have distributed *all* schema changes and then converted *all* affected objects in *all* of the deployed federated databases.

WARNING If you use the `ooschemadump` tool to write a purged schema to an output file, the resulting file will not be accepted by any deployed federated database that still contains unconverted objects from earlier schema-evolution operations (that is, objects whose shapes have been purged from the schema you are distributing). To update the schema of such a deployed federated database, your only recourse is to restore the development federated database to an earlier state, perform the schema-evolution operation again, and distribute the evolved schema without purging its history.

20

Versioning Basic Objects

The Objectivity / DB *versioning* capability enables you to track multiple copies (or *versions*) of a basic object whose state needs to be remembered each time it changes.

This chapter describes:

- General information about versioning
- Enabling and disabling versioning
- Creating versions and genealogies
- Merging version branches
- Finding versions
- Deleting versions
- Customizing version creation

Understanding Versions

While nearly all applications involve changing various basic objects, some applications also need to capture the transitions in certain objects. The standard way to capture an object transition is to make each set of changes in a copy of the object, preserving the original. The copy is called a *version* of the object.

For example, a book publisher might want to track various versions of a manuscript. Suppose the first version of a manuscript is drafted by a writer named Sam, but the publisher is not satisfied with it, so Sam drafts a second version. Sam's second version is edited by Jean, producing a third version, as shown in Figure 20-1.

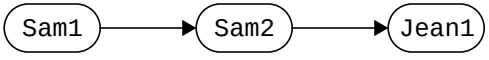


Figure 20-1 Three Versions of a Manuscript

NOTE Only basic objects can be versioned—that is, objects of class `ooObj` and of classes derived from `ooObj` (but not through `ooContObj`). Storage objects, such as databases and containers, cannot be versioned.

Next and Previous Versions

At the semantic level, a version differs from a plain copy in that a version maintains ordering information. In the example, Sam's second draft is later than his first draft, and that chronological relationship needs to be captured.

In Objectivity / C++, the next / previous information is represented by a pair of inverse bidirectional associations, `nextVers` and `prevVers`. These associations are defined in `ooObj` and inherited by all basic objects. For example, when two copies of a manuscript (`Sam1` and `Sam2`) are related as versions, `Sam2` is set as the destination object of `Sam1`'s `nextVers` association, and `Sam1` is set as the destination of `Sam2`'s `prevVers` association.

Figure 20-2 shows the associations that establish three versions of a manuscript.

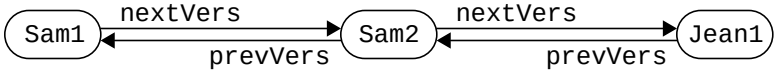


Figure 20-2 Associations that Represent Versions

Linear Versioning and Branch Versioning

A series of versions is said to be *linear* when they proceed in a linear evolution from one to the next to the next. *Branch versioning* occurs when an object has two or more versions, because the evolutionary sequence branches at that juncture.

In the `Sam1-Sam2-Jean1` example, each manuscript is a linear version of its predecessor. Suppose the publisher commissioned a second writer, named Tess, to rewrite Sam's first draft—at the same time as Sam was rewriting it. The `Tess1` version is a branch version because it begins a new branch in the evolutionary tree. Figure 20-3 illustrates the `Tess1` branch.

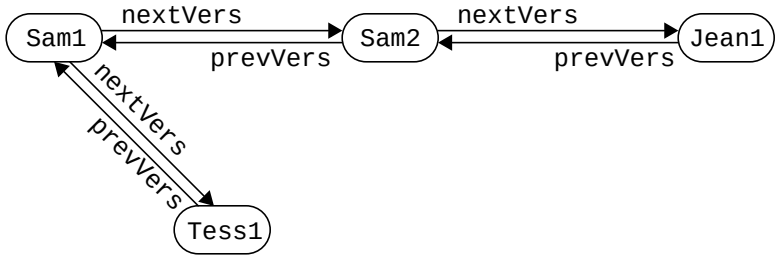


Figure 20-3 Linear and Branch Versioning

In Objectivity / C++, either linear or branch versioning can be enabled for any basic object. When linear versioning is enabled, a given object can have at most one next version (that is, it can be linked to at most one destination object by the `nextVers` association). When branch versioning is enabled, an object can have multiple next versions (that is, it can be linked to multiple destination objects by the `nextVers` association). Thus, the distinction between linear and branch versioning is implicit in the associations that are set among versions, rather than being enforced by the object model.

Genealogies and Default Versions

When you create various versions of a particular entity, it is often convenient to have a single object that represents the versioned entity itself. An Objectivity / C++ *genealogy* serves this purpose. A genealogy is an instance of the persistence-capable class `ooGeneObj` or an application-defined class derived from `ooGeneObj`. Because a genealogy is derived from class `ooObj`, it is a specialized kind of basic object.

Among the versions in a genealogy, one particular version is the *default version*. Typically, other objects that access the versioned entity (without creating new versions of it) use the default version. Some applications may successively set each new version to be the default; other applications may use an older default version until a later version has been developed, tested, approved, and finally appointed as the new default.

A genealogy's default version must be set explicitly; a genealogy doesn't contain any objects until its first default version is set. Thereafter, that version and all subsequently created versions are automatically added to the genealogy.

A genealogy maintains bidirectional associations to keep track of its current default and the other versions it manages:

- The genealogy's `defaultVers` association links it to its default version; the inverse association `defaultToGeneObj` links the default version to its genealogy.

- The genealogy's `allVers` association links it to all the objects it contains; the inverse `geneObj` links each version to its genealogy.

Figure 20-4 shows associations in a genealogy containing the versions of Sam's book that are shown in Figure 20-3. The genealogy `G` is an instance of `ooGeneObj`. The second draft (`Sam2`) is set as the current default version while Jean and Tess are finishing their edits.

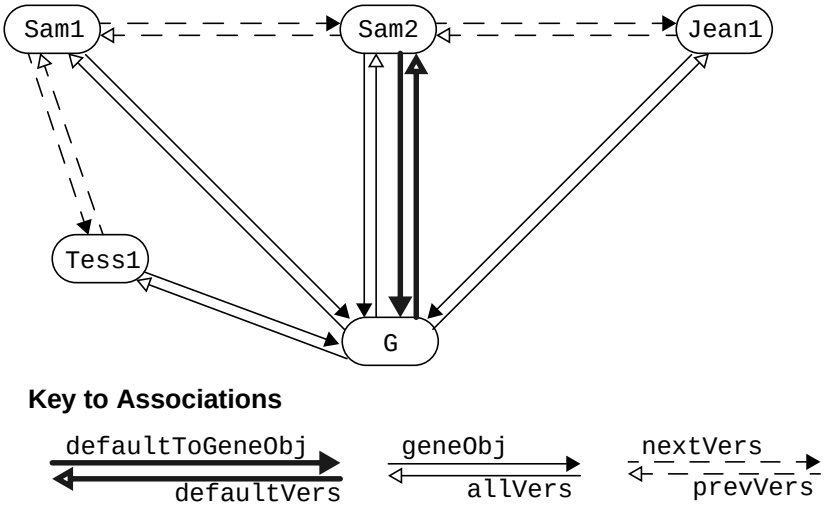


Figure 20-4 Associations that Support Default Versions

A genealogy makes it convenient to find the default version or other versions. For example, you can find all versions of an object by iterating over the destination objects of the genealogy's `allVers` association. This technique of finding versions is more convenient than the alternative, which is to recursively request the next version(s) from each previous one.

Derivative and Secondary Ancestor Versions

Sometimes two or more branches of a genealogy merge, so that a version has multiple ancestors. Because it is often useful to distinguish between a primary ancestor—the previous version—and secondary ancestors, an object is said to be *derived from* its secondary ancestors.

In Objectivity / C++, derivation information is represented as a pair of inverse bidirectional associations, `derivedFrom` and `derivatives`. These associations are defined in `ooObj` and inherited by all basic objects. You can find a version's secondary ancestors by traversing its `derivedFrom` association; similarly, you can find a secondary ancestor's derivatives by traversing its `derivatives` association.

For example, suppose the publisher in the manuscript example commissioned an editor named Pat to take the best parts from both Jean's and Tess's drafts, and combine them in a merged manuscript. Pat started with Jean's draft, added text from Tess's draft, and removed redundant text. The `Pat1` version has `Jean1` as its previous version, and is derived from `Tess1`. Looking at the inverse associations, `Jean1` has `Pat1` as a next version, and `Tess1` has `Pat1` as a derivative, as shown in Figure 20-5.

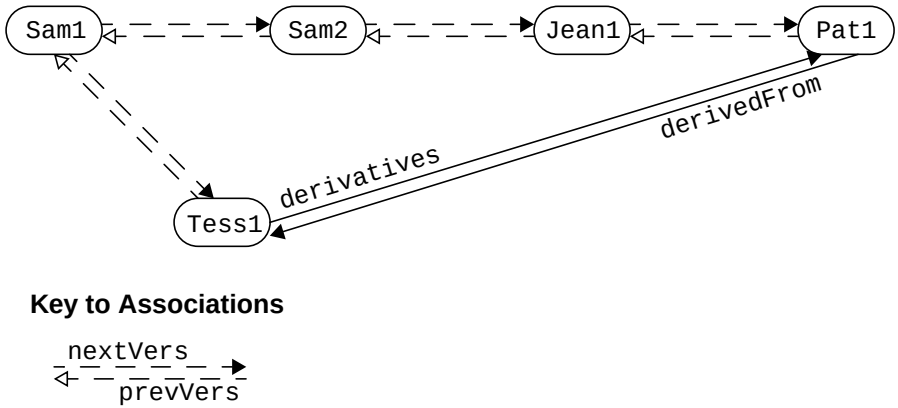


Figure 20-5 Merging Branches

Version Naming

Objectivity/C++ does not support version names or identifiers. If an application needs to be able to find particular versions of a given object, it must provide a way to identify and look up individual versions. For example, an application could use scope names, name maps, or object attributes to identify individual versions (see Chapter 16, "Individual Lookup of Persistent Objects").

Versions as Copies of Basic Objects

When you create a version of a particular basic object, you create a copy of that object (see "Copying a Basic Object" on page 197). As with copying:

- The new version is an instance of the original object's class and has its own object identifier.
- The new version has attributes that are set to *bit-wise* copies of the corresponding attributes of the original object.
- If the original object has reference attributes that link it to one or more destination objects, the new version is linked to the same destination objects through corresponding attributes.

- If the original object has any associations that link it to one or more destination objects, the new version either preserves or deletes such links, depending on the versioning behavior specified in each association definition. A given association's links may be retained by the original object only, transferred to the new version, or duplicated so that both the original and the new version are associated with the same destination object(s). See "Copying and Versioning Behavior" on page 150.

You can arrange for postprocessing to fix any attribute values for which bit-wise copying is inappropriate, or to propagate versioning to any linked destination object; see "Customizing the Created Version" on page 465.

Versioning Interface

The Objectivity/C++ versioning interface consists of:

- Member functions defined by *ooRefHandle*(ooObj) for enabling and disabling linear and branch versioning for a basic object, setting an object to be the default version of a genealogy, and finding an object's next, previous, and default versions. When you call these member functions on a handle to a basic object, Objectivity/DB automatically links the appropriate objects through the relevant associations.
- Member functions defined by ooObj (and inherited by all basic objects) for explicitly managing the individual associations that support versioning. You use these functions to merge branches, to repair or reconfigure genealogies of versions (for example, after deleting a version), and to support any complex versioning semantics required by an application. You work with these functions just as you work with the generated functions for an application-defined association. See "Linking With Associations" on page 317.

You can mix these member functions to suit your application's needs.

Enabling and Disabling Versioning

You must enable versioning for a basic object before you can create any versions from it. To enable versioning for a basic object, you call the setVersStatus member function on a handle to the object. The parameter is a constant of type ooVersMode that indicates the desired versioning behavior:

- Specify oocLinearVers to enable linear versioning for the object. This allows exactly one new version to be created from the object.
- Specify oocBranchVers to enable branch versioning for the object. This allows any number of new versions to be created from the object.

When you no longer want to allow versions to be created from an object, you can disable versioning by calling the `setVersStatus` member function on a handle to the object, specifying `oocNoVers` as the parameter.

To find out whether versioning is enabled for an object, you can call the `getVersStatus` member function on a handle to the object.

You can enable, disable, or change the versioning behavior of a basic object at any time.

EXAMPLE This example enables linear versioning for an instance of `Manuscript`.

```
// Application code file
#include "publisher.h"
...
ooHandle(Manuscript) msH;
... // Set msH to reference the desired manuscript

// Check whether versioning is already enabled; if not, enable
// linear versioning
if (msH.getVersStatus() == oocNoVers) {
    msH.setVersStatus(oocLinearVers);
}
```

Creating a Version

After versioning is enabled for a basic object, a new version is created when you open the object for update—for example, by calling the `update` member function on a handle to the object. If the object is already open through one or more handles, you must first close the object (by closing all open handles to it) and then open the object for update.

If linear versioning is enabled for an object, you can create exactly one new version from it. If branch versioning is enabled, you can close and reopen the object repeatedly to create multiple new versions. Each version has its own object identifier and is clustered with the object from which it was created.

When you create a new version from a basic object:

- The handle through which you opened the basic object is set to reference the new version.
- Objectivity / DB automatically links the basic object and the new version to each other through their `nextVers` and `prevVers` associations.

- The new version is enabled for the same versioning behavior (linear or branch) as the original basic object.
- If a linear version is created, versioning is automatically disabled for the original basic object to prevent additional versions from being created from that object.

NOTE If the versioned objects are in a genealogy, a newly created version does not automatically become the default version. You must set the default version explicitly; see “Changing the Default Version” on page 458.

It is common practice to disable versioning for a new version until that version is itself ready to be versioned. This allows you to work with the new version over multiple transactions without inadvertently creating unwanted versions from it.

EXAMPLE This example finds Sam’s original manuscript by its scope name (`"Sam1"`), enables linear versioning for it, and closes the manuscript. It then reopens the manuscript for update, which creates a new linear version (`"Sam2"`) from it. Versioning is disabled for the new version to prevent unwanted versions from being created during development.

```
// Application code file
#include "publisher.h"
...
ooHandle(ooDBObj) dbH;
ooHandle(Manuscript) msH; // Manuscript is derived from ooObj

...     // Set dbH to reference the database in which
        // manuscripts are scope-named

// Find Sam's first manuscript and set msH to reference it
msH.lookupObj(dbH, "Sam1");

// Enable linear versioning for the found manuscript
msH.setVersStatus(oocLinearVers);

// Make sure the manuscript is closed
msH.close();
```

```
// Create the new next version of the manuscript
if (msH.update()) {
   // msH now references the new version
   // Disable versioning for the new version
   msH.setVersStatus(oocNoVers);
   // Give the new version a scope name
   msH.nameObj(dbH, "Sam2");
   // Work with the new version
   msH->reformat();
   ...
}
else {
   cerr << "Manuscript already has a next version." << endl;
   ...
}
```

The following code finds Sam's original manuscript by its scope name ("Sam1"), enables branch versioning for it, and closes the manuscript. It then reopens the manuscript for update to create two new branch versions ("Sam3" and "Tess1") from it. (This code could be used either instead of, or in addition to, the previous code.)

```
// Application code file
#include "publisher.h"
...
ooHandle(ooDBObj) dbH;
ooHandle(Manuscript) msH, sam3H, tess1H;

...     // Set dbH to reference the database scope object
// Find Sam's first manuscript and set msH to reference it
msH.lookupObj(dbH, "Sam1");

// Enable branch versioning for the found manuscript and close it
msH.setVersStatus(oocBranchVers);
msH.close();

// Assign the closed original manuscript to another handle
sam3H = msH;

// Create one branch version of the manuscript
sam3H.update();
// Disable versioning for the new version
sam3H.setVersStatus(oocNoVers);
// Give new version a scope name
sam3H.nameObj(dbH, "Sam3");
```

```
// Assign the closed original manuscript to another handle
tess1H = msH;

// Create another branch version of the manuscript
tess1H.update();
// Disable versioning for the new version
tess1H.setVersStatus(oocNoVers);
// Give new version a scope name
tess1H.nameObj(dbH, "Tess1");
// Disable versioning for the original manuscript
msH.setVersStatus(oocNoVers);
...     // Work with the branch versions through sam3H and tess1H
```

Creating a Genealogy

You can create a genealogy to keep track of the various versions of an object. The genealogy represents the versioned entity itself; the objects within the genealogy are the individual versions of this entity. At any particular time, exactly one version in the genealogy is the default version. The semantics of the default version are up to your application; normally the default version is the one intended for use by other objects.

You initialize the genealogy by setting its default version. The default version becomes the first (and only) object in the genealogy. Whenever you create a new version from an object in a genealogy, the new version is automatically added to the genealogy.

NOTE To ensure that a genealogy is linked to *all* versions of an object, you should create the genealogy along with the initial version of the object and set the initial version to be the genealogy's first default version. If versions were created before the genealogy's first default version is set, you can add these versions to the genealogy; see "Adding Pre-existing Versions to a Genealogy" on page 457.

The persistence-capable class `ooGeneObj` represents a basic genealogy. Unless you need to keep application-specific information about a genealogy, you use this class; otherwise, you can define your own genealogy class as described in "Creating a Custom Genealogy" on page 452.

Creating a Basic Genealogy

You create a basic genealogy by making some existing object its default version. The usual way to do this is to call the `setDefaultVers` member function on a handle to the desired object. If this object is not already in a genealogy, this function creates a genealogy, an instance of `ooGeneObj`.

EXAMPLE This example creates the initial version of a manuscript. It then creates a genealogy for versions of the manuscript by making the manuscript the default object. It follows the `geneObj` link from the manuscript to find the genealogy, then names the genealogy in the scope of a database.

```
// Application code file
#include "publisher.h"
...
ooHandle(Manuscript) msH;
ooHandle(ooGeneObj) genH;
ooHandle(ooDBObj) dbH;

// Create the initial version of the manuscript
msH = new(...) Manuscript(...);
...                 // Set the manuscript attributes

// Enable branch versioning for the manuscript
msH.setVersStatus(oocBranchVers);

// Create a genealogy with the manuscript as its default version
msH.setDefaultVers();

// Follow the geneObj link to find the genealogy
msH->geneObj(genH);

// Name the genealogy in the scope of the database
...     // Set dbH to reference the database scope object
genH.nameObj(dbH, "Sam's Book");
```

The code assigns an editor to work on the manuscript. It looks up the genealogy for the manuscript, follows the `defaultVers` link to find the current default version, and creates a new version of the default for the editor to work on. An `Editor` object's `workingMs` reference attribute links it to its version of the manuscript.

```
// DDL file publisher.ddl
…
class Editor : public ooObj {
public:
   …
   ooRef(Manuscript) workingMs;
};
```

```
// Application code file
#include "publisher.h"
…
ooHandle(Manuscript) msH, defaultH;
ooHandle(ooGeneObj) genH;
ooHandle(ooDBObj) dbH;
ooHandle(Editor) edH;

…    // Set edH to reference the desired editor

// Look up the genealogy in the scope of the database
…    // Set dbH to reference the database scope object
genH.lookupObj(dbH, "Sam's Book");

// Follow link to find the default version
genH->defaultVers(defaultH);

if (defaultH.getVersStatus() != oocNoVers) {
   // Create the editor's version
   msH = defaultH;
   msH.update();
   // Link the editor to the new version
   edH->workingMs = msH;
}
else
   cerr << "Can't create new version from default" << endl;
}
```

Creating a Custom Genealogy

You typically define a custom genealogy class to represent versioned entities of a particular class. A custom genealogy is derived from the `ooGeneObj` class and can have application-specific attributes and associations.

For example, suppose a publishing house employs a number of publishers, each of whom manages the development of a number of books that are written on contract with the publishing house. A publisher tracks multiple manuscripts of each book. You could represent a book as a custom genealogy—for example, an

instance of class BookGene—that maintains associations with the individual versions of the book's manuscript. Each book genealogy could have attributes for its genre and information needed to arrange for its printing, such as the target number of pages, the target print date, and so on. A publisher, represented as a Publisher object, can then be linked by a pair of bidirectional associations to the book genealogy for each book that he or she manages. A book genealogy makes it easy for the publisher to find particular manuscript versions.

Figure 20-6 shows a publisher object (Publisher1) linked to a book genealogy (Sam's Book) that is, in turn, linked to individual manuscript versions.

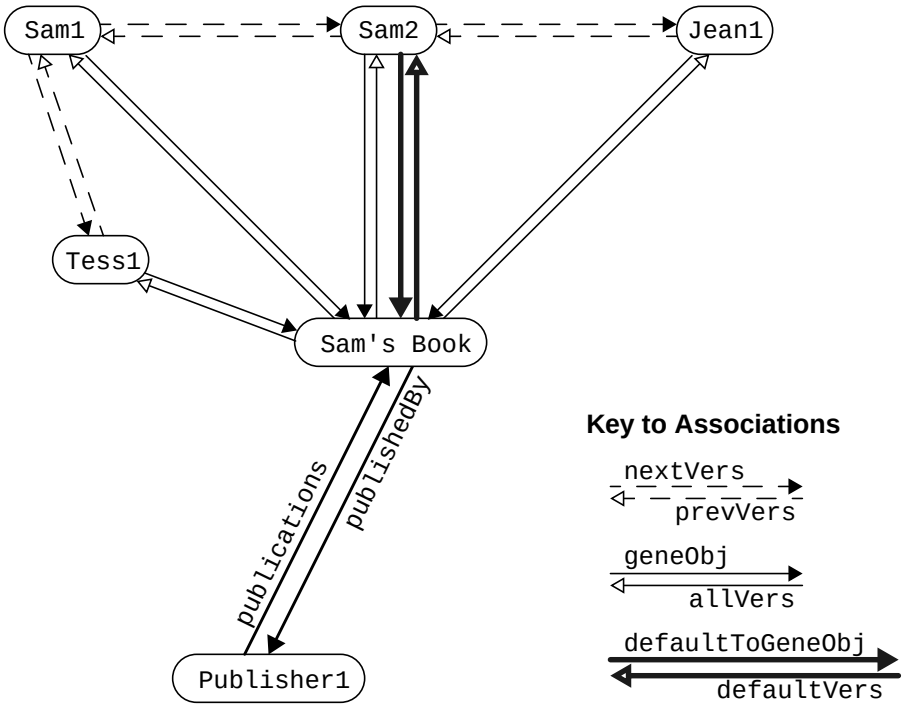


Figure 20-6 Access to Versions Through Custom Genealogy

Defining a Custom Genealogy Class

You define a custom genealogy class in a DDL file just as you would any persistence-capable class. A custom genealogy class derives from `ooGeneObj` and can have application-defined attributes and associations.

EXAMPLE This example defines a `Publisher` class and a custom genealogy class for representing books, called `BookGene`. Bidirectional associations `publishedBy` and `publications` link the `Publisher` class and the `BookGene` class.

```
// DDL file publisher.ddl
class BookGene : public ooGeneObj {
public:
   ooVString genre;
   ...                  // Additional attributes
   ooRef(Publisher) publishedBy <-> publications[];
};

class Publisher : public ooObj {
public:
   ...
   ooRef(BookGene) publications[] <-> publishedBy;
};

class Manuscript : public ooObj {
public:
   ooVString title;
   ooVString &get_title() {return title;}
   ...
};
```

Providing Version-Accessor Functions

The purpose of a custom genealogy is typically to ensure that other objects always access the default version. To facilitate this, you can define “accessor” member functions in the custom genealogy class that find the default version and return its attribute values.

