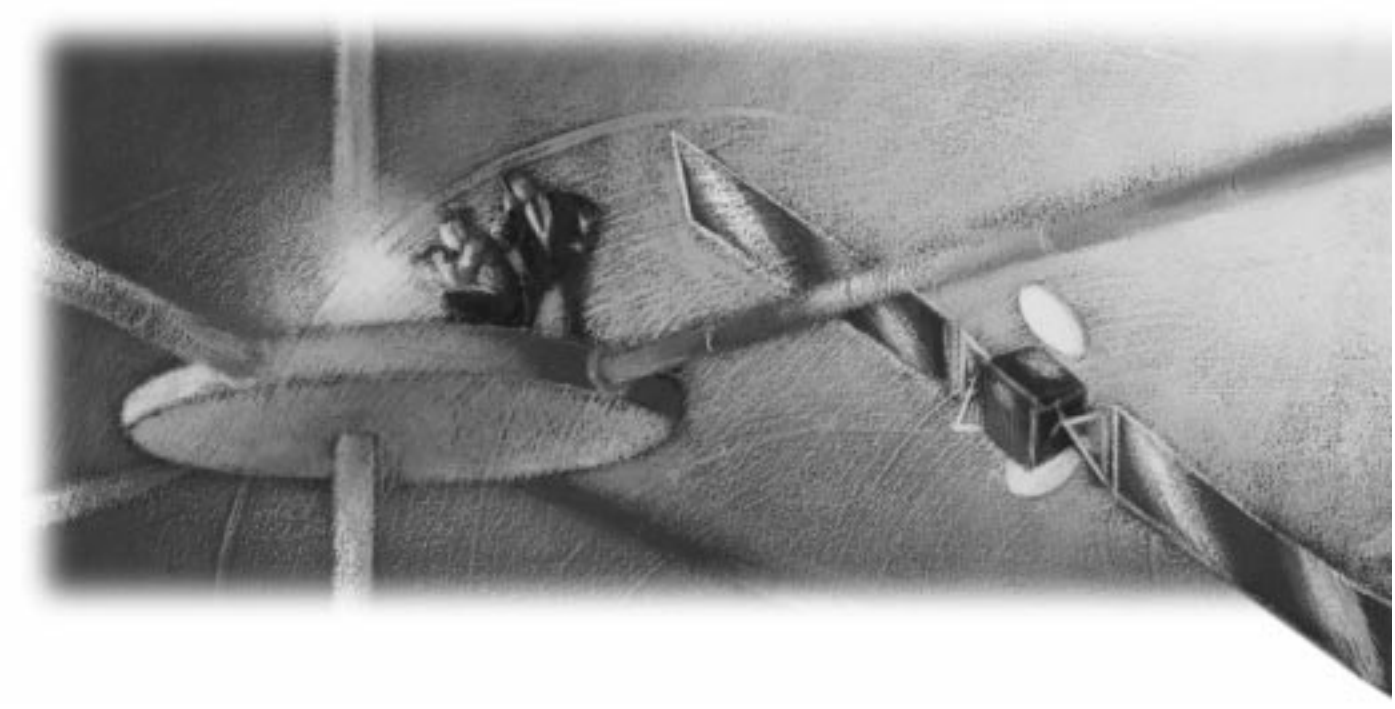

Objectivity/C++ Programmer’s Reference

Release 6.0

Objectivity/C++ Programmer's Reference

Part Number: 60-CPPRF-0

Release 6.0, September 22, 2000

The information in this document is subject to change without notice. Objectivity, Inc. assumes no responsibility for any errors that may appear in this document.

Copyright 2000 by Objectivity, Inc. All rights reserved. This document may not be copied, photocopied, reproduced, translated, or converted to any electronic or machine-readable form in whole or in part without prior written approval of Objectivity, Inc.

Objectivity and Objectivity / DB are registered trademarks of Objectivity, Inc. Objectivity / DB Fault Tolerant Option, Objectivity / FTO, Objectivity / DB Data Replication Option, Objectivity / DRO, Objectivity / DB Hot Failover, Objectivity / DB In-Process Lock Server, Objectivity / IPLS, Objectivity / DB Open File System, Objectivity / OFS, Objectivity / DB Secure Framework, Objectivity / Secure, Objectivity / C++, Objectivity / C++ Data Definition Language, Objectivity / DDL, Objectivity / C++ Active Schema, Objectivity / C++ Standard Template Library, Objectivity / C++ STL, Objectivity / C++ Spatial Index Framework, Objectivity / Spatial, Objectivity for Java, Objectivity / Smalltalk, Objectivity / SQL++, Objectivity / SQL++ ODBC Driver, Objectivity / ODBC, and Objectivity Event Notification Services are trademarks of Objectivity, Inc. Standards<ToolKit> is a trademark of ObjectSpace, Inc. Other trademarks and products are the property of their respective owners.

ODMG information in this document is based in whole or in part on material from *The Object Database Standard: ODMG 2.0*, edited by R.G.G. Cattell, and is reprinted with permission of Morgan Kaufmann Publishers. Copyright 1997 by Morgan Kaufmann Publishers.

The software and information contained herein are proprietary to, and comprise valuable trade secrets of, Objectivity, Inc., which intends to preserve as trade secrets such software and information. This software is furnished pursuant to a written license agreement and may be used, copied, transmitted, and stored only in accordance with the terms of such license and with the inclusion of the above copyright notice. This software and information or any other copies thereof may not be provided or otherwise made available to any other person.

U. S. Government Restricted Rights: Use, duplication or disclosure of the software or other information by the U. S. Government or any unit or agency thereof is subject to restrictions as set forth in subparagraph (c) (1) (ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 and the Government is acquiring only restricted rights in the software and limited rights in any technical data provided (as such terms are defined in such clause of the DFARS). If the software or other information is supplied to any unit or agency of the U. S. other than the Department of Defense, the Government's rights will be as defined in clause 52.227-19(c)(2) of the FAR or, in the case of NASA, in clause 18-52.227-86 (d) of the NASA Supplement to the FAR.

Contents

About This Book

This book, *Objectivity/C++ Programmer's Reference*, provides a detailed description of each public construct in the Objectivity / C++ programming interface. 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 guide, which describes the various tasks an application can perform with the programming interface.

Audience

This book assumes that you know how to program in C++.

Organization

- The chapter "Objectivity / C++ Programming Interface" describes the interface's naming conventions and provides an overview of the public classes and global names defined by Objectivity / C++.
- The chapter "Global Names" provides detailed descriptions of the Objectivity / C++ global types, functions, macros, and variables.
- Each subsequent chapter describes an Objectivity / C++ class, providing detailed descriptions of its public member functions. The chapters are organized in alphabetical order by class name.
- The "Topic Index" lists the topics that are presented in this book.
- The "Classes Index" contains an alphabetical list of classes, with member functions under each class.
- The "Functions and Macros Index" contains an alphabetical list of all functions, including member functions.
- The "Types and Constants Index" lists all non-class types and constants.