EXAMPLE This example defines the `getWorkingTitle` member function in the `BookGene` class for printing a book's current working title (that is, the title of the default manuscript).

```
// Application code file
#include "publisher.h"
...
ooStatus BookGene::getWorkingTitle(ooVString &title) {
   ooHandle(Manuscript) defaultMsH;

   // Find the book genealogy's default manuscript
   defaultVers(defaultMsH);
   // Get the title of the default manuscript
   title = defaultMsH->get_title();
   return oocSuccess;
}
```

The `printWorkingTitles` member function of the `Publisher` class finds all of a publisher's book genealogies and prints their working titles.

```
// Application code file
#include "publisher.h"
...
ooStatus Publisher::printWorkingTitles() {
   ooItr(BookGene) booksI;     // Iterator for book genealogies
   ooVString &title;

   // Initialize booksI to find associated book genealogies
   publications(booksI);
   while (booksI.next()) {
      booksI->getWorkingTitle(title);
      ...                      // Print title
   }
}
```

A custom genealogy class could also provide convenient ways of finding all versions or particular versions. For example:

- A member function could call the genealogy's `allVers` member function to initialize a specified object iterator to find all of the versions. As with any to-many association, it could optionally filter the found objects based on their attribute values. See "Following To-Many Association Links" on page 325.
- If the application needs to be able to find individual versions, one member function of the genealogy class could create a new version from a specified version, giving it a specified name. The function could use the genealogy

object as the scope object for naming the versions. A second member function of the genealogy class could then find an individual version by looking up its scope name. See “Finding an Object by Scope Name” on page 335.

Setting Up a Custom Genealogy

You create a custom genealogy as you would any persistent object—by calling the `new` operator on the genealogy class and assigning the result to a handle. You can then link the genealogy to its first default object with the `defaultVers` and `defaultToGeneObj` associations. These two associations are inverses, so you need only set one of the two links explicitly; the inverse link is set automatically. That is, you could link the objects in either of the following ways:

- Through a handle to the custom genealogy, you can call the `set_defaultVers` member function, specifying a handle to the desired default version.
- Through a handle to the desired default version, you can call the `set_defaultToGeneObj` member function, specifying a handle to the custom genealogy.

As with any genealogy, once you have set the first default, any version that is created from an object in the genealogy is automatically added to the genealogy.

EXAMPLE This example creates an instance of the `BookGene` class, sets its attributes, and links it to an instance of the `Publisher` class.

```
// Application code file
#include "publisher.h"
...
ooHandle(BookGene) bookH;
ooHandle(Manuscript) msH;
ooHandle(Publisher) publisherH;

// Create the new book genealogy and set its attributes
bookH = new(...) BookGene();
bookH->genre = "Technical";
...     // Set other attributes

// Create the initial version of the manuscript
msH = new(...) Manuscript(...);
...                 // Set the manuscript attributes

// Enable branch versioning for the manuscript
msH.setVersStatus(oocBranchVers);
```

```
// Set the manuscript to be the book genealogy's default version
bookH->set_defaultVers(msH);

// Link the publisher to the new book genealogy
publisherH = ...    // Set publisherH to the desired publisher
publisherH->add_publications(bookH);
```

Adding Pre-existing Versions to a Genealogy

You typically create your genealogy when you create the first version of the object. This guarantees that all versions will be included in the genealogy. However, if you create the genealogy after creating various versions, you can explicitly link each pre-existing version to the genealogy with the `geneObj` and `allVers` associations. These two associations are inverses, so you need only set one of the two links explicitly; the inverse link is set automatically. That is, you could link the objects in either of the following ways:

- Through a handle to a pre-existing version, you can call the `set_geneObj` member function, specifying a handle to the genealogy.
- Through a handle to the genealogy, you can call the `add_allVers` member function, specifying a handle to a desired pre-existing version.

EXAMPLE This example creates a genealogy with a particular object as its default, then adds all previous versions of the default to the genealogy. This code would be sufficient for strictly linear versioning. However, if any of the previous versions supported branching versioning, additional code would be required to add all their next versions to the genealogy.

```
// Application code file
#include "myClass.h"
...
ooHandle(ooObj) defaultH, existingVersionH;
ooHandle(ooGeneObj) genH;
...    // Set defaultH to reference the desired object

// Create a genealogy with the specified default version
defaultH.setDefaultVers();

// Follow the geneObj link to find the genealogy
defaultH->geneObj(genH);

// Initialize existingVersionH to reference the default's
// previous version
defaultH->prevVers(existingVersionH);
```

```
// Follow preVers links to add each previous version to the
// genealogy
while (existingVersionH) {
   // Add this version to the genealogy
   genH->add_allVers(existingVersionH);
   // Get the previous version
   existingVersionH->prevVers(existingVersionH);
}
```

Changing the Default Version

You can change the genealogy's default version at any time by calling the `setDefaultVers` member function on a handle to an object that already belongs to the genealogy. The `setDefaultVers` member function automatically sets the `defaultVers` link from the genealogy object to its new default version and the inverse `defaultToGeneObj` link from the default version to the genealogy.

Alternatively, you can explicitly adjust the `defaultVers` and `defaultToGeneObj` links, by first calling the genealogy's `del_defaultVers` member function to remove the old default version, and then calling the genealogy's `set_defaultVers` member function to set the new default version.

Merging Version Branches

You can merge two or more version branches by indicating that a version on one branch is derived from a version on the other branch. You do so by linking the objects with the `derivedFrom` and `derivative` associations. These two associations are inverses, so you need only set one of the two links explicitly; the inverse link is set automatically. That is, you can call either the `add_derivedFrom` member function of the derived version *or* the `add_derivatives` member function of the secondary ancestor version. (The versioned objects inherit these member functions from class `ooObj`.)

EXAMPLE Assume that Sam's original manuscript has two version branches, one containing a version called `Tess1` and the other ending in a version called `Pat1` (see Figure 20-5 on page 445). The following code merges the two branches by creating a `derivedFrom` link from `Pat1` to `Tess1`.

```
// Application code file
#include "publisher.h"
...
ooHandle(ooDBObj) dbH;
ooHandle(Manuscript) pat1H, tess1H;

// Find the database in which manuscripts are scope-named
dbH = ...

// Find Pat1's manuscript and set pat1H to reference it
pat1H.lookupObj(dbH, "Pat1");

// Find Tess's manuscript and set tess1H to reference it
tess1H.lookupObj(dbH, "Tess1");

// Set Tess1 as the destination object of Pat1's derivedFrom
// association
pat1H->add_derivedFrom(tess1H);
```

As an alternative to the last statement above, you could create a `derivatives` link from `Tess1` to `Pat1`, as shown in the following comment:

```
// tess1H->add_derivatives(pat1H);
```

Finding Versions

If you have a handle to a basic object, you can find its next version(s), previous version, default version, secondary ancestor version(s), derived version(s), and all versions in the entire genealogy.

Finding the Next Versions

You can find the next linear version or the next branch versions of an object. To do so, you call the `getNextVers` member function on a handle to the object, passing as a parameter an object iterator of class `ooItr(ooObj)` to be initialized. If no next version exists, the object iterator's iteration set is empty, so its `next` member function will return `oocFalse`. See "Object Iterators" on page 293 for information about working with an object iterator.

By default, this function initializes the iterator to find all next versions without opening them; you can specify an optional open mode if you want to open each next version. You typically do not use the `oocUpdate` mode because this may cause the creation of new versions.

EXAMPLE This example initializes an object iterator to find all next versions of a given manuscript.

```
// Application code file
#include "publisher.h"
...
ooHandle(Manuscript) msH;
ooItr(ooObj) nextI;
ooStatus rc;

...    // Set msH to reference the desired manuscript

// Initialize nextI to find all next versions of the manuscript
rc = msH.getNextVers(nextI);
if (rc) {
   while (nextI.next()) {   // Iterator over the next versions
      ...
   }
}
```

As an equivalent alternative, you can find the next versions of a basic object by calling the `nextVers` member function of that object itself. This function initializes an object iterator to find all destination objects linked by the `nextVers` association. If desired, you can filter the next versions based on their attribute values. See “Following To-Many Association Links” on page 325.

EXAMPLE This example initializes an object iterator to find those next versions of a given manuscript in which the title has not been changed.

```
// Application code file
#include "publisher.h"
...
ooHandle(Manuscript) msH;
ooItr(ooObj) nextI;
pred char[512];
ooStatus rc;

...    // Set msH to reference the desired manuscript

// Get the manuscript's title
const char *curTitle = msH->get_title();

// Create a predicate string that tests for that title
sprintf(pred, "title = %s", curTitle);
```

```
// Initialize nextI to find next versions with same title
rc = msH->nextVers(nextI, pred);
if (rc) {
   while (nextI.next()) {   // Iterator over the next versions
      ...
   }
}
```

Finding the Previous Version

You can find the previous version of a basic object. To do so, you call the `getPrevVers` member function on a handle to the object; the parameter is a handle to be set to the previous version. If no previous version exists, the specified handle is set to null. The `getPrevVers` function takes a general-purpose handle of class `ooHandle(ooObj)` as its parameter. If you know the class of the previous version, you can instead pass a type-specific handle as the parameter.

EXAMPLE This example finds the previous version of a given manuscript. It sets a type-specific handle of class `ooHandle(Manuscript)` to reference the previous version and then tests whether a previous version was found.

```
// Application code file
#include "publisher.h"
...
ooHandle(Manuscript) msH;
ooHandle(Manuscript) prevH;
ooStatus rc;
...    // Set msH to reference the desired manuscript

// Find the previous version
rc = msH.getPrevVers(prevH);

if (rc) {
   // Test whether there is a previous version
   if (prevH)
      printf("There is a previous version of the manuscript\n");
   else
      printf("No previous version exists\n");
}
else
   printf("Attempt to find previous version failed\n");
}
```

As an equivalent alternative, you can find the previous version of a basic object by calling the `prevVers` member function of that object itself. This function finds the destination object linked by the `prevVers` association. See “Following To-One Association Links” on page 324.

Finding the Default Version

If an object belongs to a genealogy, you can get the current default version of the genealogy. Normally an object belongs to a genealogy if it is the genealogy’s first default version *or* it is a version created from an object that already belonged to the genealogy.

To find an object’s default version, you call the `getDefaultVers` member function on a handle to the object; the parameter is a handle to be set to the default version. If no default version exists, or if the object was created before any default version was set, the specified handle is set to null. The `getDefaultVers` function takes a general-purpose handle of class `ooHandle(ooObj)` as its parameter. If you know the class of the default version, you can instead pass a type-specific handle as the parameter.

EXAMPLE This example finds the default version of the manuscript that a particular editor is working on. An `Editor` object has a reference attribute named `workingMs` that links it to its version of the manuscript; see page 452. The code first gets the editor’s version of the manuscript, then calls that object’s `getDefaultVers` member function.

```
// Application code file
#include "publisher.h"
...
ooHandle(Manuscript) msH, defaultH;
ooHandle(Editor) edH;

...     // Set edH to reference the desired editor

// Find the Editor's version of the manuscript
msH = edH->workingMs;

// Find the default version and set defaultH to reference it
msH.getDefaultVers(defaultH);
```

As an equivalent alternative, you can find an object’s default version by finding the genealogy to which the object belongs (call the object’s `geneObj` member function), and then finding the default version from the found genealogy (call the genealogy’s `defaultVers` member function).

Finding Versions in Merged Branches

After you have merged two or more version branches into an object, you can find the object's secondary ancestor version(s). To do so, you call the `derivedFrom` member function on a handle to the object. This function initializes an object iterator to find all destination objects linked by the `derivedFrom` association.

Similarly, if an object is itself a secondary ancestor version, you can find any versions that are derived from it. To do so, you call the `derivatives` member function on a handle to the object. This function initializes an object iterator to find all destination objects linked by the `derivatives` association.

In either case, you can filter the found objects, if desired, based on their attribute values. See "Following To-Many Association Links" on page 325.

Finding All Versions in a Genealogy

You can find all of the versions in a genealogy from any version in the genealogy. To do this, you first find the genealogy by calling the version's `geneObj` member function. You then call the genealogy's `allVers` member function. This function initializes an object iterator to find all destination objects linked by the `allVers` association.

EXAMPLE This example finds all versions of the manuscript that a particular editor is working on. An `Editor` object has a reference attribute named `workingMs` that links it to its version of the manuscript; see page 452.

```
// Application code file
#include "publisher.h"
…
ooHandle(Editor) edH;
ooHandle(ooGeneObj) genH;
ooItr(ooObj) msI;

…     // Set edH to reference the desired editor

// Find the Editor's version of the manuscript
msH = edH->workingMs;

// Find the genealogy and set genH to reference it
msH->geneObj(genH);

// Initialize msI to find all manuscript versions
genH->allVers(msI);
…
```

Deleting a Version

You delete a version as you would any other persistent object (see “Deleting a Persistent Object” on page 196). When you delete a version, Objectivity / DB automatically deletes its links to any genealogy and to any next, previous, derived-from, and derivative versions.

Deleting a version may leave a gap in a chain of next and previous versions. You can repair the gap in a chain of versions by linking the two adjacent versions with the `nextVers` and `prevVers` associations.

Figure 20-7 shows a chain of three versions in which the middle version `V2` is deleted and the first version `V1` is linked to the third `V3`.

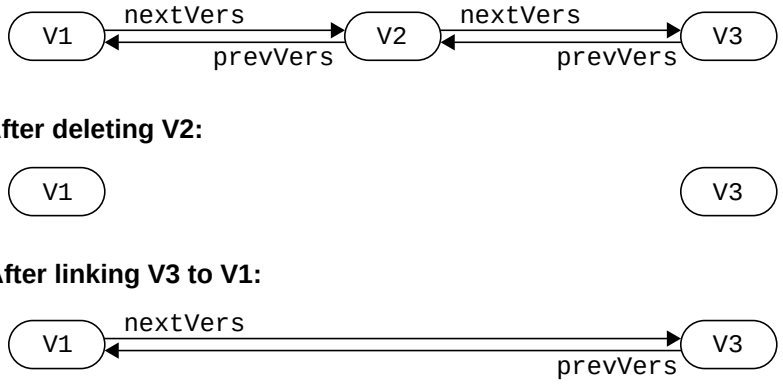


Figure 20-7 Deleting a Version

Because `nextVers` and `prevVers` are inverse associations, you can link objects by setting either one of the two links explicitly; the inverse link is set automatically. That is, you could link the objects in Figure 20-7 in either of the following ways:

- Through a handle to the previous version (`V1`), you can call that version’s `add_nextVers` member function, specifying a handle to the desired next version (`V3`):

  ```
  v1H->add_nextVers(v3H);
  ```

- Through a handle to the next version (`V3`), you can call that version’s `set_prevVers` member function, specifying a handle to the desired previous version (`V1`):

  ```
  v3H->add_prevVers(v1H);
  ```

Customizing the Created Version

When you define a class of basic objects that are to be versioned, you can arrange for each new version to adjust its values and links immediately after it is created. To do this, you override the virtual `ooNewVersInit` member function that is inherited from class `ooObj`.

A new version's `ooNewVersInit` member function is called immediately after the bit-wise copy is performed. As defined by the class `ooObj`, `ooNewVersInit` simply returns the success status. A custom `ooNewVersInit` function can perform any necessary operations on attribute data members for which bit-wise copying is inadequate. For example, the `ooNewVersInit` function might initialize attributes that should not be copied. Similarly, the `ooNewVersInit` function could propagate versioning to associated or referenced objects, linking the newly created version to a new version of each destination object.

Because a basic object can only be versioned during an update transaction, `ooNewVersInit` is always called from within a transaction, so it does not have to start a transaction itself. A custom `ooNewVersInit` function normally performs the same operations as a custom `ooCopyInit` function. For an example, see "Customizing the Copy Operation" on page 199.

21

Using Debug Mode

Debug mode provides debugging support by making it easier for you to locate software errors in Objectivity/C++ applications.

This chapter describes:

- General information about debug mode
- How to activate debug mode and specify a debug file
- Data verification
- Event tracing

Understanding Debug Mode

You can use debug mode to detect common errors made using the Objectivity/C++ programming interface. For example, it is not unusual in early development to modify an object that was opened for read (not for update), or to write past the boundary of an object. You may also want to perform data verification to detect data corruption caused by a bug either in the application or in Objectivity/C++. Data verification is particularly useful in the early phase of development, when the software is relatively unstable. Debug mode helps you detect problems early and prevent corrupted data from being written out to the federated database.

Debug mode supports two debugging activities:

- Data verification at the basic-object, logical-page, and container levels
- Event tracing for various Objectivity/DB operations on basic objects, pages, and containers

By default, debugging information is directed to the standard error device. However, you can specify a debug file; if you do so, debugging information is directed to that file instead.

An application will experience performance degradation when using debugging because the Objectivity / C++ software performs extra verification and tracing. However, this degradation should not be an issue because applications are not expected to enable the debugging activities in a production environment.

Activating Debug Mode

You can use debug mode only in an executable application that was linked with the debug version of the Objectivity / DB library. For details about linking your application with this library, see the *Installation and Platform Notes* for your platform. When you run the resulting application, you can turn the debugging options on and off without recompiling or relinking your application.

To enable the desired debugging activities, you set the corresponding environment variables before starting the application. The following table contains a summary of the environment variables for the various debugging activities. Most of these variables are used as switches—that is, they must be set to enable an activity, but their values are not significant.

Debugging Activity	Variable	Value	Meaning
Any	`OO_DEBUG_FILE`	Pathname of output file for debugging information	Write debugging information to the specified debug file
Data Verification	`OO_DEBUG_VERIFY_OBJECT`	(none)	Activate basic object verification
	`OO_DEBUG_VERIFY_PAGE`	(none)	Activate page verification
	`OO_DEBUG_VERIFY_CONTAINER`	(none)	Activate container verification
Event Tracing	`OO_DEBUG_TRACE_OBJECT`	(none)	Activate tracing for operations on basic objects
	`OO_DEBUG_TRACE_PAGE`	(none)	Activate tracing for page operations
	`OO_DEBUG_TRACE_CONTAINER`	(none)	Activate tracing for container operations
	`OO_DEBUG_TRACE_DATABASE`	(none)	Activate tracing for database operations

You use your operating system to set the environment variables. For example, in the C shell on a UNIX platform, you can set environment variables with the `setenv` command.

Debug File

Data verification and event tracing both generate debugging information that can help you identify problems in your application. By default, all debugging information is written to the standard error device. If you want to collect debugging information in a *debug file* instead, you can set the `OO_DEBUG_FILE` environment variable to the pathname for the desired file.

This chapter uses the term *debug file* to mean the output device for debugging information—either the standard error device or the file indicated by the `OO_DEBUG_FILE` variable.

Data Verification

Debug mode provides data verification at the basic-object, logical page, and container levels, so you can detect problems earlier and generate trace information, such as the before-image and after-image of a logical page. This information can help you track down the cause of the problem. Verification also prevents corrupted data from being written out to disk, thus avoiding permanent damage to the federated database.

Basic Object Verification

To activate basic-object verification, set the `OO_DEBUG_VERIFY_OBJECT` environment variable.

When you activate basic object verification, Objectivity/C++ maintains the before-image of each basic object it reads from the federated database. At the end of each transaction, it verifies that read-only objects have not been modified. When a problem is detected, the before-image and the modified page containing the basic object are written to the debug file.

NOTE Currently, debug mode does not support verification of large objects (objects larger than a page).

Basic object verification can enable you to detect the following errors:

- Modifying a basic object that was open for read, but not for update
- Writing past the end of a basic object and destroying an adjacent basic object that was open for read
- Writing to an invalid pointer, partially overwriting a basic object that was open for read

Page Verification

To activate page verification, set the `OO_DEBUG_VERIFY_PAGE` environment variable.

When you activate page verification, Objectivity/C++ verifies the consistency of each logical page after reading it from the federated database and before writing it back. It checks each page after reading and reports any problems before the application or Objectivity/C++ has a chance to operate on the (inconsistent) data. Checking a page before writing it prevents corruption of the federated database.

The following are examples of the checks performed:

- Free space + space used == storage page size
- Offsets of basic objects lie within the page
- Type numbers and sizes of basic objects are valid
- Basic objects do not overlap each other
- Page-header information, such as container ID and logical page number, are valid

Container Verification

To activate page verification, set the `OO_DEBUG_VERIFY_CONTAINER` environment variable.

When you activate container verification, Objectivity/C++ verifies the page-allocation information for each container as it is opened and closed. Container verification helps detect internal Objectivity/C++ bugs, such as allocating the same physical page more than once.

Event Tracing

When you are debugging a problem, it is useful to know the sequence of events that occurred. For example, if you can recreate an error in which you try to access a basic object and find that the object does not exist, you can turn on event tracing to log all basic-object operations. You can then determine whether the object has been deleted (a bug in the application), or has somehow disappeared (a possible Objectivity/C++ bug). Generating a trace of the events can help you narrow down the cause of the problem.

You can trace operations on basic objects, pages, containers, and databases:

- To trace operations of basic objects, such as create, open, close, delete, and resize, you set the `OO_DEBUG_TRACE_OBJECT` environment variable.
- To trace page operations, such as page read, write, and allocate, you set the `OO_DEBUG_TRACE_PAGE` environment variable.
- To trace container operations, such as create, open, close, and delete, you set the `OO_DEBUG_TRACE_CONTAINER` environment variable.
- To trace database operations, such as create, open, close, and delete, you set the `OO_DEBUG_TRACE_DATABASE` environment variable.

22

Signal Handling

Objectivity/C++ provides a predefined signal handler to respond to various signals that may be raised by an application's operating environment. By default, when `ooInit` initializes an Objectivity/C++ process, it installs this predefined signal handler. Your application can include its own signal handler that is used instead of, or in addition to, the predefined signal handler.

This chapter describes:

- The predefined signal handler
- How to write application-defined signal handlers
- How to ignore signals

NOTE When the operating system detects a problem condition in an Objectivity/C++ application, it raises a *signal*; within the application, a signal handler is called to respond to the signal. In contrast, when an Objectivity/C++ function or an application-defined function detects an error condition, it signals an *error*; an error handler is called to respond to the error. Chapter 23 describes error handlers.

Objectivity-Defined Signal Handler

When an Objectivity/DB process traps a signal that can cause process termination, the Objectivity-defined signal handler:

1. Performs any necessary cleanup, such as aborting an active transaction.
2. Reinstalls any signal handler that was installed before the call to `ooInit`.
3. Reraises the signal to the reinstalled signal handler.

See the *Installation and Platform Notes* for your platform for a list of the signals that are trapped by the predefined Objectivity/DB signal handler. Note that a signal to terminate the process cannot be trapped. Consequently, no cleanup is performed when Objectivity/C++ receives such a signal.

Application-Defined Signal Handlers

You can install an application-defined signal handler to be used in conjunction with the predefined Objectivity/C++ signal handler or to be used alone. A signal handler is a void function that takes as its parameter the `int` identifying the signal to be handled. (Signal handlers for some signals may take additional parameters; see the documentation of the `signal` function for your programming environment.)

Defining a Signal Handler

You can define a separate function to handle each different signal, or a single function to handle all signals to which your application needs to respond. A signal handler should perform whatever response your application requires for the particular signal. If necessary, it can perform Objectivity/DB operations such as aborting a transaction.

NOTE For simplicity, this chapter uses the singular term *application-defined signal handler* even though a particular application may have several such functions. If you have multiple signal-handler functions, you write and install each one just as you would do for a single application-defined signal handler.

Installing an Application-Defined Signal Handler

You call the C library function `signal` to install the desired signal handler for a particular signal. You must make one call to `signal` for each different signal that you want to handle, specifying the signal and the function that is to handle that signal. This function returns a function pointer to the old signal handler for the specified signal.

Once a signal handler has been installed for a particular signal, it is called whenever that signal is raised.

Using Both Kinds of Signal Handlers

You can install an application-defined signal handler for use in conjunction with the predefined Objectivity / C++ signal handler.

If you want the predefined signal handler to be called first, you install your signal handler *before* calling `ooInit`. After responding to a signal, the predefined signal handler automatically reinstalls any previously installed signal handler and reraises the signal to it.

If you want your application-defined signal handler to be called first, you must install it *after* calling `ooInit`. Furthermore, you must implement your signal handler to reinstall any previously installed signal handler and to reraise the signal:

1. When you install your signal handler, explicitly save the old signal handler for each signal that your function can handle.
2. Write your signal handler so that it:
   - Performs its own operations for the signal that was raised.
   - Restores the saved handler for that signal.
   - Reraises the signal.

Using Only an Application-Defined Signal Handler

If you want to use an application-defined signal handler *without* invoking the predefined Objectivity / C++ signal handler at all, you can use either of the following two approaches:

- Write a signal handler that does not restore the previously installed signal handler or reraise the signal. Install this signal handler *after* calling `ooInit`.
- Invoke `ooInit` with the *`installSigHandler`* parameter set to `oocFalse` to prevent the predefined signal handler from being installed. You can install your own signal handler either before or after calling `ooInit`.