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

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. The Objectivity / C++ programming interface also has several extensions that are available when you purchase the corresponding add-on products.

This chapter provides:

- A list of the extensions to the core Objectivity / C++ programming interface.
- Descriptions of naming and usage conventions within the interface.
- An overview of the Objectivity / C++ programming constructs, according to their intended use.

Interface Extensions

Each of the add-on products listed in the following table defines an extension to the Objectivity / C++ interface, allowing you to access the add-on product's capabilities in addition to Objectivity / DB's core capabilities. The classes, functions, and types belonging to these extensions are described in this book, although they are available only when you purchase the corresponding add-on product. When this book describes a features of a product-specific extension, the description is preceded by the indicator shown in the table:

Add-On Product	Extension Indicator
Objectivity/DB Fault Tolerant Option	(*FTO*)
Objectivity/DB Data Replication Option	(*DRO*)
Objectivity/DB In-Process Lock Server Option	(*IPLS*)

The following products also extend the Objectivity / C++ interface; these extensions are described in separate books, as listed below:

- *Objectivity/C++ Active Schema*
- *Objectivity/C++ Standard Template Library*

Interface Conventions

This section describes the naming and usage conventions of the various programming constructs within the Objectivity / C++ programming interface.

Naming Conventions

The names of Objectivity / C++ programming constructs begin with the `oo` prefix. Multiple words in a name are indicated by mixed case, for example, `ooGetOfflineMode`. Names of non-ODMG items follow these conventions:

Item	Prefix	Description
Classes and non-class types	`oo`	Names of all classes and most types begin with the `oo` prefix.The primitive types (for example, `uint8`, `int16`, `float32`) are the exception, but their alternative names do include the prefix (for example, `ooUint8`, `ooInt16`, `ooFloat32`).
Constants	`ooc`	Names of all constants begin with the `ooc` prefix. The “c” indicates “constant”.
Variables	`oov`	Names of all variables begin with the `oov` prefix. The “v” indicates “variable”.
Macros and functions	`oo`	Names of all macros and global functions begin with the `oo` prefix.
Member functions of classes derived from `ooObj`	`oo`	Most of the member functions of classes representing Objectivity/DB objects have names that begin with the `oo` prefix.
Member functions of handle and object-reference classes		Names of member functions of handle and object-reference classes do not begin with a prefix. If the name of a member function contains the underscore character, then the member function gets or sets the contents of the object reference or handle itself.

Classes and types that conform to the Object Database Management Group specification (ODMG-93) have names that begin with the `d_` prefix. Multiple words in a name are separated by underscore characters (_), for example, `d_Object`.

The name *`appClass`* is used to represent the name of any application-defined class; *`appClass`* also appears in the name of each parameterized class that is generated for an application-defined class—for example, `ooItr(`*`appClass`*`)` is the name of the object-iterator class that is generated for the class *`appClass`*.

Global Names

The Objectivity / C++ programming interface includes functions, macros, types, constants, and variables that are defined in the global scope; they are described in detail in “Global Names” on page 23.

The global functions and macros are used for the following purposes:

- Initialization and cleanup of threads and processes.
- Establishing settings for the application as a whole.
- Establishing settings for the current Objectivity context.
- Error handling.
- Performance tuning.
- Administrative operations on the federated database.

The functions and macros are listed in “Functions and Macros Index” on page 773. The types, constants, and variables are listed in “Types and Constants Index” on page 787.

Classes

The classes in the Objectivity / C++ programming interface can be classified as follows:

- Classes for application objects, which control the application’s interaction with a federated database
- Classes for Objectivity / DB objects
- Handle classes
- Object-reference classes
- Classes that can be used as types for attributes of persistent objects
- Iterator classes
- Classes to support context-based filtering
- Classes for ODMG applications

In general, detailed descriptions of the classes appear in alphabetical order, one per chapter, starting on page 93. In the following cases, descriptions for multiple classes are combined:

- A handle class `ooHandle(`*`className`*`)` and the corresponding object-reference class `ooRef(`*`className`*`)` are combined under the name *`ooRefHandle`*`(`*`className`*`)`. For example, the classes `ooHandle(ooObj)` and `ooRef(ooObj)` are described in one chapter called *`ooRefHandle`*`(ooObj)`.
- When an ODMG class consists of a renamed Objectivity / C++ class, the ODMG class is indicated in the chapter introduction for the Objectivity / C++ class.

The classes are listed alphabetically in “Classes Index” on page 763, with member functions listed under each class. All functions, including member functions, are listed alphabetically in “Functions and Macros Index” on page 773.

Handle and Object-Reference Parameters

Objectivity / C++ programming interface considers handles and object references to be syntactically interchangeable. Because of implicit type conversion, Objectivity / C++ functions can accept a handle wherever an object reference is expected, 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`*`) &`.

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 When two declarations of an overloaded function are the same except for a parameter for specifying a handle or an object reference, they are collapsed into a single declaration, and the parameter type is given as the abbreviation *`ooRefHandle`*`(`*`className`*`)`.

Reference Summary

The tables in the following subsections present an overview of the Objectivity/C++ programming constructs, according to their intended use.

Applications

Processes and Threads	ooContext class ooInit function ooInitThread function ooTermThread function ooExitCleanup function
Transactions	ooTrans class ooMode type ooIndexMode type ooDowngradeMode type ooUpdateIndexes function
Settings for the Application	ooCheckVTablePointer function ooAMSUsage type ooSetAMSUsage function ooOfflineMode type (*FTO*) ooGetOfflineMode function (*FTO*) ooSetOfflineMode function (*FTO*) ooStartInternalLS function (*IPLS*) ooStopInternalLS function (*IPLS*) ooCheckLS function
Settings for an Objectivity Context	ooSetRpcTimeout function ooSetLockWait function ooUseIndex function ooSetHotMode function ooSetLargeObjectMemoryLimit function ooNoLock function Error handling functions and variables

Error Handling	ooStatus type ooError type oovLastError variable ooErrorLevel type oovLastErrorLevel variable oovNError variable ooResetError macro ooSignal function ooSetErrorFile function ooErrorHandlerPtr type ooRegErrorHandler macro ooGetErrorHandler macro ooMsgHandlerPtr type ooRegMsgHandler macro ooGetMsgHandler macro ooCheckVTablePointer function
Performance Tuning	ooRunStatus function ooSetHotMode function ooNoLock function ooSetLargeObjectMemoryLimit function ooShortRef(appClass) class
Administration of Federated Database	ooCleanup function ooGetActiveTrans function ooGetResourceOwners function ooTransId type ooTransInfo type ooResource type ooPurgeAps function (*FTO*) ooFileNameFormat type ooRefHandle(ooDBObj) class members ooRefHandle(ooFDObj) class members
Application-Defined Functions	ooConvertFunction type ooErrorHandlerPtr type ooRegErrorHandler macro ooGetErrorHandler macro ooMsgHandlerPtr type ooRegMsgHandler macro ooGetMsgHandler macro ooNameHashFuncPtr type ooQueryOperatorPtr type ooTwoMachineHandlerPtr type (*DRO*) ooRegTwoMachineHandler function (*DRO*)

Objectivity/DB Objects

All Objectivity/DB Objects	
Deleting	ooDelete function
Accessing	ooMode type ooLockMode type
Getting Class Information	ooTypeN function ooTypeNumber type
Federated Databases	ooFDObj class ooHandle(ooFDObj) class ooRef(ooFDObj) class
Databases	ooDBObj class ooHandle(ooDBObj) class ooRef(ooDBObj) class ooItr(ooDBObj) class ooReplace function ooContainsFilter type (*FTO*)
Containers	ooContObj class ooDefaultContObj class ooGCContObj class ooHandle(ooContObj) class ooRef(ooContObj) class ooItr(ooContObj) class ooNewConts function
All Persistent Objects	
Deleting	ooDeleteNoProp function
Object Conversion	ooConvertFunction type ooConvertInObject class ooConvertInOutObject class ooRefHandle(ooFDObj) class members ooTrans class members

Basic Objects	
General	ooObj class ooHandle(ooObj) class ooRef(ooObj) class ooShortRef(ooObj) class ooItr(ooObj) class appClass classes ooHandle(appClass) classes ooRef(appClass) classes ooShortRef(appClass) classes ooItr(appClass) classes
Scalable Persistent Collections	ooCollection class ooBTree class ooTreeList class ooTreeSet class ooTreeMap class ooHashSet class ooHashMap class ooCollectionIterator class ooCompare class ooAdmin class ooTreeAdmin class ooHashAdmin class
Nonscalable Persistent Collections	ooMap class ooMapElem class ooMapItr class ooNameHashFuncPtr type
Keyed Objects	ooKey type ooNewKey function ooKeyType type ooGetMemberOffset function ooGetMemberSize function
Versions of Basic Object	ooGeneObj class ooVersMode type
Autonomous Partitions (*FTO*)	ooAPObj class ooHandle(ooAPObj) class ooRef(ooAPObj) class ooItr(ooAPObj) class ooOfflineMode type ooGetOfflineMode function ooSetOfflineMode function ooPurgeAps function

Handles and Object References

Handles	ooHandle(ooFDObj) class ooHandle(ooDBObj) class ooHandle(ooContObj) class ooHandle(ooObj) class ooHandle(appClass) classes
Object References	
Standard	ooRef(ooFDObj) class ooRef(ooDBObj) class ooRef(ooContObj) class ooRef(ooObj) class ooRef(appClass) classes
Short	ooShortRef(ooObj) class ooShortRef(appClass) classes

Common Data Types

Primitive Data Types	int8 type int16 type int32 type int64 type uint8 type uint16 type uint32 type uint64 type float32 type float64 type char type ooBoolean type
Strings	ooVString class ooString(N) class
Variable-Size Arrays (VArrays)	ooVArrayT<element_type> classes ooTVArrayT<element_type> classes
Object References	ooRef(appClass) classes ooRef(ooObj) class ooRef(ooContObj) class ooShortRef(appClass) classes ooShortRef(ooObj) class

Dates and Times	d_Date class d_Time class d_Timestamp class d_Interval class
Java compatibility **Strings**	ooUtf8String class
Dates and Times	oojDate class oojTime class oojTimestamp class
Arrays	oojArrayOfInt8 class oojArrayOfInt16 class oojArrayOfInt32 class oojArrayOfInt64 class oojArrayOfFloat class oojArrayOfDouble class oojArrayOfCharacter class oojArrayOfBoolean class oojArrayOfObject class oojString class
Standard Template Library (STL) Strings and Containers	See *Objectivity/C++ Standard Template Library*.

Iterators

Object Iterators	ooItr(appClass) classes ooItr(ooObj) class ooItr(ooContObj) class ooItr(ooDBObj) class
Name-Map Iterators	ooMapItr class
Scalable-Collection Iterators	ooCollectionIterator class
VArray Iterators	d_Iterator<element_type> classes

Content-Based Filtering

Creating Indexes	ooKeyDesc class ooKeyField class
Using Indexes	ooUseIndex function ooIndexMode type ooUpdateIndexes function
Query Objects	ooQuery class
Lookup Keys	ooLookupKey class ooLookupFieldBase class ooEqualLookupField class ooGreaterThanLookupField class ooGreaterThanEqualLookupField class ooLessThanLookupField class ooLessThanEqualLookupField class
Extending the Predicate Query Language	ooOperatorSet class ooUserDefinedOperators variable ooQueryOperatorPtr type ooDataType type

ODMG Applications

Transactions	d_Transaction class
Objectivity/DB Objects	d_Database class d_Object class
Object References	d_Ref<d_Object> class d_Ref<appClass> classes d_Ref_Any class
Primitive Data Types	d_Boolean d_Char d_Double d_Float d_Long d_Octet d_Short d_ULong d_UShort

Dates and Times	d_Date class d_Time class d_Timestamp class d_Interval class
Strings and Arrays	d_String class d_Varray<element_type> classes
VArray Iterator	d_Iterator<element_type> classes

Global Names

Objectivity/C++ global names include functions, macros, types, constants, and variables defined in the global scope. Although all of these names are syntactically global, some variables and functions manage settings that are global to the entire application, while others manage settings that are specific to the current Objectivity context.

See:

- "Reference Index" on page 26 for an alphabetical list of global names
- "Reference Descriptions" on page 31 for individual descriptions

Global Functions and Macros

Error Conditions

Most Objectivity/C++ global functions and macros recognize a number of error conditions. If a function fails due to one of these error conditions, it puts the condition's error number and error message in the variable `oovLastError`, where the currently registered error handler can obtain them. See Chapter 23, "Error Handling," of the Objectivity/C++ programmer's guide for more information about the Objectivity/DB error handling facility.

Macro Expansion

Global macros exist to accommodate multiple platforms. The DDL processor replaces each occurrence of a macro with the code given in its definition in the Objectivity/C++ header files. Because the macro is replaced prior to compilation, the compiler does not generate any error messages using the macro name and does not perform type checking on the macro arguments. The compiler sees only the code resulting from the DDL processor's expansion of the macro.