NOTE If you suppress the predefined signal handler entirely, your signal handler must take care of all Objectivity / DB-related cleanup, such as aborting any active transaction. If your application is multithreaded and runs on a Windows platform, your signal handler must call `ooExitCleanup` before exiting the program; see “Preparing Objectivity / DB for Shutdown” on page 103.

Example Signal Handler

This example defines a signal handler named `mySignalHandler` for a UNIX application. This signal handler prints a message for certain signals *before* the predefined signal handler is called.

The function `setup_signals` installs the signal handler for the relevant signals, saving the old handler functions as the values of global variables. The `main` function calls `setup_signals` after calling `ooInit` so that `mySignalHandler` will be called before the predefined signal handler.

```
// Application code file
#include <iostream.h>
#include <signal.h>
#include <unistd.h>     // For getpid()
#include "myClasses.h"

void mySignalHandler(int sig);
void setup_signals();

// Global variables to save old signal-handler function pointers
void (*oldint)(int);     // Old handler for SIGINT
void (*oldfpe)(int);     // Old handler for SIGFPE
void (*oldsegv)(int);    // Old handler for SIGSEGV

// Application-defined signal handler mySignalHandler
void mySignalHandler(int sig) {
   switch (sig) {
      case SIGINT:
         cout << "  Trapped SIGINT signal! " << sig << endl;
         signal(sig, oldint);     // Restore previous handler
         break;
      case SIGFPE:
         cout << "  Trapped SIGFPE signal. " << sig << endl;
         signal(sig, oldfpe);     // Restore previous handler
         break;
      case SIGSEGV:
         cout << "  Trapped SIGSEGV signal. " << sig << endl;
         signal(sig, oldsegv);    // Restore previous handler
         break;
      default:
         cout << "  Trapped an unknown signal: " << sig << endl;
         signal(sig, SIG_DFL);    // Restore previous handler
         break;
   } // End switch

   sigsetmask(0);              // Restore the default mask
```

```
   cout << "Old signal handler restored." << endl;

   cout << "Reraising the signal now!" << endl;

   raise(sig);                 // Reraise the signal
} // End mySignalHandler

// Install mySignalHandler as signal handler; save old handlers
void setup_signals() {

   oldint  = signal(SIGINT, mySignalHandler);     // Ctrl-C

   oldfpe  = signal(SIGFPE, mySignalHandler);  // Floating-pt
                                               // exception

   oldsegv = signal(SIGSEGV, mySignalHandler); // Segmentation
                                               // violation
} // End setup_signals

// Application main function
int main()
{
   int retval = 0;
   … // Non-Objectivity/DB operations

   // Initialize Objectivity/DB, installing the predefined
   // signal handler
   if (ooInit()) {
      // Install the application-specific signal handler
      setup_signals();
      … // Objectivity/DB operations
   }
   else {
      cerr << "Unable to initialize Objectivity/DB" << endl;
      retval = 1;
   }
   …
   return retval;
} // End main
```

Ignoring Signals

If you want your application to ignore a particular signal, you can call the C library functions `signal`, specifying `SIG_IGN` in place of a signal handler. Any such calls should be made *after* the call to `ooInit`.

WARNING If your application ignores a signal that Objectivity/C++ would normally trap, no Objectivity/DB-related cleanup will be performed if that signal is raised.

23

Error Handling

Objectivity/C++ provides a basic error-handling facility that can be customized by an application.

This chapter describes:

- General information about the error-handling facility
- Defining and responding to application-specific error conditions
- Checking for error conditions after a function call
- Error handlers
- Message handlers

NOTE When an Objectivity/C++ function or an application-defined function detects an error condition, it signals an *error*; an error handler is called to respond to the error. In contrast, when the operating system detects a problem condition in an Objectivity/C++ application, it raises a *signal*; within the application, a signal handler is called to respond to the signal. Chapter 22 describes signal handlers.

Understanding the Error Handling Facility

Objectivity/C++ defines error conditions that can arise during the execution of an application program. Each distinct error condition has an identifying error number and a parameterized message string that describes the problem for the application's user.

When Objectivity/C++ detects that an error condition has occurred, it *signals an error*, identifying the condition by its error number and the level of severity, and providing information about the circumstances in which the error occurred.

Error Handlers and Message Handlers

Whenever an error is signaled, the currently registered *error handler* is invoked automatically. At any given time, only one error handler can be registered. Objectivity / C++ includes a predefined error handler, which is registered by default. An application can register a different error handler at any time; the newly registered handler replaces the previous one.

The predefined error handler calls the currently registered *message handler*. It then returns a status code or terminates program execution, depending on the severity of the error.

As is the case with error handlers, at any given time, only one message handler can be registered. The predefined Objectivity / C++ message handler is registered by default. This message handler prints the message string describing the error condition that occurred. An application can register a different message handler, replacing the predefined one.

Error Context Variables

Signaling an error sets *error context variables* that record which error occurred, its level of severity, and the total number of errors that have occurred so far.

Status Codes

Many Objectivity / C++ functions return a status code—`oocSuccess` if the function was successful, and `oocError` if an error condition occurred. The calling function can examine the returned code to determine whether an error occurred. If so, it can get information about the error condition from error context variables that were set when the error was signaled.

Customizing the Error-Handling Facility

An application can customize the error-handling facility by:

- Defining its own error conditions.
- Signaling errors.
- Registering an application-defined error handler to replace the predefined error handler.
- Registering an application-defined message handler to replace the predefined message handler.

Error Handling in a Multithreaded Application

Signaled errors, error handlers, and message handlers are specific to a particular Objectivity context.

In a multithreaded application, an error detected in a particular thread is signaled in that thread's Objectivity context and is handled by the error handler and message handler that are currently registered in that Objectivity context. Context variables keep information about the last error that occurred in the particular Objectivity context.

If your application is multithreaded and registers its own error or message handler, remember to register the desired handler in each Objectivity context that your application uses. For more information about Objectivity contexts, see Chapter 5, "Multithreaded Objectivity/C++ Applications".

Defining Error Conditions

Each error condition is described by an *error-identifier structure* of type `ooError`. The two data members of the structure contain the error number and the parameterized error-message string describing the error condition. Objectivity/C++ defines error identifiers corresponding to the error conditions that it recognizes. You may define additional error identifiers for error conditions that you want your application to detect and signal.

You should define error identifiers for the error conditions relevant to a particular source module in an error-message header file for that module. Your application source files must include the appropriate error-message header files.

It is good programming practice to:

- Declare a variable with a mnemonic name for each error condition and initialize it to the corresponding error identifier. Include the module prefix in the variable names.
- Within the application source code, use these error-identifier variables to signal and test for the corresponding errors.

Figure 23-1 illustrates the error-message header file for an application with a single module.

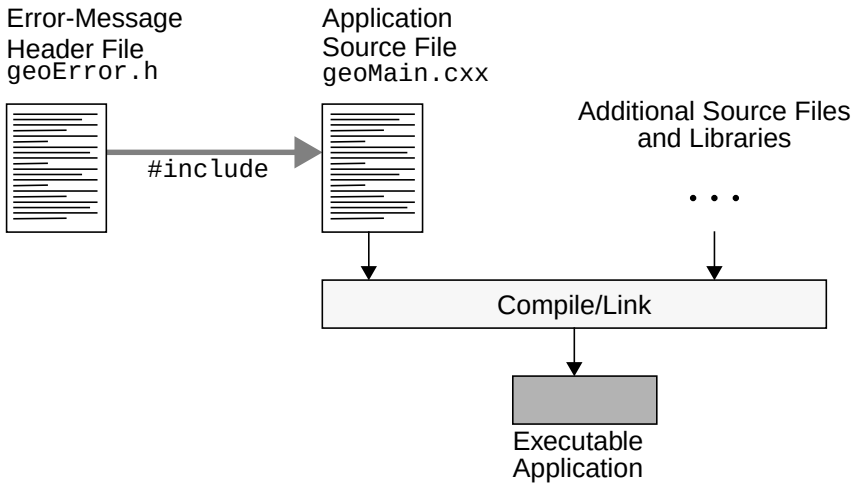


Figure 23-1 Error-Message Header File

Error Numbers

The `errorN` member of an error identifier contains a unique error number that identifies the corresponding error condition. Objectivity/C++ reserves error numbers 0 through 999999 for its error conditions.

When you define error identifiers for your application, make sure that each one has a unique error number greater than 999999. If your application is composed of multiple source modules, you may want to reserve a different range of error numbers for the errors defined in each module. For example, you could use the range 1000000 to 1000099 for error conditions that are specific to one module and the range 1000100 to 1000199 for error conditions specific to another module.

Error-Message String

The `message` member of an error identifier contains an error-message string for the corresponding error condition. The message string is formatted like a `printf` control string. The conversion specifications in the string correspond to parameters that can provide information about the situation in which the error occurred. For example, if the error condition relates to a problem with a file, the string could include the `%s` conversion specification to be filled in with the pathname of the relevant file. The function that recognizes this error condition would specify the pathname as a parameter when signaling the error.

EXAMPLE The error-message header file `geoError.h` uses the `geo` prefix in names of error-identifier variables. Errors in this module are given numbers in the range 1000000 to 1000099. Each statement in the file sets a variable to an error identifier describing an error condition that the application will check for.

The first statement sets a variable named `geoCannotOpen` to an error identifier with the error number `1000000` and the message string:

`"Cannot open file %s; aborting processing"`

When an error of this kind is signaled, the `%s` in the message string is replaced with the pathname of the file that could not be opened.

```
// Error-message header file geoError.h
#include <oo.h>

ooError geoCannotOpen = {
         1000000,
         "Cannot open file %s; aborting processing" };
ooError geoNoSuchFile = {
         1000001,
         "No such file - %s" };
ooError geoNoSuchDir = {
         1000002,
         "No such directory - %s" };
ooError geoIOError = {
         1000003,
         "System I/O Error" };
ooError geoRatioTooBig = {
         1000004,
         "Ratio %d/%d is too big" };
ooError geoRetry = {
         1000005,
         "Data temporarily unavailable" };
```

Responding to an Error Condition

When your application detects an error condition for which it has defined an error identifier, it should take the following actions to indicate the error condition exists:

1. Signal the error.
2. Inform the calling function that an error condition occurred.

Signaling an Error

You signal an error by calling the global function ooSignal. The parameters specify:

- An error level of type ooErrorLevel indicating the severity of the error
- The error identifier for the condition that occurred
- The Objectivity / DB object, if any, that was involved in the error condition
- Any parameters needed by the error-message string

When Objectivity / C++ detects an error condition, it signals the error with a similar internal call, specifying the same information.

When an error is signaled, ooSignal performs the following steps:

1. Construct an error-message string by filling parameters into the string in the message data member of the error identifier.
2. Set context variables to record information about this latest error:
   - Set the context variable oovLastErrorLevel to the error level; this variable indicates the severity level of the most recent error condition.
   - Set the context variable oovLastError to point to the error identifier; this variable indicates the most recent error condition.
3. Call the currently registered error handler. The first three parameters to the error handler are the same as the corresponding parameters to ooSignal; the fourth parameter is the string constructed in step 1.
4. Return the status code that the error handler returns.

EXAMPLE This example detects an error condition, namely the failure to open a file. It signals the error using the error identifier geoCannotOpen, defined in the example on page 483. The message string for that error identifier expects a single parameter, the name of the file that could not be opened. Thus, fileName is passed as the last parameter to ooSignal.

```
// Application code file
#include "geometry.h"
#include "geoError.h"
…
ooStatus printSummary(…) {
   FILE* fptr;                  // File pointer
   char fileName[80];           // Pathname of the file to open
   ooStatus rc = oocSuccess;    // Status code
   …
   fptr = fopen(fileName, "w+");
```

```
    // Signal an error if the file could not be opened
    if (fptr == 0) {
        return ooSignal(
                  oocUserError,   // Error level
                  geoCannotOpen,  // Error identifier
                  0,              // No relevant object
                  fileName);      // Parameter for message string
    }
    ...
    return rc;
}
```

Informing the Calling Function

Application functions that signal errors can return a status code indicating whether an error occurred. Like many Objectivity/C++ functions, these application functions should return `oocError` if an error occurred, and `oocSuccess` otherwise. When the function calls `ooSignal`, it should return the status code that `ooSignal` returns, as illustrated in the preceding example.

If the application function must be written with a return type other than `ooStatus`, it can rely on the Objectivity/C++ error context variables to indicate whether an error occurred. The application can adopt the convention that an error condition exists if either `oovLastError` or `oovLastErrorLevel` is set to anything but its nonerror value.

Table 23-1: Nonerror Values for Error Context Variables

Error Context Variable	Nonerror Value
`oovLastErrorLevel`	`oocNoError`
`oovLastError`	null pointer

As described in “Signaling an Error” on page 484, the function `ooSignal` sets the error context variables when an error occurs. If an error does not occur, a function can clear these variables to indicate that no error condition exists.

NOTE Your application should never directly modify the error context variables except to clear their values to indicate a nonerror state.

You can clear both `oovLastError` and `oovLastErrorLevel` by calling the global macro `ooResetError`. Alternatively, you can set them individually to the nonerror values shown in Table 23-1.

EXAMPLE This is an alternative to the previous example. The function `printSummary` does not return a status code, but instead uses the error context variables to indicate whether an error occurred. The function first clears information about any previous error, so the variables indicate a nonerror state. If an error is signaled (whether by `printSummary` or by a function that it calls), the error context variables will be set to indicate which error occurred.

```
// Application code file
#include "geometry.h"
#include "geoError.h"
...
void printSummary(...) {
   FILE* fptr;                  // File pointer
   char fileName[80];           // Pathname of the file to open

   // Clear the error context variables to indicate that no error has
   // occurred
   ooResetError();
   ...
   fptr = fopen(fileName, "w+");
   // Signal an error if the file could not be opened
   if (fptr == 0) {
      ooSignal(oocUserError,    // Error level
               geoCannotOpen,   // Error identifier
               0,               // No relevant object
               fileName);       // Parameter for message string
      return;
   }
   ...
}
```

Checking for Errors

After you call a function that may signal an error—either an Objectivity/C++ function or an application function—you should check whether an error occurred. Depending on the function, you can check the returned status code and/or examine the values of error context variables.

If you find that an error did occur, you should take the appropriate action. For example, if an error occurs when you try to open an Objectivity/C++ object, you should not proceed to access that object.

When you find that an application-defined error has occurred, you know that application code signaled the error, invoking an error handler—possibly an application-defined error handler. For this reason, some coordination may be necessary between software layers to establish exactly where a particular error condition will be dealt with—by the error handler itself, or by a function that checks the error context variables and finds that the error occurred.

Checking the Returned Status Code

By convention, if a function with type `ooStatus` returns the status code `oocError`, an error has occurred. When you call such a function, it is your responsibility to check the returned status code and to handle the condition in a manner appropriate to the application.

You can use a returned `ooStatus` value in a C++ condition (for example, the condition of an `if` statement); `oocSuccess` is nonzero and `oocError` is zero.

Checking the Error Context Variables

You can get information about the last error that occurred by checking the values of the Objectivity/C++ error context variables. As described in “Signaling an Error” on page 484, the function `ooSignal` sets these variables when an error occurs. As described in “Informing the Calling Function” on page 485, a function that does not return a status code can clear the variables to indicate that no error occurred.

The error context variables are global in scope, but their values are specific to the current Objectivity context. You can examine and change their values directly as you would do for any variable in the global scope.

After you call a function that returns the `oocError` status code, you can examine the error context variables to find out what error occurred and its level of severity.

You should minimize the need to test the error context variables because each test requires a thread variable lookup, which is a relatively time-consuming operation. Where possible, you should write your functions to return an error status that callers can check instead of requiring them to examine the context variables to determine whether an error occurred.

EXAMPLE The following code shows how an Objectivity/C++ application might use the error context variables to handle an error in the function `calcRect`. That function returns a status code, so the caller may assume that a return value of `oocError` indicates that an error has been signaled and the error context variables have been set.

If an error occurs in the call to `calcRect`, the caller checks whether the error was `geoRetry` (defined on page 483), which indicates that data is temporarily unavailable. If so, it repeats the call up to a specified maximum number of times.

```
// Application code file
#include "geometry.h"
#include "geoError.h"
…
ooStatus calcRect(void);    // Function prototype for calcRect
…
ooStatus myFunction(int maxTries) {
   ooStatus rc;
   …
   int retryCount = 0;

   // Call calcRect; if the call fails because data is
   // temporarily unavailable, repeat the call
   // up to maxTries times
   while (retryCount < maxTries) {
      rc = calcRect();
      if (rc == oocError) {    // An error occurred
         if (oovLastError->errorN == geoRetry.errorN) {
            // Data was temporarily unavailable; try again
            retryCount++;
         } // End if data unavailable
         else
            return rc;
      } // End if error occurred
      else
         return rc;
   } // End while
   // Exceeded maxTries without successful call to calcRect
   …
} // End myFunction
```

After you call a function that does not return a status code, you can examine the error context variables to see whether an error occurred.

EXAMPLE This is an alternative to the previous example in which the `calcRect` function must be defined as `void` to conform to a third-party calling convention. Because `calcRect` cannot return a status code, it must use the error context variables to indicate whether an error occurred. After `myFunction` calls `calcRect`, it checks the context variable `oovLastErrorLevel` to determine whether an error occurred.

Note that this alternative is less efficient than the previous example because `myFunction` must test `oovLastErrorLevel` after *every* call to `calcRect`, whereas the previous example tests this context variable only when an error is known to have occurred.

```
// Application code file
#include "geometry.h"
#include "geoError.h"
...
void calcRect(void);        // Function prototype for calcRect
...
void myFunction(int maxTries) {
   ...
   int retryCount = 0;

   // Clear the error context variables
   ooResetError();

   // Call calcRect; if the call fails because data is
   // temporarily unavailable, repeat the call
   // up to maxTries times
   while (retryCount < maxTries) {
      calcRect();
      if (oovLastError != oocNoError) {  // An error occurred
         if (oovLastError->errorN == geoRetry.errorN) {
            // Data was temporarily unavailable; try again
            retryCount++;
         } // End if data unavailable
         else
            return;
      } // End if error occurred
      else
         return;
   } // End while
   // Exceeded maxTries without successful call to calcRect
   ...
} // End myFunction
```

Error Handlers

Whenever an error is signaled, the currently registered error handler is called to handle it. Objectivity / C++ includes a predefined error handler, which is registered by default. An application may define and register its own error handler to be used instead of the predefined one.

An error handler is a function that must conform to the calling interface defined by the `ooErrorHandlerPtr` function pointer type. It takes as parameters:

- The error level
- The error identifier for the condition that occurred
- The Objectivity / DB object, if any, that was involved in the error condition
- The error-message string

An error handler returns a status code that indicates whether an error occurred.

You can obtain a function pointer to the registered error handler for the current Objectivity context by calling the global macro `ooGetErrorHandler`.

Objectivity-Defined Error Handler

The predefined error handler provided by Objectivity / C++ first constructs a formatted string describing the error condition that occurred. The string contains the error level and number and the error-message string. If an Objectivity / DB object was involved in the error condition, the string also includes the object identifier for that object. The error handler calls the currently registered message handler to print the formatted string.

The error handler's next action depends on the severity of the error:

- For a warning (level `oocWarning`), it returns `oocSuccess`.
- For a nonfatal error (level `oocUserError` or `oocSystemError`), it returns `oocError`.
- For a fatal error (level `oocFatalError`), it calls the C library function `abort`, which causes an abrupt program termination (such as a core dump on UNIX). The call to `abort` raises a signal, which is caught by the currently registered signal handler. If the predefined Objectivity / C++ signal handler is registered, it responds by aborting the active transaction and leaving Objectivity / DB in a safe state for shutdown. See Chapter 22, "Signal Handling".

Application-Defined Error Handlers

You may define a custom error handler for your application and register it with Objectivity/C++ at any time. For example, you can write a custom error handler to filter the errors that require response or to maintain a log of application-specific state.

Defining an Error Handler

Your error handler should perform whatever application-specific response is required to the particular error, making the response appropriate for the indicated error level.

When writing an error handler, you should follow these guidelines:

- Do not invoke any Objectivity/DB operations from within an error handler, because such operations will have undefined results.
- Do not throw any C++ exceptions from within an error handler. Doing so will result in a fatal error that terminates the application.
- If the error level indicates a fatal error (level `oocFatalError`), you should call the C library function `abort` to terminate the program.
- Return a status code that is appropriate for the error level. Typically, you return `oocSuccess` for a warning (level `oocWarning`) and `oocError` for a nonfatal error (level `oocUserError` or `oocSystemError`). However, if your error handler is able to determine that the error level specified by the caller is not appropriate, you may return a code that indicates the severity as judged by the error handler.

Typically, an application-defined error handler takes care of application-defined error conditions and possibly all warnings; it calls the predefined error handler to take care of errors defined by Objectivity/C++.

EXAMPLE This example shows an application-defined error handler `myErrHandler` that:

- Ignores all warnings, returning `oocSuccess`.
- Calls the predefined error handler for nonfatal and fatal errors defined by Objectivity/C++.
- Ignores all nonfatal application-defined errors, returning `oocError`.
- Aborts the program for fatal application-defined errors.

The global variable `objyErrHandler` is function pointer to the Objectivity/C++ predefined error handler. It is set when the application-defined error handler is registered. See “Registering an Error Handler” on page 493.

```
// Application code file
#include "geometry.h"
#include "geoError.h"
…
ooErrHandlerPtr objyErrHandler;    // Function pointer to
                                   // predefined error handler

ooStatus myErrHandler(
      ooErrorLevel errorLevel,        // Error level
      ooError &errorID,               // Error identifier
      ooHandle(ooObj) *contextObj,    // Relevant object
      char *errorMsg) {               // Error message string

   if (errorLevel == oocWarning) {
      return(oocSuccess);
   }
   if (errorID.errorN < 1000000) {
      // Objectivity C++ error; call the predefined
      // error handler
      return (*objyErrHandler)(errorLevel, errorID,
                               contextObj, errorMsg);
   }
   switch(errorLevel) {
      case oocUserError:
      case oocSystemError:
         // Nonfatal application-defined error
         return oocError;
      case oocFatalError:
         // Fatal application-defined error
         abort();
         break;
      default:
         // Unrecognized error level
         return oocError;
   } // End switch
} // End myErrHandler
```

Registering an Error Handler

To register an error handler, call the global macro ooRegErrorHandler, passing a function pointer to the handler function as the parameter. This function returns a function pointer to the previously registered error handler; you should save the returned value if you ever want to call or reregister the previous error handler. For example, when you first register an error handler, the returned value is a function pointer to the predefined Objectivity/C++ error handler. You can save this function pointer and use it to call the predefined error handler from within your own error handler.

In a multithreaded application, you must call ooRegErrorHandler in each Objectivity context that is to use the error handler.

EXAMPLE This example registers the error handler myErrHandler, which is defined in the previous example. It saves a function pointer to the previously registered error handler—namely, the predefined Objectivity/C++ error handler—in the variable objyErrHandler.

```
// Application code file
#include "geometry.h"
#include "geoError.h"
...
ooErrHandlerPtr objyErrHandler;    // Function pointer to
                                   // predefined error handler
...
// Register the application-defined error handler and
// save a function pointer to the predefined error handler
objyErrHandler = ooRegErrorHandler(myErrHandler);
```

The application could later reinstate the predefined error handler with the following call:

```
ooRegErrorHandler(objyErrHandler); // Restore old error handler
```

Message Handlers

The predefined Objectivity/C++ error handler calls the currently installed message handler to print a description of the error condition that occurred. Objectivity/C++ includes a predefined message handler, which is registered by default. An application may define and register its own message handler to be used instead of the predefined one.

A message handler is a function that must conform to the calling interface defined by the `ooMsgHandlerPtr` function pointer type. It is a void function that takes a string as its parameter. It should write that string to some output device.

You can obtain a function pointer to the registered message handler for the current Objectivity context by calling the global macro `ooGetMsgHandler`.

Objectivity-Defined Message Handler

The predefined message handler provided by Objectivity/C++ prints its string parameter to the *error-message output file*. By default, the error-message output file is set to the standard error device.

You can change the error-message output file at any time by calling the global function `ooSetErrorFile`, specifying the pathname for the desired file. In a multithreaded application, you must call this function in each Objectivity context whose message handler is to write errors to the specified file.