Global and Context Settings

Some Objectivity/C++ global functions and macros manage settings that pertain globally to an entire application; others manage settings that are specific to the current Objectivity context. The following table lists these functions and macros, and the scope to which they apply:

Global Settings	Context Settings
ooCheckVTablePointer ooExitCleanup ooGetOfflineMode ooInit (signal handler setting) ooRegTwoMachineHandler ooSetAMSUsage ooSetOfflineMode ooStartInternalLS ooStopInternalLS	ooGetErrorHandler ooGetMsgHandler ooInit (file descriptors setting) ooInitThread ooNoLock ooRegErrorHandler ooRegMsgHandler ooResetError ooRunStatus ooSetErrorFile ooSetHotMode ooSetLargeObjectMemoryLimit ooSetLockWait ooSetRpcTimeout ooSignal ooTermThread ooUseIndex

Global Types

Objectivity/C++ global types include:

- Primitive data types
- Types that define constants used by various Objectivity/C++ functions
- Function-pointer types that define the syntax for application-defined error handling, message handling, query operator, and object conversion functions

Primitive Type Names

An alternate name exists for most of the primitive data types. Many of these types have equivalent names in the ODMG standard. The type name, alternate name, and ODMG name for a given primitive type can be used interchangeably. The following table summarizes the primitive types and their names. For more information about primitive types, see the Objectivity/C++ Data Definition Language book.

Category	Type Name	Alternate Name	ODMG Name	Brief Description
Integer	`int8`	`ooInt8`	(None)	8-bit signed integer type
	`uint8`	`ooUInt8`	`d_Octet`	8-bit unsigned integer type
	`int16`	`ooInt16`	`d_Short`	16-bit signed integer type
	`uint16`	`ooUInt16`	`d_UShort`	16-bit unsigned integer type
	`int32`	`ooInt32`	`d_Long`	32-bit signed integer type
	`uint32`	`ooUInt32`	`d_ULong`	32-bit unsigned integer type
	`int64`	`ooInt64`	(None)	64-bit signed integer type
	`uint64`	`ooUInt64`	(None)	64-bit unsigned integer type
Floating point	`float32`	`ooFloat32`	`d_Float`	32-bit floating-point type
	`float64`	`ooFloat64`	`d_Double`	64-bit floating-point type
Character	`char`	`ooChar`	`d_Char`	8-bit character type
Boolean	`ooBoolean`	(None)	`d_Boolean`	8-bit unsigned integer type

Global and Context Variables

Objectivity/C++ defines several variables that you can use throughout your application to refer to data, such as the most recently signaled error condition. The following table lists these variables by their semantic scope—that is, whether they manage settings that are global to the entire application or specifc to the current Objectivity context:

Variables That Store Global State	Variables That Store Context State
ooUserDefinedOperators	oovLastError oovLastErrorLevel oovNError

In a single-threaded application, Objectivity/C++ context variables behave exactly like global variables. In a multithreaded application, they are expressions whose values are specific to the current Objectivity context. Thus, changing the value of a global variable in one thread does not affect its value in other threads.

Reference Index

ooCheckLS	Tests whether a lock server is running on the specified host machine.
ooCheckVTablePointer	Suppresses or re-enables the warning messages issued for classes that do not have virtual-function tables.
ooCleanup	(administration) Recovers the specified transaction; primarily used in creating database recovery tools.
ooContainsFilter	(FTO) Type for specifying the objects to iterate over in containers or databases.
ooConvertFunction	Function-pointer type for application-defined conversion functions.
ooDataType	Type for indicating the data types of operands in the definitions of custom relational operators.
ooDelete	Removes the Objectivity/DB object that is referenced by the specified handle, propagating deletion along all associations that have delete propagation enabled.
ooDeleteNoProp	Removes only the persistent object that is referenced by the specified handle, without removing any associated objects.
ooDowngradeMode	Type for specifying how update locks are treated when checkpointing a transaction with a commit-and-hold operation.
ooError	Type for representing error identifiers for application-defined and Objectivity/C++-defined errors.
ooErrorHandlerPtr	Function-pointer type for application-defined error handler functions.
ooErrorLevel	Type for specifying the severity level of Objectivity/C++ errors.
ooExitCleanup	Leaves Objectivity/DB in a safe state for process termination.
ooFileNameFormat	(administration) Type for specifying the output format of filenames.
ooGetActiveTrans	(administration) Returns an array of transaction-information structures describing the active transactions for a particular federated database or autonomous partition; used in database recovery tools.
ooGetErrorHandler	Returns a pointer to the currently registered error handler function.

ooGetMemberOffset	Provides information required for initializing a key structure for a class—specifically, the byte offset from the start of the class to the member field that is to serve as the key.
ooGetMemberSize	Provides information required for initializing a key structure for a class—specifically, the size of the member field in the class.
ooGetMsgHandler	Returns a pointer to the currently registered message handler function.
ooGetOfflineMode	(FTO) Gets a process' current offline mode. The offline mode determines whether an application ignores or enforces the offline status of all autonomous partitions.
ooGetResourceOwners	(administration) Returns information about the resource for which the specified transaction is waiting, along with the transaction(s) currently holding that resource; used in database recovery tools.
ooIndexMode	Type for specifying how a transaction updates indexes after creating or modifying objects of an indexed class.
ooInit	Initializes Objectivity/DB in a single-threaded or multithreaded application.
ooInitThread	Initializes the current thread so that it can execute Objectivity/DB operations.
ooKey	Type for specifying the key structure for a keyed object.
ooKeyType	Type for specifying the data type of a key field within a keyed object. Constants of this type are used in creating key structures.
ooLockMode	Type for specifying how to lock an Objectivity/DB object.
ooMode	Type specifying the intended level of access to an Objectivity/DB object or for specifying the concurrent access policy for a transaction.
ooMsgHandlerPtr	Function-pointer type for application-defined message handler functions.
ooNameHashFuncPtr	Function-pointer type for an application-defined hash function to be used by instances of ooMap.
ooNewConts	Creates a batch of containers in the specified database. This macro provides better performance than calling the new operator repeatedly.

ooNewKey	Creates a keyed object for the specified class.
ooNoLock	Disables the Objectivity/DB locking facilities, removing concurrent access protection.
ooOfflineMode	(FTO) Type for specifying how an application should respond to the offline status of all autonomous partitions.
ooPurgeAps	(FTO) Purges the specified autonomous partitions from the federated database.
ooQueryOperatorPtr	Function-pointer type for application-defined relational-operator functions.
ooRegErrorHandler	Registers the specified error handler function with Objectivity/DB, replacing the previously registered error handler.
ooRegMsgHandler	Registers the specified message handler function with Objectivity/DB.
ooRegTwoMachineHandler	(DRO) Registers the specified two-machine handler function with Objectivity/DB.
ooReplace	Creates a database with the specified name in the specified federated database, replacing (deleting) any existing database with the same name.
ooResetError	Clears Objectivity/DB error flags.
ooResource	Type for representing information about a locked resource (typically a container).
ooRunStatus	Prints a summary of Objectivity/DB internal statistics to `stdout`; used primarily for performance tuning.
ooSetAMSUsage	(administration) Sets the application's policy for using the Advanced Multithreaded Server (AMS).
ooSetErrorFile	Sets the error message output file.
ooSetHotMode	Enables or disables hot mode, which controls the timing of certain internal overhead operations in the Objectivity/DB cache, for purposes of improving performance.
ooSetLargeObjectMemoryLimit	Sets the maximum amount of dynamically allocated memory that is available for caching large persistent objects.
ooSetLockWait	Sets the default lock-waiting option for a series of transactions.

ooSetOfflineMode	(FTO) Sets the offline mode for the current process.
ooSetRpcTimeout	Sets how long an application is to wait for an Objectivity server to respond before signaling a timeout error.
ooSignal	Signals the specified error and reports additional error information.
ooStartInternalLS	(IPLS) Starts an in-process lock server within an application.
ooStatus	General return type for Objectivity/C++ global functions and member functions.
ooStopInternalLS	(IPLS) Shuts down an in-process lock server within an application.
ooTermThread	Terminates the current thread's ability to invoke Objectivity/DB operations.
ooTransId	Unique identifier for a transaction.
ooTransInfo	Type for representing information about a transaction.
ooTwoMachineHandlerPtr	(DRO) Function-pointer type for application-defined message handler functions.
ooTypeN	Gets the type number of the specified class of Objectivity/DB objects.
ooTypeNumber	Type number of a class in the ooObj inheritance hierarchy.
ooUpdateIndexes	Explicitly updates all applicable indexes to reflect a new or modified object.
ooUseIndex	Enables or disables the use of indexes during a predicate query.
ooUserDefinedOperators	Operator set consulted by the predicate query mechanism to resolve application-defined relational operators used in a predicate.
ooVersMode	Type for representing the versioning behavior of a basic object.
oovLastError	Pointer to the error identifier structure for the most recent error condition.
oovLastErrorLevel	Severity level of the most recent error condition.
oovNError	Count of the total number of errors (not including warnings) that have occurred so far.

ooVoidFuncPtr	Function-pointer type for a function that has no parameters and that returns no result.
uint8	Objectivity/C++ 8-bit unsigned integer type.
uint16	Objectivity/C++ 16-bit unsigned integer type.
uint32	Objectivity/C++ 32-bit unsigned integer type.
uint64	Objectivity/C++ 64-bit unsigned integer type.
Wait Options	Integer constants that specify whether to wait for locks when starting a transaction.

Reference Descriptions

char

global type

Objectivity / C++ 8-bit character type. This type is signed on architectures where the C++ primitive type `char` is signed, and unsigned on architectures where the C++ primitive type `char` is unsigned. This type is also called `ooChar`.

float32

global type

Objectivity / C++ 32-bit floating-point type. The range and precision of this type vary for each architecture. Objectivity / DB stores numbers of this type in the native format of the architecture on which they are instantiated or modified, and automatically converts the format when these numbers are accessed from a different architecture. This type is also called `ooFloat32`.

float64

global type

Objectivity / C++ 64-bit floating-point type. The range and precision of this type vary for each architecture. Objectivity / DB stores numbers of this type in the native format of the architecture on which they are instantiated or modified, and automatically converts the format when these numbers are accessed from a different architecture. This type is also called `ooFloat64`.

int8

global type

Objectivity / C++ 8-bit signed integer type. Values of this primitive type range from -128 to +127. This type is portable across all architectures supported by Objectivity / C++. This type is also called `ooInt8`.

int16

global type

Objectivity / C++ 16-bit signed integer type. Values of this primitive type may range from -32,768 to +32,767. This type is portable across all architectures supported by Objectivity / C++. This type is also called `ooInt16`.

int32

global type

Objectivity / C++ 32-bit signed integer type. Values of this primitive type may range from -2,147,483,648 to +2,147,483,647. This type is portable across all architectures supported by Objectivity / C++. This type is also called `ooInt32`.

int64

global type

Objectivity / C++ 64-bit signed integer type. Values of this primitive type may range from $-(2^{63})$ to $+2^{63}-1$. This type is portable across all architectures supported by Objectivity / C++. This type is also called `ooInt64`.

ooAccessMode

global type

Type for specifying which data members an iterator can test in a predicate query.

Constants

`oocPublic`

The predicate query tests only public data members, preserving encapsulation.

`oocAll`

The predicate query tests all data members (public, protected, or private). To preserve encapsulation, you should use this mode only within member functions of the class you are querying.

See also

`ooItr(ooObj)::scan`
`ooItr(ooContObj)::scan`

ooAMSUsage

global type

(*administration*) Type for specifying how an application uses the Advanced Multithreaded Server (AMS) as data server software on remote hosts.

Constants

`oocAMSPreferred`

The application uses AMS whenever AMS is available.

`oocAMSOnly`

The application uses only AMS; an error is signaled if AMS is not available.

`oocNoAMS`

The application does not use AMS even if it is available.

See also `ooSetAMSUsage`

ooBoolean

global type

Objectivity/C++ Boolean type.

Constants `oocTrue`

The true value. Its integer value is 1.

`oocFalse`

The false value. Its integer value is 0.

ooCheckLS

global function

Tests whether a lock server is running on the specified host machine.

```
ooBoolean ooCheckLS(const char *host = NULL);
```

Parameters *`host`*

Name of the host machine to be checked for a running lock server. If you omit the *`hostName`* parameter or specify `NULL`, the current host is checked.

Returns `oocTrue` if a lock server is running on the specified host; otherwise, `oocFalse`.

Discussion (*IPLS*) An application can call this function to decide whether to start an in-process lock server. An in-process lock server can be started only if no other lock server process is running on the same host as the application. The application could take various actions based on the result of this function. For example, if another lock server is running on the current host, the application could report it and allow the user to choose whether to quit the application or continue using the other lock server.

See also `ooStartInternalLS`

ooCheckVTablePointer

global function

Suppresses or re-enables the warning messages issued for classes that do not have virtual-function tables.

```
void ooCheckVTablePointer(
     ooBoolean checkVptr = oocTrue);
```

Parameters

checkVptr

Specifies whether to issue a warning message whenever a persistent object is opened whose class does not have a virtual table pointer. If you omit this parameter, your application may issue such warning messages. If you specify `oocFalse`, the warning messages are suppressed.

Discussion

Sometimes when an application opens a persistent object from the database, Objectivity / DB cannot obtain the virtual function table for the object's class. This can happen when an existing application accesses instances of recently added subclasses. For example, assume that an existing application `App1` iterates over persistent instances of class `A` and calls a virtual member function on each instance. A more recent application `App2` uses class `A` as the base class for a new derived class `B` that implements the virtual member function; `App2` stores persistent instances of `B` in the federated database. The next time `App1` iterates over instances of `A`, it finds instances of both `A` and `B`, and attempts to invoke the virtual function on each found object. But because class `B` was added to the schema after `App1` was compiled and linked, `App1` has no registered virtual-function table for dispatching virtual member-function calls to their implementations in class `B`. The absence of a virtual-function table results in a fatal error, usually with an illegal memory access, invalid memory address, or similar diagnostic.