Application-Defined Message Handlers

You may define a custom message handler for your application and register it with Objectivity/C++ at any time. You typically define a message handler if you prefer not to write error messages to a file (or to the standard error device). For example, you might define a custom message handler if your application runs in a graphical environment where status and error information is customarily displayed in dialogue boxes.

Remember that the message handler will be called only if your application uses the predefined error handler, or if your own error handler calls the message handler.

Defining a Message Handler

Your message handler should display error messages following whatever convention has been chosen for your application.

EXAMPLE This example shows a simple message handler named `dlgBoxMsgHandler` that calls an application-defined graphics function `msgDialog` (not shown) to display the message in a dialogue box.

```
// Application code file
# include "myClasses.h"
...
void dlgBoxMsgHandler(char* message) {
   // Display the message in a dialog box on the screen
   msgDialog(message);
}
```

Registering a Message Handler

To register a message handler, call the global macro `ooRegMsgHandler`, passing a function pointer to the handler function as the parameter. This function returns a function pointer to the previously registered message handler; you should save the returned value if you ever want to call or reregister the previous message handler.

In a multithreaded application, you must call `ooRegMsgHandler` in each Objectivity context that is to use the message handler.

EXAMPLE This example registers the message handler `dlgBoxMsgHandler`, which is defined in the previous example.

```
// Application code file
# include "myClasses.h"
...
// Register the application-defined message handler
ooRegErrorHandler(dlgBoxMsgHandler);
```

Calling a Message Handler

You typically call a message handler only from within an application-defined error handler. Note, however, that an application-defined error handler need not call a message handler.

EXAMPLE This application-defined error handler is similar to the one in the example on page 491 except that it calls the currently registered message handler for warnings and application-defined errors.

```
// Application code file
#include "geometry.h"
#include "geoError.h"
...
ooErrHandlerPtr objyErrHandler;    // Function pointer to
                                   // predefined error handler

ooStatus myErrHandler(
      ooErrorLevel errorLevel,        // Error level
      ooError &errorID,               // Error identifier
      ooHandle(ooObj) *contextObj,    // Relevant object
      char *errorMsg) {               // Error message string

   // Get function pointer to currently registered
   // message handler
   ooMsgHandlerPtr msgHandler = ooGetMsgHandler();

   if (errorLevel == oocWarning) {
      (*msgHandler)(errorMsg);     // Call message handler
      return(oocSuccess);
   }
   if (errorID.errorN < 1000000) {
      // Objectivity C++ error; call the predefined
      // error handler
      return (*objyErrHandler)(errorLevel, errorID,
                               contextObj, errorMsg);
   }
   (*msgHandler)(errorMsg);        // Call message handler
   switch(errorLevel) {
      case oocUserError:
      case oocSystemError:
         // Nonfatal application-defined error
         return oocError;
```

```
      case oocFatalError:
         // Fatal application-defined error
         abort();
         break;
      default:
         // Unrecognized error level
         return oocError;
   } // End switch
} // End myErrHandler
```

24

Performance

Within an Objectivity/C++ application, various configuration parameters can affect runtime performance. Although their default settings work well for a wide range of applications, you may be able to improve your application's performance by reviewing runtime statistics and resetting certain configuration parameters. In addition, you can adopt usage strategies that optimize the performance characteristics that are most important to your application.

This chapter describes:

- General information about performance
- How to measure the performance of an application
- Guidelines for maximizing concurrency, runtime speed, and available space

Understanding Performance

Performance of any database application has three primary dimensions:

- Maximizing concurrency
- Maximizing runtime speed
- Maximizing available space

Some adjustments to an application are one-dimensional, enabling you to improve performance along an isolated dimension without sacrificing performance along another dimension. All too often, however, an adjustment has multidimensional effects, forcing you to sacrifice and compromise. When using this chapter to tune your application, start with the performance dimension that is most important to you. When optimizing for the other two dimensions, avoid the tendency to reverse earlier adjustments.

Measuring Performance

You can use operating system utilities, such as the UNIX `time` command or the Windows NT Task Manager, to determine whether your database application is I/O bound or CPU bound. You can use any performance-monitoring utilities available in your programming environment, such as profilers and benchmarks, to measure performance on tasks that are not directly involved in access to the federated database. You can review Objectivity/DB runtime statistics to measure performance on the Objectivity/DB operations occurring in an application.

Obtaining Runtime Statistics

To obtain Objectivity/DB runtime statistics for the current Objectivity context, you call the global function `ooRunStatus`. This function prints a wide variety of runtime statistics to the standard output device. The measurements are grouped as follows:

- *Object Manager Statistics* are object-level measurements, such as the number of databases, containers, and basic objects that were created, opened, and deleted.
- *Storage Manager Statistics* are implementation-level measurements, such as the number of buffers used and the number of disk reads.

On UNIX, you can also obtain runtime statistics while you are debugging your application. To do so, you run the `oodebug` tool from within a C++ debugger. While `oodebug` is running, you issue the command `stats`, which executes the `ooRunStatus` function. See the Objectivity/DB administration book for a description of the `oodebug` tool and instructions for running it from a C++ debugger.

EXAMPLE This example shows the output from `ooRunStatus`.

```
************************************************
Object Manager  Statistics Wed Aug 09 21:21:22 2000

** Number of federated DBs created  => 0
** Number of federated DBs opened   => 1
** Number of federated DBs closed  => 1
** Number of federated DBs deleted  => 0
**
** Number of databases created      => 0
** Number of databases opened       => 2
** Number of databases closed       => 0
** Number of databases deleted      => 0
**
```

```
** Number of containers created     => 0
** Number of containers opened      => 9
** Number of containers closed      => 11
** Number of containers deleted     => 0
**
** Number of objects Created        => 0
** Number of objects opened         => 8390304
** Number of multiple opens         => 892
** Number of new versions           => 0
** Number of objects closed         => 8390304
** Number of multiple closes        => 892
** Number of objects deleted        => 0
**
** Number of objects named          => 0
** Number of new OCBs              => 256
** Number of new associations       => 0
** Number of disassociations       => 0
** Number of associations resized   => 0
** Number of transactions started   => 1
** Number of transaction commits    => 1
** Number of commit and holds       => 0
** Number of transaction aborts     => 0
** Number of system aborts          => 0
************************************************
************************************************
Storage Manager  Statistics Wed Aug 09 21:21:22 2000

** Page  size                       => 32768
** Number of buffers  used          => 500
** Number of large buffer entries   => 200
** Number of SM objects opened      => 8390241
** Number of SM objects created     => 0
** Number of objects still  opened  => 0
** Number of buffers read           => 5
** Number of disk reads             => 33865
** Number of old pages written      => 0
** Number of new pages written      => 0
** Number of openHash calls         => 1
** Number of hash overflows         => 0
** Number of times OCs extended     => 0
** Number of Pages added to OCs     => 0
** Number of SM objects resized     => 0
********************************************
```

Understanding Runtime Statistics

The meaning of most measurements reported by `ooRunStatus` is obvious. For example, "Number of containers opened" is the number of containers that were opened in the current context before the call to `ooRunStatus`. A few measurements relate to internal Objectivity / DB objects and operations. Only a few of the measurements have a significant impact in terms of performance.

The following sections list the measurements whose meanings may not be obvious and the measurements that can help you tune your application. They are ordered as they appear in the output from `ooRunStatus`.

Object Manager Statistics

`Number of objects created`
`Number of objects opened`
`Number of objects closed`
`Number of objects deleted`

These measurements give the number of *basic objects* created, opened, closed, and deleted, respectively.

`Number of multiple opens`

This measurement is the number of persistent objects that have been opened more than once before the object is closed—that is, more than one handle to a given object was open at the same time.

In general, a large number of multiple opens is not a cause for concern. However, it may indicate that your application is passing handles by value instead of by reference. As a general C++ programming rule, you should avoid passing any large object, or any object with a complex constructor, by value.

If this measurement is large, make sure that all your functions pass object references and handles by reference instead of by value. Also check that your code does not perform multiple operations through an object reference. See "Expensive In-Memory Access" on page 214.

If your application tends to open the same object many times, you should consider using hot mode. See "Using Hot Mode" on page 514.

`Number of objects named`

This measurement is the number of persistent objects that were given scope names. You should minimize the use of scope names, and rely on other mechanisms for identifying objects for individual lookup. See "Minimizing Name Scopes" on page 516.

`Number of new OCBs`

Each open persistent object has an *Object Control Block* (OCB). Handles referencing the same object use the same OCB. Initially, 256 OCBs are created.

If this measurement is greater than 256, your application has a large number of persistent objects open simultaneously. This may indicate that the application is using more virtual memory than necessary, which can have an adverse effect on performance.

If you see a large number of OCBs, check that whenever you create a persistent object, you assign the pointer returned by `new` to a handle:

```
ooHandle(Library) libH = new(...) Library(...); // Correct
ooHandle(Book) bookH;
bookH = new(...) Book(...); // Correct
```

In particular, never assign the pointer returned by `new` to an object reference:

```
ooRef(Library) libR = new(...) Library(...); // Wrong! Use handle
ooRef(Book) bookR;
bookR = new(...) Book(...); // Wrong! Use handle
```

Also, never use `new` without assigning the resulting pointer to a handle:

```
new(...) Book(...); // Wrong! Assign to handle
```

You may also see a large number of OCBs if your application uses a large array of handles to create new persistent objects or to access existing objects. Make sure that your application actually needs references to that many objects at the same time. If it does, consider saving the references in an array of object references instead of an array of handles. When you create a new persistent object, assign its pointer to a handle, then assign that handle to the appropriate element of the object-reference array.

`Number of new associations`

This measurement is the number of association links created from a source object to a destination object.

`Number of disassociations`

This measurement is the number of association links removed from a source object.

`Number of associations resized`

A source object's non-inline associations are stored in a VArray called its system-default association array. See "Storage Requirements for Associations" on page 63 in the Objectivity/C++ Data Definition Language book. When an object is created, no space is allocated for its system-default association array. When you first create a non-inline association link for a source object, this VArray is allocated with enough space for three links. The VArray is expanded dynamically as necessary to store more association links.

Each time a source object's VArray is expanded, this measurement is incremented.

If this measurement is high relative to the number of new associations, it may be worth trying to set as many non-inline associations as possible after creating the source object. If at all possible, you should set at least one non-inline association immediately as described in "Setting Associations Early" on page 512. This will reserve space for the non-inline association links in the same logical page as the source object itself.

See also the discussion of "Number of SM objects resized" on page 505.

Storage Manager Statistics

`Number of buffers used`

This measurement is the number of buffer pages in the Objectivity / DB cache for the current Objectivity context. It can help you adjust the size of the cache. See "Optimizing the Cache Size" on page 510.

`Number of large buffer entries`

This measurement is the number of open large objects (objects that require more than one page) in the Objectivity / DB cache for the current Objectivity context. The maximum number of large objects open at any time is controlled by an attribute of the Objectivity context; see "Cache Size" on page 72. Once that maximum number is reached, large objects will be closed so that new ones can be swapped in. You can change the maximum number by calling the `ooSetLargeObjectMemoryLimit` global function.

`Number of SM objects opened`

Storage manage (SM) objects are internal objects used to implement the slots on a page. Each fixed part of a persistent object has its own slot; each VArray also has its own slot.

`Number of buffers read`

This measurement is the number of reads from the cache—that is, the number of times an accessed page was already in the Objectivity / DB cache. A buffer read occurs when the object to be accessed is on a page that was read into the cache in a prior transaction. This measurement can be used to help you adjust the size of the cache. See "Optimizing the Cache Size" on page 510.

`Number of disk reads`

This measurements is the number of pages read from disk into the Objectivity / DB cache. It can be used to help you adjust the size of the cache. See "Optimizing the Cache Size" on page 510.

`Number of old pages written`

This measurement is the number of existing pages written from the Objectivity / DB cache to disk. It includes pages from both the small-object and the large-object buffer pools.

`Number of new pages written`

This measurement is the number of new pages written from the Objectivity / DB cache to disk. (Each new page is a container extension.)

`Number of forced file closes`

If this measurement is nonzero, then the number of file descriptors reserved for the current Objectivity context is too low. You should consider setting a higher number of file descriptors (greater than the number of database files to be opened). You set the number of file descriptions in the *`nFiles`* parameter to `ooInit` when you initialize Objectivity / DB.

`Number of open hash calls`

This measurement is the number of scope-named objects that were opened.

`Number of hash overflows`

This measurement is the number of hash overflow pages searched in looking up scope-named objects. A number greater than zero indicates that some of the hash table has overflowed, which may mean that the initial size of a hashed container should be increased. You should try to identify the relevant container by inspecting the code, browsing the federated database, and experimenting. Once you have identified the container, you may want to create the container with a more appropriate initial size. See “Minimizing Container Growth” on page 511. You may also consider reducing your need for hashed containers, as described in “Minimizing Name Scopes” on page 516.

`Number of times OCs extended`

An *object cluster* (OC) is the physical implementation of a container. This measurement is the number of times that pages have been added to containers to accommodate a new basic object, VArray, or other data structure, such as an index or the hash table for a hashed container. It may indicate that you need to increase the initial size of your containers. See “Minimizing Container Growth” on page 511.

`Number of pages added to OCs`

This measurement is the total number of pages added to all containers.

`Number of SM objects resized`

This measurement can indicate that you are extending storage for non-inline association links or application-defined VArrays more than necessary, or that you are clustering too many objects in the same container.

- You can avoid extending storage for non-inline association links by setting as many associations as possible when you create a source object. See “Setting Associations Early” on page 512.
- VArrays must always be contiguous in memory, so extensions to them frequently involve copying the entire VArray to a new location in memory. You can avoid extending storage for a VArray by preallocating a larger amount of storage or resizing by a larger amount less frequently. See “Setting Initial Size of VArrays” on page 512.
- If this measurement is approximately equal to the total number of times your application either extends a VArray or adds an association link, then it may indicate that your clustering is too dense. If too many objects are placed on a page, then Objectivity/DB must allocate more storage on another page each time you extend a VArray or add an association to an object located on the page.

Maximizing Concurrency

Concurrency—simultaneous access to objects by multiple transactions—is often the most important dimension of performance. It hardly matters how efficiently a federated database responds to requests if lock conflicts thwart a high percentage of those requests. The actions in this section can improve the concurrency of applications that access your federated database.

Detecting a concurrency problem relies on anecdotal evidence—that is, complaints from the users of your applications about lock-not-granted errors. Since a lock-not-granted error aborts the transaction, you can get an indirect clue about concurrency problems by checking the number of aborted transactions whenever you run an application. `ooRunStatus` lists the number of transactions started, committed, checkpointed, and aborted under Object Manager Statistics:

```
Object Manager Statistics Wed Aug 11 21:21:22 2000
...
** Number of transactions started:  10
** Number of transaction commits:  6
** Number of commit and holds:  0
** Number of transaction aborts:  4
...
```

Bear in mind, however, that other errors can cause a transaction to be aborted, and some applications enable users to abort transactions voluntarily.

Avoiding Explicit Locks

You can improve concurrency by relying on implicit locking; that is, let Objectivity/C++ functions obtain access rights to resources as they are needed by your application. A function that reads an object will implicitly obtain a read lock; a function that modifies an object will implicitly obtain an update lock.

You should lock objects explicitly only if you need to ensure that a group of objects are available before starting an operation that requires access to all of them.

See Chapter 6, "Locking and Concurrency," for a thorough discussion of implicit and explicit locks.

Using MROW Transactions

A multiple readers, one writer (MROW) transaction allows multiple transactions to read basic objects and indexes in a container while another transaction is updating an object or index in that container. By comparison, standard transactions allow either multiple readers *or* one writer, but not both at the same time. For a fuller discussion of MROW transactions, see "Concurrent Access Policies" on page 113.

Trade-off: Objects that have been read by an MROW transaction may be stale, having been updated by a concurrent update transaction. In a standard transaction, application users are assured of viewing up-to-date information.

Isolating Update-Intensive Objects

Each basic object that is frequently updated should be isolated in its own container when possible. When multiple update-intensive objects occupy the same container, access to every object in the container will be held up by an update to any one of the objects.

If your applications use MROW transactions, each update-intensive object can share a container with a group of read-intensive objects, because an update will not interfere with MROW read access to the same container.

When an object inevitably matures from an update-intensive to a read-intensive state, you may also be able to use a round-robin strategy to place each new instance in a container with mature instances. For a more thorough discussion of clustering strategies, see "Assigning Basic Objects to Containers" on page 131.

Trade-off: Isolating update-intensive objects in separate containers tends to increase the number of containers that a given transaction must open, which reduces runtime speed.

Lengthening the Lock-Timeout Period

By default, a frustrated lock request fails immediately rather than waiting a few seconds and trying again. Lengthening the timeout period improves concurrency because each request will wait longer for competing transactions to finish.

You can set the timeout period for an individual transaction or you can set the default timeout period for all subsequent transactions. See “Lock Waiting” on page 120. Note that you cannot set a lock-timeout period for an MROW transaction.

Trade-off: Lengthening the lock-timeout period may make the application seem unresponsive when a lock collision occurs.

Linking Satellite Objects

Whenever you either create or remove a bidirectional association link, an update lock is requested for both the source object and the destination object of the link. When one of those objects tends to be the center of many such operations, concurrency is compromised. If you instead use a unidirectional association from the satellite source objects to the central destination object, you avoid locking the central object, because it is not changed when a new link is created.

For example, consider the association between a library (the central object) and its books (satellite objects). If the association is bidirectional, the library is locked each time a book is added to or removed from its collection. If the association is unidirectional, such that a book knows its library but a library does not know its books, the library is not changed when a new book is associated with it.

Trade-offs: Navigational flexibility is reduced, and referential integrity is not automatically maintained. For example, to find all books in a given library, you would have to scan the entire set of books to find those having the given library. For a fuller discussion of unidirectional and bidirectional associations, see “Association Directionality” on page 146.

Maximizing Runtime Speed

Runtime speed is improved by reducing disk I/O, network overhead, and CPU burdens. Generally, actions that maximize available storage space (page 515) also improve runtime speed, by reducing the size of objects that are fetched from disk and transmitted across the network, and by reducing the number of containers that are opened.

For a discussion of the performance impact of opened containers, see “Performance Considerations” on page 139.

Using an In-Process Lock Server

When a federated database is accessed by multiple applications, 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 in-process lock server runs within the process of the application that started it, enabling the application to request locks through simple function calls without having to send the requests to an external process.

For information about starting and using an in-process lock server, see Chapter 29, “In-Process Lock Server”.

Using Read-Only Databases

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 numerous read operations on the database’s contents, because the application can grant read locks and refuse update locks without consulting the lock server.

For information about making a database read-only, see “Making a Database Read-Only” on page 168.

Combining Transactions

For a given set of Objectivity / DB operations, a single long transaction is more efficient than multiple shorter transactions.

Trade-offs: Commits are deferred, locks are held longer, and aborts undo more work; see “Transaction Usage Guidelines” on page 86.

Clustering Objects That are Accessed Together

Each container that a transaction has to open incurs CPU, disk, and network overhead, so clustering a transaction’s objects in the same container is more efficient. In addition, each disk page is more likely to contain relevant objects, so fewer page reads are required to service a given transaction.

Trade-off: Concurrency suffers, especially for update-intensive objects; see “Assigning Basic Objects to Containers” on page 131.

Optimizing the Cache Size

Ideally, the Objectivity / DB cache should be just large enough to hold the active working set of data for the current Objectivity context. Smaller is usually better, but if the cache is too small, some cached objects have to be managed on disk rather than in memory, resulting in swapping activity and slower accesses.

The default initial cache size is 200 pages in each buffer pool, and the default maximum size of each buffer pool is 500 pages. You can adjust either or both of those parameters when you initialize Objectivity / DB by specifying the desired values as parameters to the `ooInit` global function. See “Initializing Objectivity / DB” on page 70.

You can call `ooRunStatus` to determine whether you need to adjust the cache size. The “number of buffers used” measurement under Storage Manager Statistics shows how many pages of cache the current Objectivity context has used:

```
...
Storage Manager Statistics Wed Aug 09 21:21:22 2000

** Page size                        =>  32768
** Number of buffers used           =>  500
...
** Number of buffers read           =>  5
** Number of disk reads             =>  33865
...
```

You may need to adjust the cache size as follows:

- If the number of buffers used is smaller than the initial cache size, reduce the initial size.
- If the number of buffers used is significantly higher than the initial cache size, Objectivity / DB is making relatively expensive calls to extend the cache as needed; in that case, increase the initial size.
- If the number of buffers used is equal to the maximum size of the cache, increase the maximum size.
- If the ratio of disk reads to buffer reads is high, the cache may be too small. You can experiment with a larger cache size and see whether this ratio is decreased.

Optimizing the Page Size

The best storage-page size for your application depends on various factors, including the average size of your basic objects and the amount of memory available. Basic objects clustered together are placed contiguously in the pages of their containers. When your application opens a basic object, Objectivity / DB reads the object's page into memory. Only when all basic objects on a page are closed can that page be swapped back to disk to make room in the cache for another page.

Look at the pattern of object access in your application and consider the following general guidelines:

- If your application randomly accesses many basic objects, you might be better off with a small page size.
- If your application spends a great deal of time on a fixed set of data, a larger page size may be better.

While developing your application with a "test" federated database, you can experiment with a number of different page sizes to determine the best size for your application; 2048, 4096, or the default 8192 bytes tend to be best for local files. For remote files, a page size of 1024 bytes works well.

You specify the storage-page size when creating the federated database with the `oonewfd` tool. You cannot change the page size later. To try a different page size, you must rebuild your federated database with the desired page size.

Page size affects your application's cache size and the size of each container in your application. Therefore, if you use a federated database with a different page size, you should also adjust the number of buffer pages in your cache and the number of initial pages in your containers accordingly. You set the number of buffer pages in the cache with a parameter to the `ooInit` function when you initialize Objectivity / DB; you set the initial number of pages in a container with a parameter to `operator new` when you create the container. If you intend to use a database's default container, you can set its initial number of pages with a parameter to the `ooDBObj` constructor when you create the database.

Minimizing Container Growth

In general, you should cluster basic objects that will be accessed together. However, if you create more basic objects in a given container than will fit, Objectivity / DB extends the size of the container to accommodate the new objects. Extending a container at runtime to accommodate more basic objects is a fairly expensive operation. Also, extending the container may make it noncontiguous on disk, leading to higher disk latency.

An additional factor comes into play for a hashed container. You must create a hashed container if the container or any basic object in it will be used as a scope

object. The container must accommodate not only the basic objects stored in it, but also the hash table used for looking up scope names. If the hash pages allocated are insufficient for the hashing required, a *hash overflow* occurs, which causes Objectivity / DB to extend both the hash table and the container. After a hash overflow, scope-name lookup becomes less efficient.

For additional information about scope objects and scope names, see “Individual Lookup in Name Scopes” on page 332.

To avoid unnecessary growth operations, consider making your containers larger at creation time. You can call `ooRunStatus` to determine whether and by how much to increase the initial size of new containers. Under Storage Manager Statistics, check the number of hash overflows, the number of times containers were extended, and the number of pages added to containers.

```
...
Storage Manager Statistics Wed Aug 09 21:21:22 2000
...
** Number of hash overflows        =>  0
** Number of times OCs extended    =>  0
** Number of Pages added to OCs    =>  0
...
```

If you find one or more hash overflows, or if the containers were extended more than a few times, consider making your containers larger when you create them. The number of pages added can help you decide how much bigger to make the containers.

Setting Associations Early

Whenever possible, establish the association links between two objects as soon as they are created. This will reserve space for the non-inline association links in the same logical page as the source object itself, making traversals and updates faster, and reducing the need to extend the storage for the source objects’s non-inline association links.

Setting Initial Size of VArrays

Resizing a VArray is a relatively slow operation. Try to set the initial size of a VArray close to its ultimate size, but avoid setting it larger than necessary so that you do not waste disk space.

Minimizing Search for Persistent Objects

Organize your federated database so that applications can find the persistent objects they need with minimum search:

- Whenever possible, find an object by following a link to it in a reference attribute or association. See Chapter 15, “Creating and Following Links”.

- If you need to operate on an individual object chosen from a particular group of objects, provide a way to look up an object in the group by some identifying key.
  - Favor name maps over scope names.
  - Minimize the number of persistent objects in the group to be searched—for example, the number of objects named in a name scope or the number of elements in a persistent collection.

  See Chapter 16, “Individual Lookup of Persistent Objects”.
- If you need to operate on entire groups of persistent objects, try to organize the objects to reduce the number of objects that need to be examined.
  - Scan containers instead of databases and databases instead of the entire federated database.
  - Create persistent collections of objects that are needed for an operation but that reside in different storage objects.

  See Chapter 17, “Group Lookup of Persistent Objects”.
- To find all containers in a database or all basic objects in a container, use the `contains` member function instead of `scan`.
- Define indexes as appropriate to speed predicate scans, using the most specific indexed class possible. This minimizes the number of indexed objects and thus reduces the time needed to search the index. See “Indexes” on page 390.
- Specify the open mode when initializing an object iterator if each target object is to be accessed in the same mode during the iteration. For example, when counting objects, use `oocNoOpen`; for reading, use `oocRead`; for updating, use `oocUpdate`. See “Object Iterators” on page 293.

Trade-off: Any name scopes, persistent collections, and indexes you create to reduce search time increase the space used by your federated database.

Using Handles and Object References Appropriately

Make sure that your application code follows the guidelines in Chapter 10, “Handles and Object References,” for working with handles and object references. In particular:

- Always pass object references and handles by reference.
- If you have an object reference to an object and you need to perform more than one operation on the object, assign the object reference to a handle and perform the operations with the handle instead of the object reference.
- Avoid repeatedly opening, closing, and then reopening the same object, which is time consuming.

Using Hot Mode

An application can open persistent objects that were created by applications running on different architectures. For example, an application running on Windows can open a persistent object that was created by an application running on one of the UNIX architectures. Similarly, an application running on one UNIX architecture can open persistent objects created on a Windows architecture or on a different UNIX architecture.

When a persistent object is opened on a different architecture, Objectivity/DB performs various cache operations that convert the object from its *disk format* (which is determined by the application that created it) to its *memory format* (which is determined by the application that is opening it). When the object is closed, it is converted back to its disk format and marked as available for swapping (although the buffer page containing the object can be swapped only if *all* objects on that page are closed). Reopening the object causes it to be reconverted to memory format.

For most applications, the cost of these conversion operations is insignificant. For example, the cost of opening an object is often completely masked by the I/O costs of reading the object into memory. However, if an application repeatedly opens, closes, and reopens objects that were created on other architectures, the cost of format conversion may become significant. To improve its performance, such an application should enable *hot mode* by calling the global function `ooSetHotMode` in each of its Objectivity contexts.

Hot mode reduces the amount of format conversion that takes place when an object is closed and reopened:

- In default mode (when hot mode is disabled), an object is converted to disk format as soon as its pin count falls to zero (see “Reference Counting With Handles” on page 212). The object must be reconverted to memory format if it is subsequently reopened.
- When hot mode is enabled, the object is converted to disk format only if the buffer page(s) containing the object are swapped out. In effect, the object is partially closed after its pin count falls to zero, when the object is marked as available for swapping, but left in memory format. As long as the object is still in memory format, it can be reopened without format conversion.

Trade-off: In hot mode, every open object uses 48 more bytes of memory than a closed object uses. Depending on the application, this additional memory usage may require the operating system to swap more frequently, possibly offsetting the performance gain from enabling hot mode. For this situation, you should try to reduce the size of the buffer pools in the Objectivity context. Fewer pages in a buffer pool means that fewer objects can be open at any time, which reduces the amount of memory used for open objects.

Updating Indexes Explicitly

An update-intensive application that uses indexes defined on a database or the federated database may be able to improve performance by updating indexes only when necessary.

The index mode `oocExplicitUpdate` gives you explicit control over when and whether indexes are updated during a transaction. See “Explicitly Updating Indexes” on page 403. If a transaction makes numerous modifications to indexed objects but never updates any key fields of the objects, there is no need to update the indexes. In that situation, using `oocExplicitUpdate` will improve performance by avoiding unnecessary updates to indexes.

Maximizing Available Space

Available space is often a secondary concern, either because the federated database is small relative to the available space, or because more storage space can be added in the form of new databases on additional storage devices. However, eliminating wasted storage space is sometimes crucial to avoid filling up available resources, and often pays a dividend in improved runtime speed as well.

Minimizing the Number of Containers

Each container has empty or partially filled pages, so the more containers you use, the more empty and partially filled pages your database will contain.

Trade-offs: Minimizing the number of containers helps improve runtime speed, but tends to reduce concurrency.

Minimizing Default Container Size

Each database has a default container, which is a hashed container. If your application does not use the default container of a particular database, set the initial number of pages in its default container to 1 when you create a new database. You can specify the number of pages in the default container as a parameter to the `ooDBObj` constructor (if you specify 0, the size defaults to 4 pages).

This guideline is important only if you create more than a few databases.

Minimizing Growth of Stable Containers

When a container's ultimate size can be estimated, set the initial size near that estimate, and consider reducing the growth percentage from the default of 10 percent. A container expands by a percentage increment rather than a fixed increment, so each successive growth operation allocates a larger chunk of disk space. If you rely on growth operations to bring a container to its ultimate size, there is a good chance that much of the space allocated by the final growth operation will be wasted.

For example, suppose a container grows to 100 pages of objects, and then grows once more to accommodate an additional object. The default growth percentage is 10 percent, so an additional 10 pages will be allocated to accommodate the new object, which might occupy only a small fraction of 1 page.

You set the growth factor when you create the container, with a parameter to `operator new`. If you intend to use a database's default container, you can set its growth factor with a parameter to the `ooDBObj` constructor when you create the database.

Minimizing Name Scopes

Name scopes can provide fast random lookup of individual persistent objects. However, they are best used only when a few objects need to be looked up. You should minimize both the number of name scopes and the size of each name scope in your application. For example, use name scopes to name and lookup a small number of name maps; use those name maps to name and lookup other persistent objects. Chapter 16, "Individual Lookup of Persistent Objects," describes name scopes, name maps, and other mechanisms for assigning unique keys to persistent objects to facilitate lookup.

A name scope is storage intensive because it requires a hashed container and because it uses more than 61 bytes for each scope name. The storage (in bytes) required for a scope name that is *nameLength* bytes long is calculated as:

`61 + (2 x nameLength)`

The hash table in a hashed container is initially 70% of the initial size of the container, measured in the number of pages. If the hash table grows beyond this size in order to accommodate more scope names, hash overflow occurs, degrading performance for looking up scope-named objects in the hash table.

Because hashed containers occupy more storage than nonhashed containers, you should create as few of them as possible. When you use a database, federated database, or autonomous partition as the scope object, the name scope uses the hashing mechanism of a (hashed) default container, which already exists. On the other hand, when you use a container or a basic object as the scope object, you must create an additional hashed container.

Deleting Basic Objects Efficiently

The manner in which you delete basic objects may affect the disk space used. The packing density of a container may be very low if most of the objects in it are deleted near the end of a transaction. If you do find it necessary to delete many objects, you should run the `ootidy` tool periodically to reduce the fragmentation in secondary storage, as described in the Objectivity/DB administration book. This tool will eliminate empty storage pages but it will not move objects off of nearly empty pages so that those pages can be deleted.

Consider putting temporary basic objects (ones that you know will be deleted) in their own container. When you no longer need the objects, delete the container, which deletes all basic objects it contains.

Trade-off: Temporary objects will not be clustered near the other basic objects, possibly resulting in more cache activity.

Simplifying Links

Whenever possible, reduce the sophistication of links from a source class to a destination class. The more sophisticated the link, the more disk space it requires. In order of decreasing sophistication, the types of links are:

- Bidirectional association (24 bytes)
- Unidirectional association (12 bytes)
- Standard object reference (8 bytes)
- Short object reference (4 bytes)

Trade-offs: Short object references require the destination object to be in the same container as the source object. Unidirectional relationships limit your navigational flexibility and do not enforce referential integrity.

Selecting Array Types

Use a VArray only when necessary. If you can predict the sizes of the arrays in a particular attribute and if the range of sizes is not great, you can save space by defining the attribute as a fixed-size array, which is inherently smaller than a VArray.

Trade-off: Fixed-size arrays are not accessible by Objectivity for Java or Objectivity/Smalltalk applications; you must use VArrays for interoperability.

Selecting String Types

If you can predict the lengths of the strings in a particular attribute and if the range of lengths is not great, it is more efficient to use an optimized string class `ooString(`*`N`*`)` instead of the variable-sized string class `ooVString`. These classes are described in Chapter 13, “Objectivity/C++ Strings”.

Trade-off: Optimized strings are not accessible by Objectivity for Java or Objectivity/Smalltalk applications; you must use the class `ooVString` for interoperability.

Creating Indexes Judiciously

Indexes provide fast and predictable search capabilities at the cost of additional disk space and memory space to maintain the index. An index duplicates a portion of the information that already resides within the indexed objects. Each index is stored in a container and can account for a large percentage of the container space usage. The space required to support more than one index in a container can be quite large, so you may want to minimize the number of different indexes over a given storage object. See “Indexes” on page 390.

25

Conforming to the ODMG Interface

Certain Objectivity / C++ types and classes conform to a subset of Release 1 or Release 2 of the Object Database Management Group (ODMG) interface. The ODMG interface is an object database standard that allows you to develop applications that can be shared with other systems that support this standard.

This chapter describes:

- The ODMG logical storage hierarchy and how its terminology differs from that of Objectivity / DB
- The Objectivity / C++ support for the ODMG interface
- The basic process for developing an Objectivity / DB application that conforms to the ODMG standard

This chapter assumes that you are familiar with the ODMG interface, and have access to the book *The Object Database Standard: ODMG 2.0*

Logical Storage Hierarchy

The ODMG standard, as reflected in the Objectivity / C++ interface, recognizes a two-level storage hierarchy:

- At the lower level, persistent data is stored in *persistent objects*, which are instances of class `d_Object` or its derived classes.
- At the higher level, persistent objects are stored in a *database*, which is an instance of class `d_Database`.

In contrast, the Objectivity / DB storage hierarchy has additional levels; at the lowest level, persistent objects are grouped into *containers*, which are grouped into *databases* (instances of `ooDBObj`), which are grouped into a *federated database* (an instance of `ooFDObj`) at the highest level.

This book uses the terms in Table 25-1 for the levels in each storage hierarchy.

Table 25-1: Objectivity/DB and ODMG Terminology

Objectivity/DB Storage Levels		ODMG Storage Levels	
Federated database	Top	ODMG database	Top
Objectivity/DB database	2nd	*(No equivalent)*	
Container	3rd	*(No equivalent)*	
Persistent object	Bottom	Persistent object	Bottom

WARNING Do not confuse an ODMG database with an Objectivity/DB database; they refer to different logical storage levels.

Objectivity/C++ Support for the ODMG Interface

Support for ODMG Classes

Certain Objectivity/C++ classes are equivalent to classes in the ODMG interface. Because of this equivalence, you can use the ODMG class names instead of the corresponding Objectivity/C++ names in your DDL files and application code files. Table 25-2 lists the equivalent Objectivity/C++ and ODMG classes.

Table 25-2: ODMG and Objectivity/C++ Class Name Equivalents

For This Objectivity/C++ Class Name	Substitute This ODMG Class Name
`ooObj`	`d_Object`
`ooRef(`*`userClass`*`)` `ooRef(ooObj)`	`d_Ref<`*`appClass`*`>` `d_Ref<d_Object>`
`ooTrans`	`d_Transaction`
`ooVString`	`d_String`
`ooVArrayT<`*`element_type`*`>`	`d_Varray<`*`element_type`*`>`

Objectivity/C++ implements the following additional ODMG-standard classes:

- `d_Database`

  Note: You use the `d_Database` class instead of `ooHandle(ooFDObj)` or `ooRef(ooFDObj)` to manipulate the highest storage level.

- d_Ref_Any
- d_Iterator
- d_Date
- d_Time
- d_Timestamp
- d_Interval

You can use Objectivity/C++ features such as containers in an ODMG application, depending on the level of ODMG compliance you choose.

NOTE As implemented by Objectivity/C++, ODMG object references (instances of class d_Ref<*userClass*>) are identical to Objectivity/C++ object references. Consequently, you can use them either as persistent links between classes or as smart pointers for accessing referenced objects in memory. For performance reasons, however, you should consider departing from strict ODMG compliance and using handles for in-memory access to persistent objects.

Support for ODMG Types

Certain Objectivity/C++ types are equivalent to types in the ODMG interface. Because of this equivalence, you can use the ODMG type names instead of the corresponding Objectivity/C++ names in your DDL files and application code files. Table 25-3 lists the correspondences between ODMG and Objectivity/C++ types.

Table 25-3: ODMG and Objectivity/C++ Type Name Equivalents

Use This ODMG Type Name	Instead of This Objectivity/C++ Type Name
d_Boolean	ooBoolean
d_Char	char
d_Double	float64
d_Float	float32
d_Long	int32
d_Octet	char
d_Short	int16
d_ULong	uint32
d_UShort	uint16

Application Development

Enabling ODMG Support

By default, Objectivity/C++ support for the ODMG application interface is disabled. To enable ODMG support (that is, to add all of the ODMG types and definitions), you must use the `-DOO_ODMG` flag when running both the DDL processor and your C++ compiler.

General Development Steps

To develop an Objectivity/DB application that uses Objectivity/C++ support for the ODMG interfaces, you:

1. Create an ODMG database using the `oonewfd` tool (see the Objectivity/DB administration book).
2. Design your database schema and create one or more DDL files containing declarations for all persistence-capable classes used in your application. You can use the standard Data Definition Language, substituting the ODMG class and type names wherever possible.
3. If you use the ODMG time and date classes, you must include the `ooTime.h` header file in any DDL files and source code files that use these classes.
4. Process your DDL files using the DDL processor, specifying the `-DOO_ODMG` option. For information about the DDL processor, see the Objectivity/DDL book.
5. Write the source code for your application using the ODMG classes and types supported Objectivity/C++; include the DDL-generated primary header files wherever necessary.
6. Set up a default Objectivity/DB database:
   a. Use the `oonewdb` tool to create the Objectivity/DB database (see the Objectivity/DB administration book).
   b. Set the environment variable `OO_DB_NAME` to be the system name of the Objectivity/DB database you just created.

   The persistent objects you create are stored in the default container of the default Objectivity/DB database. If you do not set the `OO_DB_NAME` environment variable, an Objectivity/DB database with the system name `default_odmg_db` is automatically created and used.
7. Specifying the `-DOO_ODMG` flag, compile your C++ application source code files and the method-implementation files generated by the DDL processor.
8. Link the compiled files with Objectivity/DB libraries.

Example ODMG Application

This section presents a simple example that uses the Objectivity / C++ support for the ODMG interface.

In this example, two persistence-capable classes—`Person` and `Phone`—create and manage a telephone list. In the DDL file, class `Person` has object references to `Phone` for home and work telephone numbers. `Person` also has a reference to itself for spouse information. `Phone` has a reference to `Person` for the owner of the telephone number.

The application code stores a new `Person` object and `Phone` object if a person's name and telephone numbers are given as command-line parameters. The person's last name is used as its scope name in the federated database (which implies that all `Person` objects must have unique last names). If the only command-line parameter is a name string, the application finds the person with that last name and prints information about the person, including telephone numbers.

The application assumes:

- An ODMG database (Objectivity / DB federated database) has been created with a boot file named `myODMGdb`.
- The DDL processor is run with the `-DOO_ODMG` option to process the `phoneList.ddl` file; this step generates the header file `phoneList.h`.
- An Objectivity / DB database has been created and its name is set in the environment variable `OO_DB_NAME`.

```
// DDL file phoneList.ddl

class Phone;

class Person : public d_Object {
public:
   d_String firstName;
   d_String lastName;
   d_Ref<Person> spouse;
   d_Ref<Phone> home;
   d_Ref<Phone> work;

   Person(const char *argArray[]);
   void print() const;
};
```

```
class Phone : public d_Object {
public:
   d_ULong number;
   d_Ref<Person> owner;

   Phone(d_ULong theNumber, d_Ref<Person> theOwner);
   void print(const char *label = "phone") const;
};
```

// Application code file phoneList.cxx

```
#include "phoneList.h"
#include <stdlib.h>
#include <iostream.h>

// Constructor for Person
Person::Person(const char *argArray[]) :
   firstName(argArray[2]),
   lastName(argArray[1]),
   home(new(this) Phone(atol(argArray[3]), this)),
   work(new(this) Phone(atol(argArray[4]), this))
{ }

// print member function for Person
void Person::print() const
{
   cout << "name: " << this->lastName << ", " <<
      this->firstName << endl;
   this->home->print("home");
   this->work->print("work");
   if (this->spouse) {
      cout << "(Married.)" << endl;
   }
}

// Constructor for Phone
Phone::Phone(d_ULong theNumber, d_Ref<Person> theOwner) :
   number(theNumber),
   owner(theOwner)
{ }

// print member function for Phone
void Phone::print(const char *label) const
{
   cout << label << ": " << (this->number) / 10000
      << "-" << (this->number) % 10000 << endl;
}
```

```
// Global createPerson function
static void createPerson(const char *argArray[])
{
   d_Database odmgDB;              // ODMG database
   d_Transaction trans;

   trans.begin();
   odmgDB.open("myODMGdb");
   d_Ref<Person> person = new(&odmgDB) Person(argArray);

   odmgDB.set_object_name(person, person->lastName);
   trans.commit();
   odmgDB.close();
} // End createPerson

// Global displayPerson function
static void displayPerson(const char *lastName)
{
   d_Database odmgDB;              // ODMG database
   d_Transaction trans;

   trans.begin();
   odmgDB.open("myODMGdb", d_Database::read_only);

   d_Ref<Person> person = odmgDB.lookup_object(lastName);
   if (person != 0) {
      person->print();
   }
   trans.commit();
   odmgDB.close();
} // End displayPerson

// Application main program
int main(unsigned int numOfArgs, const char *argArray[])
{
   if (ooInit()) {
      if (numOfArgs > 2) {
         createPerson(argArray);
      } else {
         displayPerson(argArray[1]);
      }
      return 0;
   }
```

```
   else {
      cerr << "Unable to initialize Objectivity/DB" << endl;
      return 1;
   }
} // End main
```

26

Writing Administration Tools

This chapter provides information about performing administration tasks with the Objectivity/C++ programming interface. You typically create your own administration tools to perform these tasks.

If an administrative task can be performed through an Objectivity/DB tool as an alternative to using the programming interface, the task description identifies that tool. The Objectivity/DB tools are described in the Objectivity/DB administration book.

This chapter describes how to:

- Perform administration tasks for the federated database and for an individual database
- Create a recovery application

Federated Database Administration

Member functions of a federated-database handle allow you to get information about the federated database, change various attributes, and consolidate fragmented storage.

Getting Information About a Federated Database

Tool alternative: oochange with the boot filename and no other options

You can obtain a federated database's individual attributes by calling the following member functions on a handle to the federated database. Names are returned as strings and numbers are returned as integers:

- The `lockServerName` member function gets the name of the host running the lock server for the federated database.
- The `number` member function gets the identifier of the federated database.
- The `name` member function gets the system name of the federated database.

- The pageSize member function gets the storage page size of the federated database.

You can list all the files in a federated database by using the dumpCatalog member function on a handle to the federated database. By default, the information is printed to the standard output. You can specify parameters to direct the output to a file and change the format of the list.

EXAMPLE This example lists the federated database's files, printing the list to a file called `catalog.txt` and then to the standard output device.

```
// Application code file
#include <oo.h>
...
ooHandle(ooFDObj) fdH;
FILE* fp;

fp = fopen("catalog.txt", "w");

// Print filenames to the file catalog.txt
fdH.dumpCatalog(fp);

// Print the filenames to the standard output device
fdH.dumpCatalog();
```

Changing Federated Database Attributes

Tool alternative: `oochange`

You can write a special-purpose application to change certain attributes of the federated database. This application must consist of a single update transaction in which you call the change member function on a handle to the federated database. The parameters to this function specify:

- The new pathname for the boot file
- The name of the new lock server host
- The new federated-database identifier

You can specify the value 0 (the default) for a parameter to leave the corresponding attribute unchanged. An optional parameter allows you to report the changes to a transcript file.

You cannot change the system name of the federated database or its storage-page size. Note that if you change the location of the boot file, the updated boot file is written to the new location, but the old boot file remains. You must delete the old boot file manually.

Your special-purpose application must exit immediately after the transaction commits. This is because the new state of the federated database is inconsistent with information cached by the executing application.

When you run your application, you must guarantee that no other process has access to the federated database or data corruption could result. For example, you could stop the lock server before running the application, call the `ooNoLock` global function in the application to make it run in single-user mode, then restart the lock server after the application completes.

EXAMPLE This application changes the lock server host for the federated database to the host `moon`.

```
// Application code file
#include <oo.h>

int performChanges() {
   ooTrans trans;
   ooHandle(ooFDObj) fdH;
   ooStatus status;
   trans.start();
   if (!fdH.open("Documentation", oocUpdate)) {
      cerr << "Cannot update federated database" << endl;
      trans.abort();
      return 1;
   }
   // Change the lock server host
   status = fdH.change(0,         // Don't change boot file path
                       "moon");  // New lock server host
   if (status)) {
      trans.commit();
      return 0;
   }
   else {
      cerr << "Cannot change lock server host" << endl;
      trans.abort();
      return 1;
   }
} // End performChanges
```

```
int main() {
   int retval = 0;
   if (!ooInit() ) {
      cerr << "Unable to initialize Objectivity/DB" << endl;
      return 1;
   }
   // Run in single-user mode
   if (!ooNoLock()) {
      cerr << "Unable to run in single-user mode" << endl;
      return 1;
   }

   retval = performChanges();  // Call function to change FD
   return retval;
} // End main
```

To run this program, you:

1. Shut down the federated database's current lock server.
2. Execute the program.
3. Restart the old lock server if any other federated databases use it.
4. Start (if necessary) the lock server on the host `moon`.

Tidying a Federated Database

Tool alternative: `ootidy`

You can consolidate data that has become fragmented over time. To do so, you call the `tidy` member function on a handle to the federated database. The tidy operation:

- Requests an exclusive update lock on each database in the federation. If a particular database cannot be locked, the operation skips it and continues to the next database.
- Requires free disk space equal to the size of the largest database in the federation (to create a temporary database during execution).

You should call `tidy` in a single-purpose update transaction. That is, you must not manipulate any database, container, or basic object before calling `tidy` in the same transaction, and you must commit the transaction immediately after `tidy` completes. This is because compacting and relocating physical storage renders the databases inconsistent with any data that was cached during the transaction, and committing the transaction discards the obsolete cached data.

EXAMPLE This example uses a single-purpose transaction to tidy the federated database.

```
// Application code file
#include <oo.h>
...
ooTrans trans;
ooHandle(ooFDObj) fdH;
ooStatus status;
...
trans.start();
status = fdH.open("Documentation", oocUpdate);
if (status == oocSuccess) {
   status = fdH.tidy();        // Tidy the federated database
   if (status == oocSuccess) {
      trans.commit();
   }
}
if (status == oocError)
   trans.abort();
```

Database Administration

Member functions of a database handle allow you to get information about the referenced database, move the database file, replace the database, and consolidate fragmented storage.

Getting Information About a Database

Tool alternative: `oochangedb` with the boot filename, the `-db` or `-id` option, and no other options

You can obtain information about a database's attributes by calling the following member functions on a handle to the database. Each of these member functions returns a string:

- The `name` member function gets the database's system name.
- The `hostName` member function gets the name of the network host where the database file is located.
- The `pathName` member function gets the full pathname of the directory containing the database file.
- The `fileName` member function gets the fully qualified path and filename of the database file.

EXAMPLE This example prints host, path, and filename information about a database.

```
// Application code file
#include <oo.h>
...
ooHandle(ooDBObj) dbH;
...        // Set dbH to reference the database
// Get and print information about the database file
printf("hostname: %s\n", dbH.hostName());
printf("pathname: %s\n", dbH.pathName());
printf("filename: %s\n", dbH.fileName());
```

Moving a Database File

Tool alternative: oochangedb

You can rename or relocate a database file on a network. To do so, you:

1. Programmatically change the database's host, pathname, or filename in the federated-database catalog by calling the change member function on a handle to the database.
2. Physically move or rename the database file on your file system by executing the appropriate operating-system command.