The system default is for an application such as `App1` to issue a warning message when it opens an object for which a virtual function table cannot be obtained. Depending on how the object is opened, the application can trap such warnings and handle the problem gracefully. If, however, the object is opened implicitly by a call to a virtual member function, graceful recovery is not possible.

The best solution is to relink an old application (such as `App1`) when instances of a new subclass are added to the federated database. The old application should link with the object file corresponding to the method implementation file for the new subclass. (A method implementation file is generated by the DDL processor for each persistence-capable class; if a class implements virtual functions, the file contains code for registering a virtual-function table.) Of course, relinking is necessary only for applications that will encounter instances of the new subclass (for example, through iteration over the base class) *and* call virtual member functions implemented by the subclass.

If you can guarantee that an application does not call any virtual member functions on any potential base classes, you can use the `ooCheckVTablePointer` function to suppress the warning messages that result from opening instances of a class for which no virtual-function table exists. For example, this might be useful in a conversion application that simply opens every existing object of a particular class without invoking any virtual member functions.

ooCleanup

global function

(*administration*) Recovers the specified transaction; primarily used in creating database recovery tools.

```
ooStatus ooCleanup(
     char **ppBootFilePath,
     ooTransId tId,
     int ignoreHost,
     int standalone,
     int resetLock,
     ooTransInfo *pLockOwner,
     oo2PCTransState reservedParameter);
```

Parameters

ppBootFilePath

Pointer to the string name (a `char*`) of the boot file for the federated database or any autonomous partition that is accessed by the specified transaction. If **ppBootFilePath* is 0, the boot file path is obtained from the environment variable `OO_FD_BOOT` and the pointed-to string is set to the obtained name.

tId

Transaction identifier of the transaction to recover. Transaction identifiers are typically obtained through the `ooGetActiveTrans` function.

ignoreHost

Specifies whether to permit the recovery of a transaction that started on another node. If you specify 0, the specified transaction is recovered only if it started on the same node from which `ooCleanup` was invoked. If you specify a number other than 0, the transaction is recovered regardless of where it was started.

standalone

Specifies whether to contact the lock server during the recovery process. If you specify 0, `ooCleanup` contacts the lock server to release any locks left by the transaction. You must specify a number other than 0 if you want to run `ooCleanup` when the lock server isn't running.

resetLock

Specifies `ooCleanup`'s response when it encounters another `ooCleanup` process. If you specify 0 and another `ooCleanup` process owns the recovery lock on the transaction:

- Your `ooCleanup` process fails.
- The data pointed to by *pLockOwner* is set to information about the competing `ooCleanup` process.

If you specify a number other than 0, ooCleanup resets any existing recovery lock on the transaction and recovers the transaction.

pLockOwner

Pointer to a transaction-information structure that, on return, contains information about any competing ooCleanup process that already has a recovery lock on the transaction. You can specify 0 to prevent *pLockOwner* from being updated with such information.

reservedParameter

Reserved for future development. You must specify 0.

Returns

ocSuccess if successful; otherwise ocError.

Discussion

You use this function only in special-purpose recovery tools. When you use this function, you must not call the ooInit function or perform any other nonrecovery Objectivity/DB operations (such as starting a transaction) in the same application. You may, however, use ooCleanup along with ooGetActiveTrans or ooGetResourceOwners. You should use ooCleanup only in single-threaded applications.

ooCleanup uses both the specified transaction identifier and the lock server named in the specified boot file to determine which transaction to recover. ooCleanup recovers a transaction only if the process owning that transaction is inactive.

When ooCleanup recovers a transaction, it rolls back the transaction's uncommitted changes, restoring the federated database to the logical state it was in before the transaction started. If the transaction left uncommitted changes in multiple autonomous partitions, the changes are rolled back in all of the available partitions. In particular, if the transaction left uncommitted changes in a replicated database, the changes are rolled back in all of the available images.

If the lock server is still running and *standalone* is 0, all locks held by the transaction are released. If the lock server stops—or stops and restarts—the transaction's locks are lost, so ooCleanup just rolls back changes. (If ooCleanup is to run while the lock server is stopped, *standalone* must be nonzero.)

ooCleanup puts a *recovery lock* on the specified transaction before proceeding. The recovery lock helps you determine whether additional ooCleanup processes are running. You typically run ooCleanup with resetLock set to 0 to find out whether another ooCleanup process is recovering the specified transaction. If so, you can find out about this process by inspecting the information pointed to by pLockOwner. Based on this information, you may decide to rerun ooCleanup with resetLock set to a nonzero integer value.

When you use the ooCleanup function in an application, you must include the Objectivity/C++ header file ooRecover.h. UNIX applications additionally need

to be linked with the Objectivity / DB administration library (see *Installation and Platform Notes for UNIX*).

See also "Creating a Recovery Application" on page 528 of the Objectivity / C++ programmer's guide

ooContainsFilter

global type

(*FTO*) Type for specifying the objects to iterate over in containers or databases.

Constants

`oocAllObjs`

Includes all objects in a federated database; none are filtered out.

`oocNotTransferred`

Includes only objects in the boot autonomous partition (local objects); remote objects are filtered out.

`oocTransferred`

Includes only objects *not* in the boot autonomous partition (remote objects); local objects are filtered out.

See also *ooRefHandle*(ooDBObj)::contains

ooConvertFunction

global type

Function-pointer type for application-defined conversion functions.

```
typedef void (*ooConvertFunction)(
     const ooConvertInObject &existObj,
     ooConvertInOutObj &convObj);
```

Parameters

existObj

Existing object before it has been converted.

convObj

Existing object after it has been converted.

Discussion Use *ooRefHandle*(ooFDObj)::setConversion to register this function to be invoked automatically on each converted object of a changed class. You can have no more than one conversion function for each changed persistence-capable class, and the function must not access any other persistent object.

See also *ooRefHandle*(ooFDObj)::setConversion

ooDataType

global type

Type for indicating the data types of operands in the definitions of custom relational operators.

Constants

`oocInt64T`

Indicates an operand of type `int8`, `int16`, `int32`, or `int64`.

`oocUint64T`

Indicates an operand of type `uint16`, `uint32`, or `uint64`.

`oocFloat64T`

Indicates an operand of type `float32` or `float64`.

`oocCharPtrT`

Indicates an operand of type `char *`, `char[]`, `ooString(`*`N`*`)`, `ooVString`, `ooUtf8String`, or `ooVArrayT<char>`.

`oocBooleanT`

Indicates an operand of type `ooBoolean`.

`oocInvalidTypeT`

Indicates a type that is not handled by the predicate query mechanism.

Discussion

When you define your own relational operators for use in predicate queries, you use the constants of this type to provide appropriate return values for the possible operand types.