The parameters to the change function specify:

- The new system name for the database. *This feature is currently not implemented.* Always pass the value of 0 for this parameter.
- The name of the data server host on which the database is to reside.
- The new pathname for the database file.

You can specify the value 0 (the default) for a parameter to leave the corresponding attribute unchanged. An optional parameter allows you to report the changes to a transcript file.

EXAMPLE This example changes the host and pathname of a database file.

```
// Application code file
#include <oo.h>
...
ooHandle(ooDBObj) dbH;
...        // Set dbH to reference a database

// Change database host and pathname
dbH.change(0,                   // Don't change system name
           "myHost",                          // New host
           "/mnt/john/design/adder.ecad.DB");  // New pathname
```

A database can also be moved with the `oochangedb` administration tool (see the Objectivity / DB administration book).

Tidying a Database

Tool alternative: `ootidy` with the `-db` option

You can consolidate the data in a database that has become fragmented over time. To do so, you call the `tidy` member function on a handle to that database. The tidy operation:

- Must be able to obtain an exclusive update lock on the database.
- Requires free disk space equal to the size of the database you are tidying (to create a temporary database during execution).

You should call `tidy` in a single-purpose update transaction. That is, you must not manipulate any database, container, or basic object before calling `tidy` in the same transaction, and you must commit the transaction immediately after `tidy` completes. This is because compacting and relocating physical storage renders the database inconsistent with any system data that was cached during the transaction, and committing the transaction discards the obsolete cached data.

EXAMPLE This example uses a single-purpose transaction to tidy a database.

```
// Application code file
#include <oo.h>
...
ooTrans trans;
ooHandle(ooFDObj) fdH;
ooHandle(ooDBObj) dbH;
trans.start();
fdH.open("Documentation", oocUpdate);
dbH.open(fdH, "Introduction", oocUpdate);
dbH.tidy();
trans.commit();
```

Replacing a Database

You can replace an existing database with a new database by using the ooReplace global macro. This macro:

1. Deletes any existing database with the specified system name.
2. Creates a new database with the same system name.
3. Returns a handle to the new database.

This macro is generally used only to clean up a federated database between test runs.

Creating a Recovery Application

You can create your own general recovery tool by writing an Objectivity/C++ database application that opens a federated database with automatic recovery enabled. You enable automatic recovery by invoking the open member function on a federated-database handle with the *recover* parameter set to oocTrue. For performance reasons, you should do this only one time per application.

Alternatively, you can create a special-purpose application that calls one or more of the administrative recovery functions:

- The ooGetActiveTrans global function gets information about all active transactions against a federated database.
- The ooGetResourceOwners global function gets information about the transactions for which a specific transaction is waiting.
- The ooCleanup global function recovers a specific transaction.

When you write a recovery application that calls the administrative recovery functions:

- Include the header file `ooRecover.h` in the application code. This file contains declarations for the administrative recovery functions, as well as the types that these functions use.
- Do not call `ooInit` within the application.
- Do not call any nonrecovery Objectivity / DB functions within the application. If you need to call a nonrecovery function, create a separate execution environment using a `system` call.
- Link your application with the Objectivity / DB library.
  - On UNIX, you must additionally link with the Objectivity / DB administration library *before* linking with other Objectivity / DB libraries. See *Installation and Platform Notes for UNIX*.
  - On Windows, the administration interfaces are in the standard Objectivity / DB library.

Getting Information About Transactions

Every transaction has an *identifier* of type `ooTransId` that uniquely identifies it to the lock server. Recovery functions use parameters of this type to identify a transaction of interest.

Recovery functions return information about a transaction in a structure of type `ooTransInfo`. The `tid` member of a transaction-information structure is the transaction identifier. If a transaction is waiting for a lock on an Objectivity / DB object (typically a container), a resource-information structure of type `ooResource` identifies the object and its lock status. For more information about transaction-information structures and resource-information structures, see the definitions of `ooTransInfo` and `ooResource` in the `ooRecover.h` header file.

The `ooGetActiveTrans` function sets a pointer to point to an array of transaction-information structures, one for each active transaction. Parameters to the function allow you to request that the array contain information about only those transactions started on a particular host or only those transactions started by a particular user.

The `ooGetResourceOwners` function sets a pointer to point to a resource-information structure describing the resource for which a particular transaction is waiting. Its also sets a pointer to point to an array of transaction-information structures, one for each transaction that holds a lock on that resource. This array may contain information about a single transaction with a read or update lock on the resource, or it may contain information about one or more transactions with MROW read locks and possibly a transaction with an update lock.

The arrays of transaction-information structures created by these two functions are terminated by a structure in which the transaction identifier is `oocInValidTransId`.

Recovering a Transaction

The `ooCleanup` function recovers a specified transaction if it is inactive, rolling back uncommitted changes to restore the federated database to its logical state before the transaction was started. This function puts a *recovery lock* on the specified transaction, which can be used to determine whether multiple processes are attempting to recover the same transaction simultaneously.

Parameters allow you to specify:

- Whether to permit the recovery of a transaction that started on another host or to recover only if the transaction started on the same host from which `ooCleanup` was invoked.
- Whether to contact the lock server to release any locks left by the transaction or to run when the lock server isn't running.
- What to do if another cleanup process holds the recovery lock on the transaction.
- A transaction-information structure in which to return information about any competing cleanup process that holds the recovery lock on the transaction.

EXAMPLE This example gets a list of all active transactions and recovers them. It does not check to see if a given user is active, and therefore may clean up a currently running transaction. Because of this, you must be sure that the federated database is not currently being used before running this application.

```
#include <stdio.h>
#include <ooRecover.h>

void main()
{
   char *bootfilepath = "EXAMPLE";  // Path to the boot file

   ooTransInfo *activeTrans;  // Pointer to be set to array
                              // of transaction-info structures
   ooStatus stat;
```

```
   // Get the array describing the active transactions
   ooGetActiveTrans(
      &activeTrans,   // Pointer to be set to point to array
      &bootfilepath,  // Pointer to pathname of boot file
      0,              // Include transactions on all hosts
      0);             // Include transactions by all users

   // Recover each active transaction
   while(activeTrans->tid != oocInValidTransId) {
      // Cleanup up the transaction
      stat = ooCleanup(
         &bootfilepath,  // Pointer to pathname of boot file
         activeTrans->tid,  // ID of transaction to recover
         1,        // Clean up transaction from any host
         0,        // Use the lock server
         0,        // Fail if another cleanup process owns lock
         0);       // Do not return information on competing
                   // cleanup process

      // Check whether cleanup worked
      if (stat)
         printf("Cleaned up transaction ID %d \n",
            activeTrans->tid);
      else
         printf("Could NOT cleanup transaction ID %d \n",
            activeTrans->tid);

      activeTrans++;
   } // End while more transactions
} // End main
```

27

Autonomous Partitions

You can support widely distributed database environments by dividing a federated database into independent pieces, called *autonomous partitions*. You can create and administer autonomous partitions only if you have purchased and installed Objectivity / DB Fault Tolerant Option (Objectivity / FTO). Unless otherwise indicated, these tasks are also valid when you use Objectivity / DB Data Replication Option (Objectivity / DRO) along with Objectivity / FTO.

This chapter describes:

- General information about autonomous partitions
- The Objectivity / FTO extensions to the Objectivity / C++ programming interface
- Tasks that involve partitions, such as creating, finding, deleting, and purging
- Troubleshooting and recovery

NOTE You should use the information in this chapter only after reading Chapter 1, "Fault Tolerant Option," and Chapter 3, "Working with Autonomous Partitions," in the Objectivity / FTO and Objectivity / DRO book.

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.

Physically, an autonomous partition is maintained in a system-database file which stores the 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, Objectivity / DB needs access to all partitions in the federated database so that it can update each partition's global catalog. Each autonomous partition is listed in the global catalogs by its system name. The system name must be unique within the federated database. When an autonomous partition is deleted, its system name is also removed from the global catalogs.

Managing Partitions From an Application

Objectivity / FTO extends the Objectivity / C++ programming interface, providing classes and global functions that allow you to:

- Open a federated database with a particular autonomous partition as the boot autonomous partition.
- Allow your application to access offline partitions.
- Create a new autonomous partition.
- Test whether an autonomous partition exists.
- Find an existing autonomous partition.
- Get or change attributes of an autonomous partition.
- Get or change the databases and containers controlled by an autonomous partition.
- Delete a partition.
- Purge partitions that have become permanently inaccessible.

Some tasks can be performed only from an application; other tasks can be performed through an Objectivity / DB or an Objectivity / FTO tool as an alternative to using the programming interface.

NOTE In this chapter, 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, the task description identifies that tool.

Using a Handle to a Partition

As is the case for all Objectivity / DB objects, you work with an autonomous partition through a handle. For example, you perform many Objectivity / FTO tasks by calling a member function on a handle to a partition.

Unlike handles to other kinds of Objectivity / DB objects, however, the usage of partition handles is somewhat limited. An autonomous partition is neither a storage object nor a persistent object, so it cannot be used in operations that apply only to storage objects, only to persistent objects, or only to basic objects. In particular, an autonomous-partition handle *cannot* be a value for parameters that specify:

- The storage object used as a clustering directive
- The destination object of an association, because associations link persistent objects together
- The storage object to be scanned to initialize an object iterator

However, an autonomous-partition handle *can* be used to specify the scope object for a scope name, because persistent objects can be named in the scope of an autonomous partition. See “Scope Objects” on page 333.

Linking With Objectivity/FTO

For complete information about compiling and linking Objectivity / DB applications, see *Installation and Platform Notes* for your platform.

Windows

Objectivity / DB applications rely on an option-enabling DLL to enable appropriate features in the Objectivity / DB release and debug libraries. When you run an application that uses Objectivity / FTO features, you must ensure that the correct version of the option-enabling DLL is available. The correct version of the DLL is placed in the Objectivity / DB installation directory when you install Objectivity / FTO.

UNIX

You must add the object module `ooPart.o` in your link line before `liboo.a` or any other Objectivity / DB libraries. If you are also using Objectivity / DRO, you should link with `ooRepl.o` instead of `ooPart.o`, because `ooRepl.o` is a superset of `ooPart.o`.

Running an Objectivity/FTO Application or Tool

Within an Objectivity / FTO application or tool, any operation that creates, deletes, or modifies the various files used by an autonomous partition requires the normal file-system permissions:

- If a tool performs the operation, the user account under which the tool is run must have the appropriate permissions.
- If an application performs the operation:
  - If the files are being accessed remotely by the Advanced Multithreaded Server (AMS), the user account under which AMS is running needs the appropriate permissions.
  - Otherwise, the user account running the application needs the appropriate permissions.

  For information about AMS, see the Objectivity / DB administration book.

Specifying the Boot Autonomous Partition

Must have access to: The desired boot partition

An Objectivity / DB application opens a partitioned federated database using the boot file of a particular autonomous partition, called the *boot autonomous partition*. To open 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 as the parameter when you call `open` on a federated-database handle. You can optionally check whether an autonomous partition exists before attempting to use its boot file; see "Checking Whether an Autonomous Partition Exists" on page 544.

Controlling Access to Offline Partitions

A partition's *offline status* controls whether it may be accessed. By default, applications enforce the offline status of partitions; this means that an application can access data in an offline autonomous partition only if that partition is the boot autonomous partition.

If you want to be able to access offline partitions other than your application's boot autonomous partition, call the global function `ooSetOfflineMode` to set the offline mode for your application. Specify the offline mode `oocIgnore` to ignore the offline status of partitions.

If you later want to return to enforcing the offline status of partitions, call `ooSetOfflineMode` again, specifying the offline mode `oocEnforce`.

To check whether the application is enforcing or ignoring the offline status of partitions, call the global function `ooGetOfflineMode`.

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.

Each newly created autonomous partition is assigned an integer identifier that is unique within the federated database. This identifier cannot be changed. The first partition, which is created implicitly, is given the reserved identifier 65535; partitions that you create are given sequential identifiers starting with 1.

To create an autonomous partition, use `operator new` and the constructor for the class `ooAPObj`.

The constructor creates:

- A system-database file
- A journal directory
- A boot file
- An autonomous partition in the federated database
- An instance of `ooAPObj` in your application

The constructor requires you to specify the system name of the partition, the name of the lock server host for the partition, and the data server host and directory path where the partition's system-database file is to be located. Optional parameters to the constructor allow you to specify the host and directory path for the boot file and for journal files. By default, the boot file and journal files are created in the same directory, on the same host as the system-database file.

You must assign the result of `operator new` directly to an autonomous-partition handle.

When you commit or checkpoint the transaction in which you add the partition, the partition is added to the global catalogs of all partitions, making it visible to other applications. If you instead abort the transaction, the global catalogs are not updated and all files created for the partition are deleted.

EXAMPLE This example creates an autonomous partition named AP1 in the open federated database. This code explicitly sets all attributes of the autonomous partition except for the host and path to the journal directory. By default, these are set to the host and directory path of the partition's system-database file.

```
// Application code file
#include <oo.h>
...
// Create the partition
ooHandle(ooAPObj) apH = new ooAPObj(
   "AP1",                    // System name
   "borg",                   // Lock server host
   "sys55",                  // System-database file host
   "/mnt/AP1/AP1.AP",        // System-database file path
   "sys55",                  // Boot file host
   "/mnt/AP1/objectivityFd" // Boot file path
);

// Test that the partition was created
if (apH == 0) {
   ...    // Error
}
```

Checking Whether an Autonomous Partition Exists

Must have access to: The desired autonomous partition

You can test whether a particular autonomous partition exists by calling the exist member function on an autonomous-partition handle. The parameters to exist specify a handle to the currently open federated database, the system name of the desired partition, and an optional open mode. If the specified partition is found, this member function returns ooTrue and sets the autonomous-partition handle to reference the specified database; if not, it returns ooFalse.

Testing for a partition's existence helps to avoid the errors that are signaled if you attempt to create a partition with a nonunique system name or if you attempt to find a nonexistent partition.

Finding an Autonomous Partition

Must have access to: The partition(s) to be found

You can find an existing partition from the federated database that contains the partition, from a database that the partition contains, or from a container that the partition controls.

- You can find the boot autonomous partition by calling `bootAP` on a handle to the federated database.
- You can look up an autonomous partition by its system name by calling the `open` member function on an autonomous-partition handle.
- You can initialize an object iterator of class of `ooItr(ooAPObj)` to find all autonomous partitions in the federated database in either of the following ways:
  - Call the `contains` member function on a handle to the federated database, passing the object iterator as a parameter.
  - Call the `scan` member function on the object iterator, passing a handle to the federated database as a parameter.

  After initializing the object iterator, you advance it through the iteration set by calling the iterator's `next` member function. See "Object Iterators" on page 293 for information about working with an object iterator.
- If a single image of a particular database exists, you can find the autonomous partition that contains the database by calling the `containingPartition` member function on a handle to the database.
- You can find the autonomous partition that controls a particular container by calling `controlledBy` on a handle to the container.

Opening an Autonomous Partition

Some of the member functions that find autonomous partitions have an optional open mode that specifies whether to open the found partition. Opening an autonomous partition locates and opens the partition's system-database file. Any number of transactions can concurrently open the same partition for either read or update access.

It is normally not necessary to open partitions explicitly, because they are usually opened automatically by operations that access them or their contents. For example, once you have a handle to a partition, creating a database in it automatically opens it for update. In general, you open a referenced partition explicitly only when you want to guarantee access to the partition in advance—for example, before starting a complex operation.

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 and path
- Boot file host and path
- Journal directory host and path
- Offline status (set to online by default)

You can get any of these attributes and you can change all except the system name.

Getting the Attributes of a Partition

Must have access to: The partition of interest

Tool alternative: `oochange` with the boot filename, the `-ap` or `-id` option, and no other options (see the Objectivity/DB administration book)

You can call the following member functions on an autonomous-partition handle to get the attributes of the referenced autonomous partition.

Member Function	Gets Attribute
`name`	System name
`lockServerHost`	Lock server host
`sysDBFileHost`	Host for system-database file
`sysDBFilePath`	Path for directory of system-database file
`bootFileHost`	Host for boot file
`bootFilePath`	Path for directory of boot file
`jnlDirHost`	Host for journal directory
`jnlDirPath`	Path for journal directory
`isOffline`	Offline status

EXAMPLE This example finds the autonomous partition whose system name is `VehiclePartition` and prints the host and pathname of its boot file and the host of its lock server.

```
// Application code file
#include <oo.h>
...
ooTrans transaction;
ooHandle(ooFDObj) fdH;
ooHandle(ooAPObj) apH;

// Start a transaction and open the federated database
transaction.start();
fdH.open("myFD", oocUpdate);

// Find the partition named "VehiclePartition"
if (!apH.open(fdH, "VehiclePartition")) {
   cerr << "Can't find VehiclePartition" << endl;
   trans.abort();
}

// Print the partition's boot file host and path,
// and lock server host
cout << "Boot file host: " << apH.bootFileHost() << endl;
cout << "Boot file pathname: " << apH.bootFilePath() << endl;
cout << "Lock server host: " << apH.lockServerHost() << endl;
trans.commit();
```

Changing the Host and Path Attributes

Must have access to: All autonomous partitions

Tool alternative: `oochange` with the `-ap` or `-id` option (see the Objectivity / DB administration book)

Call the change member function on an autonomous-partition handle to change any of the following host and file attributes of the referenced autonomous partition:

- Lock server host
- System-database file host and path
- Boot file host and path
- Journal directory host and path

EXAMPLE This example changes the host and path attributes of an autonomous partition.

```
// Application code file
#include <oo.h>
...
ooHandle(ooAPObj) apH;
...     // Set apH to reference the desired partition

// Change the partition's attributes
ooStatus rc = apH.change(
      "borg",                     // Lock server host
      "sys05",                    // System DB file host
      "/tmp/devel/develTest.AP",  // System DB file path
      "sys05",                    // Boot file host
      "/tmp/devel/objectFd"       // Boot file path
      "mach44",                   // Journal directory host
      "/tmp/devel/develJnl")      // Journal directory path
if (!rc) {
   ...     // Error
}
```

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 make a partition offline, call the `markOffline` member function on a handle to the partition.
- To make a partition online, call the `markOnline` member function on a handle to the partition.

Finding 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 member function 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

You can use the Objectivity/C++ programming interface to:

- Create a database in a specified partition.
- Move an existing database from one partition to another.
- Find all databases in a partition.

Creating a Database in a Partition

Must have access to: All autonomous partitions

By default, a database is created in the federated database's initial partition. If the federated database contains multiple partitions, you can create a database in a particular partition by using the `operator new` variant that takes a handle to an autonomous partition as a parameter.

EXAMPLE This example creates a database in an autonomous partition. The database is created with the system name `DB1` and default values for other constructor parameters. The database file is located in the same directory as the autonomous partition's system-database file.

```
// Application code file
#include <oo.h>
...
ooHandle(ooFDObj) fdH;
ooHandle(ooAPObj) apH;
...     // Set fdH to reference the federated database
...     // Set apH to reference the desired partition

// Create database named DB1 in partition referenced by apH
ooHandle(ooDBObj) dbH = new(fdH, apH) ooDBObj("DB1");
if (dbH == 0) {
   ...     // Error
}
```

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` member function on a handle to the database. The parameter to this function is a handle to the destination autonomous partition—that is, the partition to which the database is to be moved. This member function creates a new image of the database in the destination 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 error occurs.

This member function 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` options.

EXAMPLE This example moves a database to a different autonomous partition.

```
// Application code file
#include <oo.h>
...
ooHandle(ooAPObj) apH;
ooHandle(ooDBObj) dbH;
...     // Set apH to reference the desired partition
...     // Set dbH to reference the desired database in the
        // federated database's initial autonomous partition

// Move database to the partition referenced by apH
if (!dbH.changePartition(apH)) {
   ...     // Error
}
```

Finding Databases in a Partition

Must have access to: The partition of interest

You can initialize an object iterator of class `ooItr(ooDBObj)` to find all databases in a particular autonomous partition. To do this, you call the `imagesContainedIn` member function on a handle to the partition, passing the object iterator as a parameter.

After initializing the object iterator, you advance it through the iteration set by calling the iterator's `next` member function. See "Object Iterators" on page 293 for information about working with an object iterator.

Controlled Containers

A newly created container is controlled by its database's containing partition. You can use the Objectivity / C++ programming interface 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.
- Find 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 `transferControl` on a handle to the container. The parameter to this function is a handle to the destination autonomous partition—that is, the partition to which control is to be transferred. 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 `returnControl` on a handle to 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 `returnAll` on a handle to the partition.

Finding Containers Controlled by a Partition

Must have access to: The partition of interest

You can initialize an object iterator of class `ooItr(ooContObj)` to find all containers controlled by a particular autonomous partition. To do this, you call the `containersControlledBy` member function on a handle to the partition, passing the object iterator as a parameter.

After initializing the object iterator, you advance it through the iteration set by calling the iterator's `next` member function. See "Object Iterators" on page 293 for information about working with an object iterator.

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 the `ooDelete` global function.

Deleting an autonomous partition:

- Clears all previously transferred containers from the autonomous partition, implicitly calling `returnAll` on a handle to the partition.
- Deletes the system database and boot file for the autonomous partition from the file system. **Warning:** This operation cannot be undone by aborting the transaction.
- Deletes the database files for all database images in the autonomous partition from the file system.

  If the autonomous partition controls the last or only image of any database, `ooDelete` signals an error and the autonomous partition is left unchanged. (You can use the `oodeleteap` tool with the `-deleteLastImage` option to delete a partition that contains the only image of a database.)

If there are any outstanding journal files for the autonomous partition, these must be cleaned up before the autonomous partition can be deleted. You can clean up the journal files using the `oocleanup` administrative tool. See the Objectivity / DB administration book for a discussion of journal files and a description of the administrative tools.

Purging Autonomous Partitions

Must have access to: All autonomous partitions that are *not* being purged

Tool alternative: `oopurgeaps` (see the Objectivity / FTO and Objectivity / DRO book)

If some autonomous partitions become permanently unavailable, you can remove them from the federated database by *purging* them. For example, if a data server host machine that holds one or more partitions is in a building that has been destroyed by an earthquake, you can purge those partitions from the federated database. Purging a partition is similar to deleting it; the difference is that you can purge a partition that is not accessible, but you cannot delete an inaccessible partition (you would not be able to delete a partition whose host machine was destroyed in an earthquake).

To purge unavailable autonomous partitions, call the ooPurgeAps global function.

The following actions are taken for each database that has at least one image in the partitions being purged:

- If a quorum of images for the database still exists after removal of all the unavailable partitions, then any catalog reference to the images and / or tie-breakers in those partitions is deleted.
- If a quorum of images is no longer available, then the database is deleted completely from the federated-database catalog. The database files, however, are not deleted.

Troubleshooting and Recovery

You can create special-purpose applications to perform recovery tasks that accommodate autonomous partitions. Such applications call the ooGetActiveTrans, ooGetResourceOwners, and ooCleanup global functions. Their interactions with partitions are described here:

- Call the ooGetActiveTrans function to get information about all active transactions against a federated database. The information returned by ooGetActiveTrans is limited to the active transactions that are managed by the lock server of the autonomous partition whose boot file path is passed as the *ppBootFilePath* parameter.
- Call the ooGetResourceOwners function to get information about the transactions for which a specific transaction is waiting. The information returned is limited to the active resources (typically containers) that are managed by the lock server of the autonomous partition whose boot file path is passed as the *ppBootFilePath* parameter.
- Call the ooCleanup function to recover a specific transaction. The *ppBootFilePath* parameter passed to ooCleanup indicates *any* autonomous partition involved in the transaction. ooCleanup recovers all affected objects, even if the transaction involves multiple autonomous partitions, but autonomous partitions in the transaction must be available for the transaction to be recovered.

For more information about recovery applications, see “Creating a Recovery Application” on page 534.

28

Database Images

In a partitioned federated database, you can create and manage *database images* to replicate data across multiple autonomous partitions. You can perform database replication tasks only if you have purchased and installed Objectivity / DB Data Replication Option (Objectivity / DRO) in addition to Objectivity / DB Fault Tolerant Option (Objectivity / FTO).

This chapter describes:

- General information about database images
- The Objectivity / DRO extensions to the Objectivity / C++ programming interface
- Tasks that involve database images, such as creating, managing a tie-breaker partition, and deleting
- Installing two-machine handler functions
- Resynchronizing database images

NOTE You should use the information in this chapter only after reading Chapter 2, "Data Replication Option," and Chapter 4, "Working With Database Images,"in the Objectivity / FTO and Objectivity / DRO book.

Understanding Database Images

Objectivity / DRO enables you to create and manage multiple replicas of a database, called *database images*. Each image of a database contains all the data in that database. Location and order of creation do not distinguish an image in any way.

Each image is controlled by a single autonomous partition; each partition can control at most one image of any given database. 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 a different available image.

All images of a database share the same system name and database identifier, and each image is controlled by a different autonomous partition. Each image has a *weight*, which is used to determine whether a *quorum* of replicated 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 168). 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 attributes of an individual image.

Managing Database Images from an Application

Objectivity / DRO extends the Objectivity / C++ programming interface, providing classes and global functions that allow you to:

- Create a new image of a database.
- Get or change attributes of a database image.
- Check the number and availability of images of a database.
- Get or set the tie-breaker partition for a database.
- Find all partitions that contain an image of a particular database.
- Delete a database image.
- Allow a transaction to read from a replicated database even if a quorum of its images is not available.
- Install two-machine handler functions in applications at a site with a hot-failover configuration.
- Resynchronize images after recovering inaccessible partitions.

Some tasks can be performed only from an application; other tasks can be performed through an Objectivity / DB or an Objectivity / DRO tool as an alternative to using the programming interface.

NOTE In this chapter, 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, the task description identifies that tool.

You perform most Objectivity / DRO programming tasks by calling one or more member functions on a handle to a particular database or database image. In general, the programming interface for operating on a database is the same, whether or not the database has multiple images. However, as shown in Table 28-1, the member functions that return information about single-image databases signal an error if the target database has multiple images; an alternative member function must be used when multiple images exist.

Table 28-1: Functions for Single and Multiple Database Images

Member Function for Single Image	Member Function for Multiple Images
`containingPartition`	`partitionsContainingImage`
`fileName`	`getImageFileName`
`hostName`	`getImageHostName`
`pathName`	`getImagePathName`
`changePartition`	(None)

Because each image of a database must be controlled by a different autonomous partition, at least two autonomous partitions must exist in the federated database before an application can perform any data replication tasks.

Linking With Objectivity/DRO

For complete information about compiling and linking Objectivity / DB applications, see *Installation and Platform Notes* for your platform.

Windows

Objectivity / DB applications rely on an option-enabling DLL to enable appropriate features in the Objectivity / DB release and debug libraries. When you run an application that uses Objectivity / FTO and Objectivity / DRO features, you must ensure that the correct version of the option-enabling DLL is available. The correct version of the DLL is placed in the Objectivity / DB installation directory when you install Objectivity / DRO. This version enables features for both Objectivity / FTO and Objectivity / DRO.

UNIX

You must add the object module `ooRepl.o` in your link line before `liboo.a` or any other Objectivity / DB libraries. Do not also link with `ooPart.o`, because `ooRepl.o` is a superset of `ooPart.o`.

Running an Objectivity/DRO Application or Tool

Before you run an Objectivity / DRO application or tool, you must verify that the Advanced Multithreaded Server (AMS) is installed and running on every host that is to contain a replicated 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.

Within an Objectivity / DRO application or tool, any operation that creates, deletes, or modifies a database image requires the normal file-system permissions:

- If a tool performs the operation, the user account under which the tool is run must have the appropriate permissions.
- If an application performs the operation:
  - If the database-image file is being accessed remotely by the Advanced Multithreaded Server (AMS), the user account under which AMS is running needs the appropriate permissions.
  - Otherwise, the user account running the application needs the appropriate permissions.

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 data server 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 `replicate` on a handle to the database to be replicated. 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 weight for the image to be used in quorum calculations.

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 168).

Getting and Changing Attributes of an Image

The following attributes of a database image are set when the image is created:

- The data server 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 boot filename, the `-db` or `-id` option, the `-ap` option, and no other options (see the Objectivity / DB administration book)

You can call the following member functions on a database handle 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

EXAMPLE This example gets and prints various attributes of the image of database `DB1` that is contained in autonomous partition `AP1`.

```
// Application code file
#include <oo.h>
...
ooTrans transaction;
ooHandle(ooFDObj) fdH;
ooHandle(ooAPObj) apH;
ooHandle(ooDBObj) dbH;

// Start a transaction and open the federated database
transaction.start();
fdH.open("myFD", oocUpdate);
```

```
// Find the partition named "AP1"
if (!apH.open(fdH, "AP1")) {
   cerr << "Can't find partition AP1" << endl;
   trans.abort();
}

// Find the database named DB1
if (!dbH.open(fdH, "DB1")) {
   cerr << "Can't find database DB1" << endl;
   trans.abort();
}

// Print the image's host, directory path, filename, and weight
cout << "Host: " << dbH.getImageHostName(apH) << endl;
cout << "Path: " << dbH.getImagePathName(apH) << endl;
cout << "File: " << dbH.getImageFileName(apH) << endl;
cout << "Weight: " << dbH.getImageWeight(apH) << endl;
trans.commit();
```

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 `setImageWeight` on a handle to the database. The parameters specify the autonomous partition containing the image and the new weight for that image. This function fails 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 168).

EXAMPLE This example changes the weight of a database image for database `DB1` in autonomous partition `AP1`.

```
// Application code file
#include <oo.h>
...
ooHandle(ooAPObj) apH;
ooHandle(ooDBObj) dbH;

...    // Set apH to reference the partition AP1
...    // Set dbH to reference the database DB1
```

```
// Change weight of the image of DB1 in partition AP1 to 3
dbH.setImageWeight(apH, 3);
```

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 `isReplicated` on a handle to the database.

To find out how many images of a database exist, call `numImages` on a handle to the database.

EXAMPLE This example checks whether database `DB1` is replicated, and, if so, prints the number of images that exist.

```
// Application code file
#include <oo.h>
...
ooHandle(ooDBObj) dbH;
...    // Set dbH to reference the database DB1

// Get number of images of DB1
if (dbH.isReplicated()) {
   cout << "Number of images: " << dbH.numImages() << endl;
}
else {
   cout << "Database DB1 is not replicated" << endl'
}
```

Checking Availability

Must have access to: At least one partition containing an image of the database

To test whether a database is available, call `isAvailable` on a handle to the database. 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 `hasImageIn` on a handle to the database; the parameter specifies the partition of interest.

To test whether the image of a database in a particular partition is available, call isImageAvailable on a handle to the database; the parameter specifies the partition of interest. The result is true if the specified partition has an image of the database and that partition is accessible to the current process.

EXAMPLE This example checks the availability of a database image for database DB1 in autonomous partition AP1.

```
// Application code file
#include <oo.h>
...
ooHandle(ooAPObj) apH;
ooHandle(ooDBObj) dbH;

...     // Set apH to reference the partition AP1
...     // Set dbH to reference the database DB1

// Check whether DB1 has an image in AP1
if (dbH.hasImageIn(apH)) {
   // Check whether DB1's image in AP1 is available to
   // the current process
   cout << "Image of DB1 in AP1 is ";
   if (!dbH.isImageAvailable(apH)) {
      cout << "not ";
   }
   cout << "available" << endl;
}
else {
   cout << "Partition AP1 does not contain an image of DB1"
        << endl;
}

// Check whether entire database DB1 (i.e. a quorum of
// its images) is available to the current process
cout << "A quorum of images for DB1 is ";
if (!dbH.isAvailable()) {
   cout << "not ";
}
cout << "available" << endl;
```

Managing the Tie-Breaker Partition

If an even number of equally weighted database images might be separated into two equal parts by a network failure, you can assign another autonomous partition to serve as a *tie-breaker* for a particular database without actually creating a new image of the database. The tie-breaker partition has its own lock server and functions as a *pseudo-image* of the database.

You can set, 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` option (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 `setTieBreaker` on a handle to the database. The parameter is a handle to 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 data server host for the tie-breaker partition must be running a lock server.

EXAMPLE This example sets partition `AP3` to be the tie-breaker partition for database `DB1`.

```
// Application code file
#include <oo.h>
...
ooHandle(ooAPObj) apH;
ooHandle(ooDBObj) dbH;

...     // Set apH to reference the partition AP3
...     // Set dbH to reference the database DB1

// Set AP3 as tie-breaker partition for DB1
dbH.setTieBreaker(apH);
```

Removing the Tie-Breaker Partition

Must have access to: All autonomous partitions

Tool alternative: `oodeletedbimage` with the `-tiebreaker` option (see the Objectivity / FTO and Objectivity / DRO book)

To remove the tie-breaker partition for a database, call `setTieBreaker` on a handle to the database, passing a null handle as the parameter.

Finding the Tie-Breaker Partition

Must have access to: The tie-breaker partition

To find the tie-breaker partition (if any) of a database, call `getTieBreaker` on a handle to the database. This function allocates and returns a handle to the autonomous partition. If the database does not have a tie-breaker, it returns a null handle.

Finding Partitions That Contain an Image

Must have access to: All autonomous partitions that contain an image of the database

You can initialize an object iterator of class `ooItr(ooAPObj)` to find all partitions that contain an image of a particular database. To do this, you call the `partitionsContainingImage` member function on a handle to the database, passing the object iterator as a parameter.

After initializing the object iterator, you advance it through the iteration set by calling the iterator's `next` member function. See "Object Iterators" on page 293 for information about working with an object iterator.

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 `deleteImage` on a handle to the database; the parameter indicates the autonomous partition that contains the image to be deleted. This member function 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 member function signals an error.

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 168).

EXAMPLE This example deletes the database image for database `DB1` from autonomous partition `AP1`.

```
// Application code file
#include <oo.h>
...
ooHandle(ooAPObj) apH;
ooHandle(ooDBObj) dbH;

...     // Set apH to reference the partition AP1
...     // Set dbH to reference the database DB1

// Delete the image of DB1 in AP1, unless it is the
// last image of DB1
dbH.deleteImage(apH, oocFalse);
```

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 `setAllowNonQuorumRead` on a handle to the database, passing `oocTrue` 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, `setAllowNonQuorumRead` on a handle to the database, passing `oocFalse` as the parameter.

To test whether your application can read from the database if a quorum is not available, call `getAllowNonQuorumRead` on a handle to the database.

To test whether your application is currently reading a database without an available quorum, call `isNonQuorumRead` on a handle to the database.

Installing Two-Machine Handler Functions

Some sites replicate databases on two separate machines to ensure continuous availability of the data. If one machine fails or otherwise becomes unavailable, the other machine continues to provide access to the data. In this *hot-failover* configuration, one machine provides automatic hot backup for the other.

If your installation has two machines in a hot-failover configuration, you may choose to install *two-machine handler function*s in your applications. An application's two-machine handler function is called when the application can access only one image of a database, typically when the application can access the local database image but not the image on the other machine. The function tests whether the local database image can be considered to constitute a quorum by itself.

Your installation can use two-machine handler functions if it meets both the following conditions:

- Your federated database contains exactly two partitions, each local to a different data server host.
- Special hardware and software on the two machines enable applications on each machine to check on the status of the other machine.

Operation of Two-Machine Handler Functions

A two-machine handler function should check whether the other machine is still running. If not, a *machine failure* has occurred and applications on the remaining machine should be able to access the available image. If the other machine is running, however, a *network failure* has occurred; applications on one, but not both, of the two machines should be able to access the local image.

If an application's two-machine handler function returns true, the application proceeds as if the local database image by itself constituted a quorum; if it returns false, the local image does not constitute a quorum and the application cannot access the database. An application with no registered two-machine handler function is equivalent to an application with a handler function that always returns false.

Machine Failure

A two-machine handler function that detects machine failure should return true. In that case, the local database image is the only image accessible to any application. Applications that can access that image can safely update it; inconsistencies cannot occur because machine failure prevents the other image from being updated simultaneously.

Network Failure

In the case of network failure, the two-machine handler function of applications on one machine should return true and the two-machine handler function of applications on the other machine should return false. This combination will permit updates to one image while prohibiting simultaneous updates to the other image.

Working With Two-Machine Handler Functions

To install two-machine handler functions at a deployment site, you must:

- Design and implement two-machine handler functions for the site to ensure coordinated behavior between the two machines.
- In each application that runs at the site, register the appropriate handler function.

Designing the Handler Functions

A two-machine handler function is an application-defined function that must conform to the calling interface defined by the `ooTwoMachineHandlerPtr` function pointer type. It that takes no parameters and returns an `ooBoolean` value. The function should return `oocTrue` if the local database image can be considered to constitute a quorum by itself, and `oocFalse` otherwise.

Your design must ensure coordinated behavior of the two-machine handler functions registered by applications running on the two machines. Typically the same handler function is registered with all applications on a given machine. If you plan to run a particular application on both machines, the executable that runs on each machine must be created so that the application will use the appropriate handler function for the data server host where it is running.

WARNING In any given network failure, if at least one application on one machine has a two-machine handler function that returns true, no application on the other machine should have a two-machine handler function that returns true. Otherwise, applications on different machines might make inconsistent modifications to their images of the same database. If that occurs, the two images of that database will remain inconsistent even after the images are resynchronized; see “Resynchronizing Database Images” on page 569.

You can use any mechanism to ensure that handler functions on the two machines do not return true at the same time. One approach is for all applications on one machine to register a handler that returns true whenever it detects network failure and for all applications on the other machine to register a handler that returns false whenever it detects network failure.

EXAMPLE An installation has a "primary" autonomous partition and a "backup" autonomous partition. In a network failure, applications that can access the primary partition continue, but applications that can access only the backup partition are not allowed to modify replicated databases. The function `otherRunning()` returns nonzero when the other machine is running and zero when the other machine is down.

Applications on the machine with the primary autonomous partition use the handler `primaryTwoMachineHandler`. It returns true in both machine failure and network failure.

```
ooBoolean primaryTwoMachineHandler() {
   // Machine failure or network failure
   return oocTrue;
}
```

Applications on the machine with the hot-backup autonomous partition use the handler `backupTwoMachineHandler`. It returns true in a machine failure but false in a network failure.

```
ooBoolean backupTwoMachineHandler() {
   if (otherRunning()) {
      // Network failure
      return oocFalse;
   }
   else {
      // Machine failure
      return oocTrue;
   }
}
```

The preceding example gives the machine with the "primary" autonomous partition priority over the machine with the "backup" autonomous partition. To avoid giving one machine an artificial priority over the other, the handler functions might be designed so that applications on one machine have priority at certain times and those on the other machine have priority at other times. For example, the handler function used on one machine might return true if the network failure occurs on Monday, Wednesday, or Friday, while the handler function used on the other machine returns true if the network failure occurs on Tuesday, Thursday, Saturday, or Sunday.

Registering a Handler Function

A single-threaded application must register its handler function after calling `ooInit`; each thread of a multithreaded application must register the application's handler function after calling `ooInitThread`.

To register a two-machine handler function, call the global function `ooRegTwoMachineHandler`, passing a function pointer to the handler function as the parameter.

EXAMPLE In the installation described in the previous example, every application on the machine with the primary autonomous partition registers its handler with the statement:

```
ooRegTwoMachineHandler(primaryTwoMachineHandler);
```

Every application on the machine with the hot-backup autonomous partition registers its handler with the statement:

```
ooRegTwoMachineHandler(backupTwoMachineHandler);
```

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 should 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; see "Creating a Recovery Application" on page 534. The recovery procedure can resynchronize every database that has an image in the restored partitions.

WARNING During a network failure in a hot-failover configuration, if two applications on different machines both have two-machine handler functions that return true, those applications may make inconsistent modifications to their images of the same database (see "Installing Two-Machine Handler Functions" on page 566). If this situation occurs, the changes made in one partition are kept, and the changes made in the other partition are lost, when the images are resynchronized.

To resynchronize the images of a particular database, call `negotiateQuorum` on a handle to the database with the *`openMode`* parameter indicating your desired

level of access to the database. This function 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` member function 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) or the `ooCleanup` global function.

EXAMPLE This example first checks that all partitions are available. If so, it resynchronizes all replicated databases in the federated database.

```
// Application code file
#include <oo.h>
...
void resynch (const ooHandle(ooFDObj)& fdH) {
   ooItr (ooAPObj) apI;
   // Initialize an object iterator to find all partitions
   fdH.contains(apI);
   while (apI.next()) {
      if (!apI.isAvailable()) {
         return;
      }
   }
   // All partitions are available

   ooItr(ooDBObj) dbI;

   // Initialize an object iterator to find all databases
   fdH.contains(dbI);

   // Resynchronize each database
   while (dbI.next() {
      dbI.negotiateQuorum(oocRead);
   }
}
```

29

In-Process Lock Server

You can improve performance in certain Objectivity / C++ applications by using an *in-process lock server.*

This chapter describes:

- General information about the in-process lock server
- How to start an in-process lock server
- How to stop an in-process lock server
- An example application that uses 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). This product extends the Objectivity / C++ programming interface and provides a shared library that is dynamically loaded at run-time as needed.

Understanding In-Process Lock Servers

When a federated database is accessed by multiple applications, 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 function 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 opened federated database names the application's host as the lock server host, as shown in Figure 29-1.

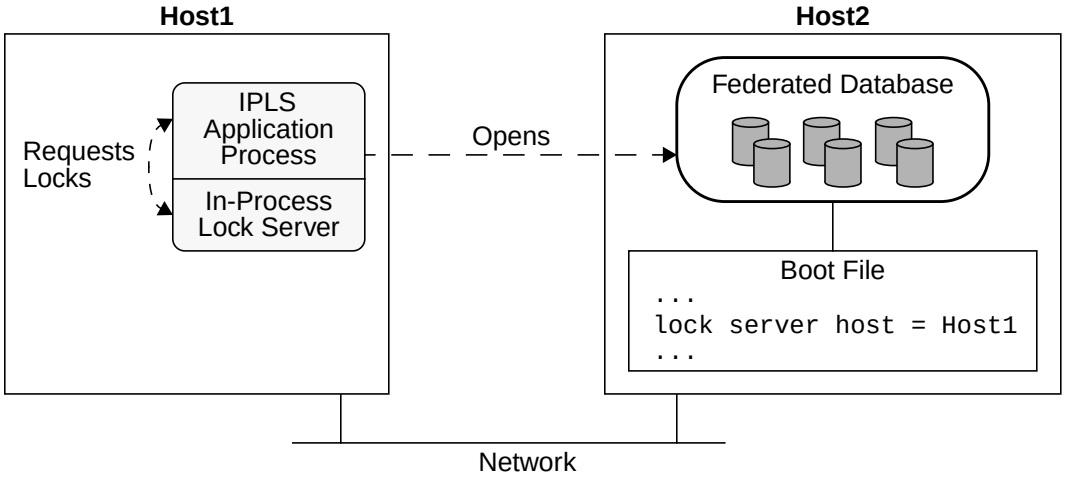


Figure 29-1 Configuration for an IPLS Application

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 standard (separate) lock-server process during development—for example, while you are creating the federated database and running the DDL processor. 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 by calling the `ooStartInternalLS` global function after calling `ooInit` and before starting the first transaction. 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 for a running lock server using the `ooCheckLS` global function.

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 the workstation if you plan to run an IPLS application on it.

You can ensure that an in-process lock server will be used by the application that started it. To do this, you call the `ooStartInternalLS` function with a parameter specifying the boot file of the federated database or autonomous partition that the application intends to open. The `ooStartInternalLS` function inspects this boot file to get the lock server host for the desired federation or partition. The in-process lock server is started only if the lock server host in the boot file is same as the current host. Otherwise, if the lock server host in the boot file is *not* the current host, `ooStartInternalLS` returns an error without starting an in-process lock server.

Stopping an In-Process Lock Server

You stop an in-process lock server by calling the `ooStopInternalLS` global function at the end of the IPLS application, after committing or aborting any transactions, and before calling `ooExitCleanup` or `exit`.

The `ooStopInternalLS` function 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 function 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 `ooStopInternalLS` returns, even if transactions are not terminated. If active transactions do not finish and the wait period expires, the function 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 a multithreaded IPLS application. Its `main` function calls `ooInit` and starts the in-process lock server, after checking whether any other lock server is running on the current host. The application then performs its Objectivity / DB operations in several threads, and finally stops the in-process lock server before calling `ooExitCleanup` to shut down.

```
// Application code file
#include <oo.h>
...
int main(const int argc, const char *const argv[]) {
   int retval = 0;
   ... // Non-Objectivity/DB operations

   // Initialize Objectivity/DB
   ooInit();

   // Find out whether an external lock server is running
   // on the current host
   if (ooCheckLS()) {
      // Give user a chance to stop the external lock server
      if (!ask_user("Lock server already running; continue?"))
         exit(1);
   }

   // If no external lock server is running, try to start
   // the in-process lock server.
   if (ooStartInternalLS() != oocSuccess)
      tell_user("Using external lock server");

   // Call functions that create threads in which Objectivity/DB
   // operations are performed
   ...
   createThread (...,&someFunc,...,parameters,...);// Pseudocode
   ...
   createThread (...,&otherFunc,...,parameters,...);// Pseudocode
   ...

   // After all threads have completed, 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
   ooStopInternalLS(300, oocTrue);
```

```
    // Prepare Objectivity/DB for shutdown
    ooExitCleanup();

    … // Non-Objectivity/DB operations
    return retval;
}
```

A

Objectivity/C++ Include Files

The source files of an Objectivity / C++ application must include the Objectivity / C++ header files containing definitions of the classes and other global names that the source file uses.

This appendix contains:

- An overview of the Objectivity / C++ header files
- A list of the Objectivity / C++ core classes that are made available by the header file `oo.h`
- A list of the special-purpose classes made available by other Objectivity / C++ header files

Overview

The following table contains an overview of the Objectivity / C++ include files and what they provide.

To Use Any of	Include
General Objectivity/C++ classes, global functions, macros, types, and constants (but no special-purpose classes or application-defined classes)	`oo.h`[a]
Application-defined class *`appClass`* (defined in the DDL file *`myClasses`*`.ddl`) Handle, object-reference, and iterator classes for *`appClass`*	Generated primary header file *`myClasses`*`.h`
Name-map class `ooMap` Handle, object-reference, and iterator classes for `ooMap` Name-map element class `ooMapElem` Handle and object-reference classes for `ooMapElem` Name-map iterator class `ooMapItr`	`ooMap.h`

To Use Any of	Include
Scalable-collection classes (`ooCollection` and derived classes) Handle, object-reference, and iterator classes for scalable-collection classes Scalable-collection iterator classes (`ooCollectionIterator` and derived classes) Administrator and comparator classes Handle, object-reference, and iterator classes for administrator and comparator classes	`ooCollections.h`
ODMG date and time classes	`ooTime.h`
Java-compatibility classes Handle, object-reference, and iterator classes for Java compatibility classes	`javaBuiltins.h`
Key-description class `ooKeyDesc` Handle, object-reference, and iterator classes for `ooKeyDesc` Key-field class `ooKeyField` Handle, object-reference, and iterator classes for `ooKeyField` Lookup-key class `ooLookupKey`	`ooIndex.h`
Administration functions `ooCleanup`, `ooGetActiveTrans`, or `ooGetResourceOwners`	`ooRecover.h`

a. A DDL file never needs to include oo.h explicitly. A source file does not need to include oo.h explicitly if it includes a generated primary header file *myClasses*.h, because generated files include oo.h.

Core Functionality

The header file `oo.h` contains definitions of the general Objectivity/C++ classes, global functions, macros, types, and constants. It provides the core functionality for Objectivity/C++ applications.

The general Objectivity/C++ classes are summarized in the following table.