See also

`ooQueryOperatorPtr`

ooDelete

global function

Removes the Objectivity / DB object that is referenced by the specified handle, propagating deletion along all associations that have delete propagation enabled.

```
ooStatus ooDelete(ooRefHandle(ooObj) &object);
```

Parameters

`object`

Object reference or handle to the object to be deleted. The object may be a database, an autonomous partition, a container, or a basic object. The object may have other open handles to it.

Returns

`oocSuccess` if successful; otherwise `oocError`.

Discussion

Deleting a basic object or container:

- Calls the object's destructor, if any, before deallocating storage.

- Deletes all associations from the object to destination objects. If an association has delete propagation enabled, the destination objects are deleted as well (see the Objectivity / C++ Data Definition Language book).

For basic objects and containers, the `ooDelete` function is equivalent to `operator delete`, which is available for each persistence-capable class. However, `ooDelete` is recommended because it is simpler to use.

Deleting a container or database:

- Deletes all of the persistent objects in it.
- Deletes all associations from each contained object to destination objects; if delete propagation is enabled for any of these associations, the destination objects are deleted as well.
- Does *not* call the destructors of the contained objects for performance reasons. To ensure that destructors are called, you must iterate through the contained objects and explicitly delete them before deleting the container or database.

Each deleted object is automatically removed from any bidirectional associations to maintain referential integrity. The delete operation must therefore be able to obtain an update lock on every object that is bidirectionally associated with a deleted object. Note that if another perisistent object references a deleted object through a unidirectional association or directly in one of its data members, you are responsible for removing that reference.

Deleting a database removes its file from the file system. You may not delete a database that has been made read-only; you change the database back to read-write before you can delete it (see `ooRefHandle(ooDBObj)::setReadOnly`).

WARNING Deleting a database cannot be undone by aborting the transaction.

(*FTO*) Deleting an autonomous partition:

- Clears all previously transferred containers from the autonomous partition, implicitly invoking *`ooRefHandle`*`(ooAPObj)::returnAll` (see the Objectivity / FTO and Objectivity / DRO book).
- Removes the system database and boot file for the autonomous partition from the file system. This cannot be undone by aborting the transaction.
- Removes the database images controlled by the autonomous partition from the file system. If the autonomous partition controls the last or only image of any database, an error is signalled and the autonomous partition is left unchanged.

An autonomous partition cannot be deleted if any journal files exist in its journal directory. You can correct this by using the `oocleanup` administration tool to recover the incomplete transactions.

WARNING Deleting an autonomous partition cannot be undone by aborting the transaction.

See also `ooDeleteNoProp`

ooDeleteNoProp

global function

Removes only the persistent object that is referenced by the specified handle, without removing any associated objects.

```
ooStatus ooDeleteNoProp(ooRefHandle(ooObj) &object);
```

Parameters *object*

Object reference or handle to the object to be deleted. This object may be a container or a basic object. This object may have other open handles to it.

Returns `oocSuccess` if successful; otherwise `oocError`.

Discussion This function is identical to `ooDelete`, except that it:

- Applies only to containers and basic objects, and not to databases or autonomous partitions.
- Does not propagate deletion, even along associations that have delete propagation enabled. Thus, only the specified object is deleted.

The deleted object is automatically removed from any bidirectional associations to maintain referential integrity. The operation must therefore be able to obtain an update lock on every object that is bidirectionally associated with a deleted object. Know that if another perisistent object references the deleted object through a unidirectional association or directly in one of its data members, you are responsible for removing that reference.

See also `ooDelete`

ooDowngradeMode

global type

Type for specifying how update locks are treated when checkpointing a transaction with a commit-and-hold operation.

Constants `oocNoDowngrade`

All locks obtained during the transaction are preserved as is.

`oocDowngradeAll`

All update locks obtained during the transaction are downgraded to read locks (MROW read locks if the transaction is an MROW transaction, and normal read locks otherwise), thereby improving concurrency.

See also `ooTrans::commitAndHold`

ooError

global type

Type for representing error identifiers for application-defined and Objectivity / C++-defined errors.

```
struct ooError {
     uint32 errorN;
     char *message;
};
```

Members

`errorN`

Error number. Error numbers 0 through 999,999 are reserved for Objectivity / DB system error codes. Within your application, make sure that you assign modules unique error numbers greater than 999,999.

`message`

Identical to a `printf` format string, allowing specification of a variable number of arguments.

Discussion

You declare and initialize all error identifier structures in an error message header file, and you include this file in your application source file.

See also

`ooErrorLevel`
`ooSignal`

ooErrorHandlerPtr

global type

Function-pointer type for application-defined error handler functions.

```
typedef ooStatus (*ooErrorHandlerPtr)(
     ooErrorLevel errorLevel,
     ooError &errorID,
     ooHandle(ooObj) *contextObj,
     char *errorMsg);
```

Parameters

`errorLevel`

Error level for the event being handled.

errorID

Symbolic name of the error identifier for the event being handled.

contextObj

Pointer to the handle of an object that may help the system error handler to pinpoint the context of the error. Set this parameter to 0 if you do not want to use context information.

errorMsg

String created by running `vsprintf` over the message part of the error identifier and its arguments.

Returns `oocSuccess` if the error level of the event is acceptable, otherwise `oocError`.

Discussion A registered error handler is a function that is invoked whenever the `ooSignal` function signals an error for an event. Objectivity/DB provides a predefined error handler that is automatically registered when you start your application. You can replace the predefined error handler by defining a custom error handler and using the `ooRegErrorHandler` function to register it.

Your custom error handler must be able to accept the arguments passed to it by the `ooSignal` function. Therefore, your error handler must conform to the calling interface defined by this function-pointer type.

See also Chapter 23, "Error Handling," of the Objectivity/C++ programmer's guide
`ooRegErrorHandler`
`ooSignal`

ooErrorLevel

global type

Type for specifying the severity level of Objectivity/C++ errors.

Constants `oocNoError`

Indicates an event with no severity level.

`oocWarning`

Indicates an abnormal event. No action should be taken beyond notifying the user of the condition.

`oocUserError`

Indicates a nonfatal user error detected directly by the programming interface. Such errors can be directly attributable to the application programmer—for example, inconsistent user data passed to the programming interface.

`oocSystemError`

Indicates a nonfatal system error detected by an Objectivity / DB operation, or a nonfatal user error that slipped through the consistency checks in the programming interface. In-depth analysis is usually required to determine whether the error is attributable to Objectivity / DB or to the user.

`oocFatalError`

Indicates the most severe type of error that can occur in the system. Such errors are signalled when Objectivity / DB detects an unrecoverable internal inconsistency that might have already caused data corruption. An application should respond to such errors by aborting the active transaction and shutting down as quickly as possible.

See also `ooError`
`ooSignal`

ooExitCleanup

global function

Leaves Objectivity / DB in a safe state for process termination.

```
void ooExitCleanup();
```

Discussion

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` 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`.

WARNING Executing `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 (see the discussion below). 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`.