Use	Class	Description
Objectivity/DB Processes	`ooContext`	Objectivity context
	`ooTrans`	Transaction object
Objectivity/DB Objects	`ooFDObj`	Federated database
	`ooDBObj`	Database
	`ooContObj`	Standard container
	`ooDefaultContObj`	Default container for a database
	`ooGCContObj`	Garbage-collectible container
	`ooObj`	Defines shared persistence behavior; base class for all application-defined classes of basic objects
	`ooGeneObj`	Genealogy
	`ooAPObj`	Autonomous partition
Handles	`ooHandle(ooFDObj)`	Federated-database handle
	`ooHandle(ooDBObj)`	Database handle
	`ooHandle(ooContObj)`	Container handle
	`ooHandle(ooDefaultContObj)`	Default-container handle
	`ooHandle(ooGCContObj)`	Garbage-collectible-container handle
	`ooHandle(ooObj)`	General-purpose handle
	`ooHandle(ooGeneObj)`	Genealogy handle
	`ooHandle(ooAPObj)`	Autonomous-partition handle

Use	Class	Description
Object References	`ooRef(ooFDObj)`	Object reference to federated database
	`ooRef(ooDBObj)`	Object reference to database
	`ooRef(ooContObj)`	Object reference to container
	`ooRef(ooDefaultContObj)`	Object reference to default container
	`ooRef(ooGCContObj)`	Object reference to garbage-collectible container
	`ooRef(ooObj)`	General-purpose object reference
	`ooRef(ooGeneObj)`	Object reference to genealogy
	`ooRef(ooAPObj)`	Object reference to autonomous partition
	`ooShortRef(ooObj)`	General-purpose short object reference
	`ooShortRef(ooGeneObj)`	Short object reference to genealogy
Object Iterators	`ooItr(ooDBObj)`	Database iterator
	`ooItr(ooContObj)`	Container iterator
	`ooItr(ooDefaultContObj)`	Default-container iterator
	`ooItr(ooGCContObj)`	Garbage-collectible-container iterator
	`ooItr(ooObj)`	General-purpose object iterator
	`ooItr(ooGeneObj)`	Genealogy iterator
	`ooItr(ooAPObj)`	Autonomous-partition iterator
VArrays	`ooVArrayT<`*`element_type`*`>`	Standard VArray
	`ooTVArrayT<`*`element_type`*`>`	Temporary VArray
	`d_Iterator<`*`element_type`*`>`	VArray iterator
Strings	`ooVString`	Variable-size string
	`ooString(`*`N`*`)`	Optimized string
Content-Based Filtering	`ooQuery`	Query object for testing a predicate
	`ooOperatorSet`	Operator set for application-defined relational operators

Use	Class	Description
Object Conversion	`ooConvertInObject`	Unconverted object that is affected by schema evolution and is to be converted to a new shape
	`ooConvertOutObject`	Converted object that is affected by schema evolution and is has been converted to a new shape
ODMG Applications	`d_Transaction`	Transaction object
	`d_Database`	ODMG database (equivalent to Objectivity/DB federated database)
	`d_Persistent_Object`	ODMG persistent object (equivalent to Objectivity/DB basic object)
	`d_Ref_Any`	ODMG generic object reference
	`d_Ref<`*`className`*`>`	ODMG object reference to objects of class *`className`* and its derived classes

Special-Purpose Classes

The Objectivity/C++ programming interface includes special-purpose classes for:

- Scalable and nonscalable persistent collections
- Date and time data
- Creating and searching indexes
- Interoperating with Java applications

To use any of these special-purpose classes, an Objectivity/C++ application must include the corresponding header file.

Scalable Collections

The header file `ooCollections.h` contains definitions of scalable persistent-collection classes and their supporting classes. Classes for scalable collections, their administrators, and their comparators are all persistence capable; the header file also includes their corresponding handle, standard object-reference, short object-reference, and object-iterator classes.

Type of Class	Class	Description
Scalable Collection	`ooCollection`	Abstract base class for all scalable-collection classes
	`ooBTree`	Abstract base class for all ordered scalable-collection classes
	`ooHashSet`	Set
	`ooTreeSet`	Sorted set
	`ooTreeList`	List
	`ooHashMap`	Object map
	`ooTreeMap`	Sorted object map
Scalable-Collection Iterator	`ooCollectionIterator` and its derived classes	Abstract base class for all scalable-collection iterator classes
	…	Its descendant classes are not documented; all share the same interface as `ooCollectionIterator`
Administrator	`ooAdmin`	Abstract base class for all administrator classes
	`ooTreeAdmin`	Administrator for ordered collections
	`ooHashAdmin`	Administrator for unordered collections
Comparator	`ooCompare`	Default comparator for scalable collections

Nonscalable Collections

The header file `ooMap.h` contains definitions of the nonscalable persistent-collection class for name maps and its supporting classes. The name-map and name-map-element classes are persistence capable; the header

file also includes their corresponding handle, standard object-reference, short object-reference, and object-iterator classes.

Class	Description
ooMap	Name map
ooMapItr	Name-map iterator
ooMapElem	Name-map element

Date and Time Data

The header file ooTime.h contains definitions of classes that represent information about date and time, as described in the ODMG standard.

Date/Time Class	Description
d_Date	Calendar date; no representation of null
d_Time	Clock time; no representation of null
d_Timestamp	Instant in time to the nearest millisecond; no representation of null
d_Interval	The duration of elapsed time between two points in time

Indexes

The header file ooIndex.h contains definitions of classes used to create and search indexes. The key-description and key-field classes are persistence capable; the header file also includes their corresponding handle, standard object-reference, short object-reference, and object-iterator classes.

Use	Class	Description
Creating Indexes	ooKeyDesc	Key description from which index can be created
	ooKeyField	Key field, describing a single attribute within a key description
Searching Indexes	ooLookupKey	Lookup key for searching an index
	…	Additional classes for creating lookup keys are described in the Objectivity/C++ programmer's reference

Java Compatibility

The header file `javaBuiltins.h` contains definitions of Java-compatibility classes. These classes are used for attributes of persistence-capable classes whose descriptions were added to the federated database schema by an Objectivity for Java application. To access objects of the Java classes, your application must include C++ classes whose data members correspond to the types of the Java attributes.

All Java compatibility classes except `ooUtf8String` are persistence capable. The header file also includes the handle, standard object-reference, short object-reference, and object-iterator cases for each persistence-capable class.

Type of Persistent Data	Class	Description
Variable-size Arrays	`oojArray`	Abstract base class for other array classes
	`oojArrayOfBoolean`	Array of Boolean elements
	`oojArrayOfCharacter`	Array of character elements
	`oojArrayOfDouble`	Array of double-precision (64-bit) floating-point numbers
	`oojArrayOfFloat`	Array of single-precision (32-bit) floating-point numbers
	`oojArrayOfInt8`	Array of 8-bit integers
	`oojArrayOfInt16`	Array of 16-bit integers
	`oojArrayOfInt32`	Array of 32-bit integers
	`oojArrayOfInt64`	Array of 64-bit integers
	`oojArrayOfObject`	Array of object references to persistent objects
Strings	`ooUtf8String`	Variable-size string of Unicode characters with the UTF-8 encoding
	`oojString`	Persistence-capable string class; used only as referenced class of the elements of an array of type oojArrayOfObject
Date and Time	`oojDate`	Instant in time with millisecond precision
	`oojTime`	Clock time with millisecond precision
	`oojTimestamp`	Instant in time with nanosecond precision

Glossary

abort a transaction. To terminate the transaction unsuccessfully, discarding (rolling back) any changes to the federated database that were made during the transaction.

ACID. Acronym for the properties—*atomicity*, *consistency*, *isolation*, and *durability*—maintained when the operations within a transaction are applied to a federated database.

affected objects. Persistent objects that are affected by a *conversion operation* on a persistence-capable class; such objects require *object conversion*. Affected objects include all existing objects of the changed class, existing objects of classes derived from the changed class, existing objects that embed objects of the changed class, and so on.

association. A kind of link between persistent objects. A source object can have a link of this kind if its class includes a definition of the association, which specifies the *destination class* of the association, its *cardinality*, *directionality*, and its propagation, copying, and versioning behavior.

atomicity. Property of a transaction that ensures that the operations within the transaction are an indivisible unit—either all the operations are performed on the federated database or none is performed.

attribute. The component data of an instance of a persistence-capable class; attribute values express the state of a persistent object. Attributes correspond to standard data members of a C++ class, fields of a Java class, or instance variables of a Smalltalk class.

autonomous partition. (*FTO*) A partitioning of data within a federated database that can perform most Objectivity/DB operations independently of any other autonomous partition, even if the others are completely unavailable.

basic object. The fundamental unit of storage in an Objectivity/DB federated database. A basic object is an instance of a class that is derived from `ooObj` but not from `ooContObj`.

bidirectional association. An *association* that has an inverse. Whenever a bidirectional association links a source object to a destination object, its inverse association links that destination object back to the source object.

boot file. A file that contains information used by an application or tool to locate and open a federated database.

buffer page. The in-memory representation of a *storage page* after it has been placed in the *Objectivity/DB cache*.

cache. See *Objectivity/DB cache*.

cardinality. A property of an *association* that specifies whether a source object can be linked to multiple destination objects. The cardinality of an association can be any one of the following: one-to-one, one-to-many, many-to-one, or many-to-many.

checkpoint a transaction. To cause any changes made during the transaction to be stored in the federated database, making them accessible to other processes. Checkpointing a transaction does not terminate the transaction.

closed handle. A *handle* to a persistent object that has the object identifier of the referenced object, but no pointer to the memory representation of that object. The referenced object may be open or closed.

closed persistent object. A *persistent object* whose persistent data is not guaranteed to be in the Objectivity / DB cache.

clustering. The process of assigning a persistent object to a storage location. Clustering a basic object assigns it to a location in a particular container; clustering a container assigns it to a particular database.

clustering directive. A parameter to `operator new` for a persistence-capable class that indicates where in the federated database the new persistent object is to be stored.

commit a transaction. To terminate the transaction successfully, causing any changes made during the transaction to become permanent in the federated database and visible to other processes using the federated database.

comparator. A persistent object used internally by a set or an object map. A sorted set uses its comparator to compare and order the elements; a sorted object map uses its comparator to compare and order the keys. An unordered set uses its comparator to compare and hash the elements; an unordered object map uses its comparator to compare and hash the keys.

composite object. An *object graph* composed of persistent objects that, together, contain the information about one complex entity.

concurrent access policy. A property of transactions that the lock server uses to determine whether a requested lock is compatible with existing locks. See *standard access policy* and *MROW*.

consistency. Property of a transaction that ensures that the transaction takes the federated 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.

container. A physical 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. A container object is both a *storage object* and a *persistent object*.

container object. The persistent-object part of a container—that is, the object that manages the group of disk pages allocated for the container within the database file of its database.

content-based filtering. A means of modifying a search operation to find only those persistent objects that meet some condition on the values of one or more of their attributes.

conversion application. An application whose only purpose is to trigger *object conversion* using any kind of conversion mechanism.

conversion function. A function whose only purpose is to set primitive data-member values in *affected objects* during *object conversion*. A conversion function uses a special interface and may be registered with any kind of application, including a *conversion application*.

conversion operation. A schema-evolution change that affects the shape of a class's objects (how these objects are laid out in storage). For example, adding a data member to a class is a conversion operation. Following a conversion operation, *object conversion* must be performed on any *affected objects*.

conversion transaction. A transaction that invokes special functions to trigger the *object conversion* of all *affected objects* in particular containers, databases, or the entire federated database. A conversion transaction can be included in any kind of application, including a *conversion application*.

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.

DDL. Data Definition Language. A language for declaring persistence-capable classes, which consists of standard C++ syntax with extensions for declaring associations and other Objectivity / DB-specific features.

DDL processor. A tool that processes DDL files, which contain DDL declarations of classes. The DDL processor generates the corresponding C++ class definitions and adds descriptions of those classes to the federated database schema.

deadlock. A situation in which eacho of two or more concurrent transactions is waiting indefinitely for a lock that will never become available because the lock is being held by one of the other waiting transactions.

deep copy. An object created by setting the value of each attribute to a copy of the value in the corresponding attribute of another object. If the original attribute has associations linking it to destination objects, the deep copy has corresponding associations linking it to deep copies of each of those destination objects.

default container. The container that is created automatically within each database.

destination class. A persistence-capable class that is related to a source class by a reference attribute or an association. An instance of the source class (or its derived classes) can be linked to an instance of the destination class (or its derived classes).

destination object. A persistent object that is the destination of a directional link from a persistent object, called the *source object* of the link. If the source object is an instance of an application-defined class, the link is a reference in a reference attribute or an association of the source object; if the source object is a persistent collection, the link is a reference to an object that the collection contains.

directionality. A property of an *association* that specifies whether an inverse link exists that allows the application to find a source object from a destination object.

durability. Property of a transaction that ensures that the effects of committed transactions are preserved in the event of system failures such as crashes or memory exhaustion.

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*.

federation. See *federated database*.

fetch. In Objectivity for Java and Objectivity / Smalltalk, an operation that reads application-specific persistent data into the memory representation of a persistent object. See also *retrieve*.

finding an Objectivity/DB object. Getting a reference to a particular Objectivity / DB object—for example, by looking up an application-assigned name or by following a link to it from a source object.

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 links from a named root, that object is considered to be garbage.

general-purpose handle. A *handle* that can reference any Objectivity / DB object. A general-purpose handle is an instance of the class `ooHandle(ooObj)`.

general-purpose object iterator. An *object iterator* that can find Objectivity / DB objects of any kind. A general-purpose object iterator is an instance of the class `ooItr(ooObj)`.

general-purpose object reference. An *object reference* that can reference any Objectivity / DB object. A general-purpose object reference is an instance of the class `ooRef(ooObj)`.

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.

handle. An Objectivity / C++ object that references an Objectivity / DB object; applications work with objects through handles. Handles serve as smart pointers for accessing the members of persistent objects, and control how long a persistent object remains represented in memory. See also *object reference*.

hashed container. A container with a hashing mechanism that can be used to maintain a *name scope*. A hashed container can be a *scope object*, as can and any basic object in the hashed container.

identifier. Integer that uniquely identifies a storage object from other objects of the same type within the same containing storage object, or that uniquely identifies an autonomous partition within the federated database. See *object identifier*.

index. A data structure that sorts persistent objects according to the values in one or more attributes of the objects. The sorting order is determined by the ordering of key fields in the key description from which the index was created.

isolation. Property of a transaction that ensures that until that transaction commits, any changes made to objects are visible only to other operations within the same transaction.

journal file. A file that contains a log of changes made during a transaction. It is used to restore the federated database if a transaction is aborted. Journal files are removed after the normal completion of a transaction.

key description. A persistent object used to create an index; it describes the class of objects to be indexed, the key fields on which to sort the indexed objects, and whether the index is to be unique.

key fields (of an index). The data members whose values are used for sorting the indexed objects.

large object. A persistent object whose persistent data spans multiple storage pages.

link. An arc in a directed graph of persistent objects. The link points from its source object to its destination object and represents a relationship that exists between the two objects. If the source object is an instance of an application-defined class, the link can be a reference in a reference attribute or an association of the source object; if the source object is a persistent collection, the link is a reference to an object that the collection contains.

list. A scalable ordered persistent collection whose elements are persistent objects; a list can contain duplicate elements. If the source object is an instance of an application-defined class, the link is a reference in a reference attribute or an association of the source object; if the source object is a persistent collection, the link is a reference to an object that the collection contains.

lock. Permission granted to an application to access an Objectivity / DB object. Locks maintain consistency during simultaneous access by multiple processes. See *read lock* and *update lock*.

lock server. A process that administers locks on Objectivity / DB objects. Before an operation can be performed on an object, an application must obtain a lock on the object from the lock server.

lock waiting. A property of a transaction that allows the transaction to wait for an object that is locked by another transaction.

logical page. A storage page containing either one or more small objects, or the header information for a large object. The logical pages within a container are numbered; the object identifier for a basic object contains the logical page number on which the object resides.

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. The concurrent access policy that allows a transaction to read the last-committed or checkpointed version of a container that is being modified. See also *standard access policy*.

MROW read lock. A read lock held by an MROW transaction.

MROW transaction. A transaction that uses the MROW concurrent transaction policy.

name map. A nonscalable unordered persistent collection of key-value pairs in which the key is a string and the value is a persistent object.

name scope. A group of persistent objects defined by a particular *scope object*; the scope object maintains a unique name for each persistent object in the name scope.

named root. In Objectivity for Java and Objectivity/Smalltalk, 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 basic objects that are no longer required must be explicitly tracked and deleted by an application. Such containers are used by applications written in a non-garbage-collected language, such as C++, although they may also be used by Objectivity for Java and Objectivity/Smalltalk applications for interoperability with Objectivity/C++ applications.

non-MROW read lock. A read lock held by a *standard transaction*.

non-persistence-capable class. A class whose instances cannot be saved independently in a federated database. Instances of some non-persistence-capable classes can be embedded within the data of instances of a persistence-capable class. Such an embedded instance cannot be found independently; it can be found only as part of the data of the containing persistent object.

nonscalable collection. A persistent collection that must fit entirely within memory when it is accessed or resized.

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 that models a group of related persistent objects. Each node in the graph is a persistent object. Each arc is a link from a *source object* to a *destination object*; it represents a relationship that exists between the two objects.

object identifier. A value that uniquely identifies a basic object within a federated database. An object identifier contains 4 components indicating the object's database, container within the database, logical page within the container, and logical slot on the page. The identifier of any storage object or autonomous partition can also be expressed in the 4-component object-identifier format.

object iterator. An object that can be initialized to find a group of Objectivity/DB objects and to step through the found objects, referencing each one in turn.

object map. A scalable persistent collection of key-value pairs in which the key and the value are both persistent objects.

object reference. An Objectivity/C++ object that references an Objectivity/DB object. Object references primarily serve as persistent addresses for linking persistent objects together. See also *handle*.

Objectivity context. An object that defines a distinct Objectivity / DB operating environment in which to execute a series of transactions. An Objectivity context is an instance of the class `ooContext`. In a multithreaded application, each thread that interacts with Objectivity / DB has its own Objectivity context; each Objectivity context has its own set of data and memory resources.

Objectivity/DB cache. A part of virtual memory assigned to an *Objectivity context*. The cache is allocated and managed by Objectivity / DB to allow high-speed access to persistent objects. When a persistent object is opened, Objectivity / DB places the logical page in which the object resides into the cache.

Objectivity/DB object. Any *storage object*, *persistent object*, or *autonomous partition*.

OID. See *object identifier*.

open. In Objectivity / C++, an operation that represents an Objectivity / DB object in memory and locks it.

open handle. A *handle* to a persistent object that has a valid pointer to the memory representation of the referenced object, which is also open. An open handle pins its referenced object in the cache; see *pinning*.

open persistent object. A *persistent object* that is locked and represented in memory.

ordered collection. A persistent collection whose objects are maintained in a particular order.

page. See *storage page*, *logical page*, and *buffer page*.

persistence-capable class. A class whose instances can be saved independently in a federated database and found independently. All persistence-capable classes are derived from `ooObj`.

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 (of a persistent object). The values in the non-pointer attributes of the object; these values are saved persistently in the federated database.

persistent object. An instance of a persistence-capable class that has been assigned a storage location in the federated database where it will be stored. When you commit the transaction in which you create a persistent object, the object is saved in the federated database, identified by a unique object identifier. A persistent object continues to exist and retains its data beyond the duration of the process that created it. Persistent objects are basic objects and containers.

pinning. A reference-counting mechanism by which Objectivity / C++ manages memory for persistent objects. Each open handle has a pin on its referenced object. The object's pin count is the number of pins on it; when the pin count falls to zero, the object is closed and may be swapped out of the Objectivity / DB cache.

predicate query. A search operation that performs content-based filtering: either a *predicate scan*, or a search for the destination objects of a to-many association that satisfy some condition.

predicate scan. A *scan* operation that searches a storage object for persistent objects of a given class that meet a condition. Such searches may be optimized by indexes.

predicate string. A string in the Objectivity / DB predicate query language that expresses a condition for content-based filtering.

read lock. A *lock* that gives an application read-only access to a particular Objectivity / DB object.

read transaction. A transaction in which the application can obtain read locks only.

reference. See *object reference*.

reference attribute. An *attribute* of a persistent object that can contain one or more object references.

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.

retrieve. In Objectivity for Java and Objectivity / Smalltalk, an operation that finds a persistent object and obtains a basic memory representation for it as a Java or Smalltalk object.

root name. In Objectivity for Java and Objectivity / Smalltalk, 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 named roots of the database or federated database.

scalable collection. A persistent collection that organizes its elements in segments that can be accessed, resized, and clustered independently of each other. This enables a scalable collection to increase in size—up to millions of elements—with minimal performance degradation.

scan. An operation that searches a storage object for objects at one or more lower levels in the storage hierarchy.

schema. A language-independent data model that describes all types and classes used in a particular federated database.

schema class name. The name used in the schema to identify a persistence-capable class. If a class description is added to the schema by the DDL processor, the name of the C++ class is used as the schema class name. (If the class description is added by Objectivity for Java or Objectivity / Smalltalk, the schema class name need not be the same as the Java or Smalltalk 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 Objectivity / C++ classes.

scope name. The name that identifies a persistent object within a particular name scope to the scope object that defines the name scope.

scope object. An Objectivity / DB object that defines a *name scope*; each persistent object in the name scope has a unique scope name that identifies the object to the scope object (but not to other objects).

set. A scalable persistent collection whose elements are persistent objects; a set cannot contain duplicate elements.

shallow copy. An object created by setting the value of each field to the value in the corresponding field of another object.

short object identifier. A value that uniquely identifies a basic object within its container. A short object identifier contains two components indicating the basic object's logical page within the container and logical slot on the page.

short object reference. An object that references a basic object, uniquely identifying that basic object within its container. Reference attributes and associations can be defined to use short object references; if they are so defined, they can link a source object to basic objects that are stored in the same container.

small object. A persistent object whose persistent data is smaller than a storage page.

sorted collection. An ordered persistent collection whose elements are sorted according to some criteria of the elements themselves.

source class. A persistence-capable class that defines either a reference attribute or an association, which can link instances of the source class to instances of the class referenced by the attribute or association.

source object. A persistent object that is the source of a directional link to a persistent object, called the *destination object* of the link. If the source object is an instance of an application-defined class, the link is

a reference in a reference attribute or an association of the source object; if the source object is a persistent collection, the link is a reference to an object that the collection contains.

standard access policy. The concurrent access policy that prevents a transaction from reading a container that is being modified by another transaction. See also *MROW*.

standard container. A container that is an instance of the class `ooContObj`.

standard object reference. An *object reference* that uniquely identifies an Objectivity / DB object within the entire federated database.

standard transaction. A transaction that uses the *standard access policy*.

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. An Objectivity / DB object that can contain other Objectivity / DB objects. The storage objects are federated databases, databases, and containers.

storage page. The minimum unit of transfer to and from disk and across networks. Objects in a federated database reside on pages. The Objectivity / DB page size can be chosen by the database developer. These pages are not the same as operating system pages. See also *logical page*.

system database. A database within a federated database or autonomous partition that contains the *schema* and other administrative information.

system name. A name, similar to a file name, that uniquely identifies a storage object or autonomous partition. The system name of a federated database uniquely identifies it to the lock server. The system name of a database uniquely identifies it among the databases of its federated database. The system name of a container uniquely identifies it among the containers of its database. The system name of an autonomous partition uniquely identifies it among the autonomous partitions of its federated database.

to-many association. An *association* that can link a source object to multiple destination objects.

to-one association. An *association* that can link a source object to a single destination object.

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.

transaction object. An object that controls interaction between an application and the federated database through transactions. A transaction object is an instance of the class `ooTrans`.

transient object. An object that exists only within the memory of the process that created it.

type number. A number of the global type `ooTypeNumber` that uniquely identifies a particular type, class, or version of a class within the schema of the federated database.

unidirectional association. An *association* with no inverse relationship. When a unidirectional association links a source object to a destination object, that destination object is not also linked to the source object.

unique index. An *index* in which each indexed object has a unique combination of values in its key fields.

unordered collection. A persistent collection whose objects are kept in an unspecified order.

update lock. A *lock* that gives an application read / write access to a particular Objectivity / DB object, allowing the application to modify that object.

update transaction. A transaction in which the application can obtain either read locks or update locks.

upgrade application. A special kind of *conversion application* that is required after certain kinds of *conversion operation*.

VArray. A one-dimensional variable-size array of elements of the same type. The element type can be a primitive type, an object-reference type, or an embedded-class type.

versioning. The ability to create and maintain many versions of a basic object in the federated database.

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W