You *must* call `ooExitCleanup` explicitly in any multithreaded application that will run on a Windows platform and any multithreaded application that must support future portability to Windows. 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.

Although `ooExistCleanup` is not required in a single-threaded application, you can call it from any application. In a single-threaded application, `ooExitCleanup` aborts the active transaction, if any.

In a multithreaded application, 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 (see *Installation and Platform Notes* for your platform). You must not attempt to restart threads suspended by `ooExitCleanup`.

ooFileNameFormat

global type

(*administration*) Type for specifying the output format of filenames.

Constants

`oocNative`

Specify filenames as full pathnames—for example, `/net/mach3/usr/mnt/project/myfd.FDDB`.

`oocHostLocal`

Specify filenames in host format *`hostName`*`::`*`localPath`*—for example, `mach3::/mnt/fred/project/myfd.FDDB`.

ooGetActiveTrans

global function

(*administration*) Returns an array of transaction-information structures describing the active transactions for a particular federated database or autonomous partition; used in database recovery tools.

```
ooStatus ooGetActiveTrans(
     ooTransInfo **ppTrans,
     char **ppBootFilePath,
     char *pHost,
     unsigned int *pUid);
```

Parameters

ppTrans

Pointer to the returned array (an `ooTransInfo*`) of transaction-information structures describing the active transactions. This array is allocated by `ooGetActiveTrans`. The array is terminated by an entry whose transaction identifier is `oocInValidTransId`. The array of transaction-information structures is valid until a subsequent call to `ooGetActiveTrans`.

ppBootFilePath

Pointer to the string name (a `char*`) of a federated-database or autonomous-partition boot file. If *`*ppBootFilePath`* is 0, the boot file path is obtained from the environment variable `OO_FD_BOOT` and the pointed-to string is set to the obtained name.

pHost

Host for filtering the active transactions. If specified, the returned array describes only transactions started on the specified host, subject to filtering based on other parameters. Specify 0 to include transactions started on any node.

pUid

User identifier for filtering the active transactions. If specified, the returned array describes only transactions started by the specified user, subject to filtering based on other parameters. Specify 0 to include transactions started by any user.

Returns

`oocSuccess` if successful; otherwise `oocError`.

Discussion

This function uses the journal files of the specified federated database or autonomous partition to identify the active transactions to be included in the array.

You use this function only in special-purpose recovery tools. When you use this function, you must not call the `ooInit` function or perform any other nonrecovery Objectivity / DB operations (such as starting a transaction) in the same application. You may, however, use `ooGetActiveTrans` along with

ooCleanup or ooGetResourceOwners. You should use ooGetActiveTrans only in single-threaded applications.

When you use the ooGetActiveTrans function in an application, you must include the Objectivity/C++ header file ooRecover.h. UNIX applications additionally need to be linked with the Objectivity/DB administration library (see *Installation and Platform Notes for UNIX*).

See also "Creating a Recovery Application" on page 528 of the Objectivity/C++ programmer's guide

ooGetErrorHandler

global macro

Returns a pointer to the currently registered error handler function.

```
ooErrorHandlerPtr ooGetErrorHandler();
```

Returns Pointer to the currently registered error handler function.

Discussion In a multithreaded application, this macro returns a pointer to the error handler that is registered in the current Objectivity context.

See also ooRegErrorHandler

ooGetMemberOffset

global macro

Provides information required for initializing a key structure for a class—specifically, the byte offset from the start of the class to the member field that is to serve as the key.

```
uint32 ooGetMemberOffset(className, memberName);
```

Parameters *className*

Name of the persistence-capable class for which a key structure is to be initialized.

memberName

Name of the nonstatic data member that is to serve as the key. This data member must be public.

Returns Offset to the member field in number of bytes from the start of the class.

See also ooKey
ooNewKey

ooGetMemberSize

global macro

Provides information required for initializing a key structure for a class—specifically, the size of the member field in the class.

```
uint32 ooGetMemberSize(className, memberName);
```

Parameters

className

Name of the persistence-capable class for which a key structure is to be initialized.

memberName

Name of the nonstatic data member that is to serve as the key. This data member must be public.

Returns

Size of the member field in bytes.

See also

ooKey
ooNewKey

ooGetMsgHandler

global macro

Returns a pointer to the currently registered message handler function.

```
ooMsgHandlerPtr ooGetMsgHandler();
```

Returns

Pointer to the currently registered message handler function.

Discussion

In a multithreaded application, this macro returns a pointer to the message handler that is registered in the current Objectivity context.

See also

ooRegMsgHandler

ooGetOfflineMode

global function

(*FTO*) Gets a process' current offline mode. The offline mode determines whether an application ignores or enforces the offline status of all autonomous partitions.

```
ooOfflineMode ooGetOfflineMode();
```

Returns

Either `oocIgnore` or `oocEnforce`.

ooGetResourceOwners global function

(*administration*) Returns information about the resource for which the specified transaction is waiting, along with the transaction(s) currently holding that resource; used in database recovery tools.

```
ooStatus ooGetResourceOwners(
     ooTransInfo **ppOwners,
     ooResource *pResource,
     char **ppBootFilePath,
     ooTransId tId);
```

Parameters

ppOwners

Pointer to an array (an `ooTransInfo*`) of the transaction-information structures for the transactions that own the resource in question. This array is allocated by `ooGetResourceOwners`. The array is terminated by an entry whose transaction identifier (*`tId`* member) is set to `oocInValidTransId`. The array of transaction-information structures is valid until a subsequent call to `ooGetResourceOwners`.

pResource

Pointer to the resource-information structure describing the resource for which the transaction is waiting.

ppBootFilePath

Pointer to the string name (a `char*`) of the boot file that names the lock server to be queried. You may specify a federated database or autonomous partition boot file. If *`*ppBootFilePath`* is 0, the boot file path is obtained from the environment variable `OO_FD_BOOT` and the pointed-to string is set to the obtained name.

tId

Transaction identifier of the waiting transaction for which resource information is to be returned.

Returns

`oocSuccess` if successful; otherwise `oocError`.

Discussion

This function queries the lock server that is listed in the boot file to identify:

- The resource for which the specified transaction is waiting. A resource is an Objectivity / DB object, typically a container.
- The resource's owner(s). This may be a single transaction with a read or update lock on the resource, or one or more transactions with MROW read locks and possibly a transaction with an update lock.

You use this function only in special-purpose recovery tools. When you use this function, you must not call the `ooInit` function or perform any other non-recovery Objectivity / DB operations (such as starting a transaction) in the

same application. You may, however, use `ooGetResourceOwners` along with `ooCleanup` or `ooGetActiveTrans`. You should use `ooGetResourceOwners` only in single-threaded applications.

When you use the `ooGetResourceOwners` function in an application, you must include the Objectivity/C++ header file `ooRecover.h`. UNIX applications additionally need to be linked with the Objectivity/DB administration library (see *Installation and Platform Notes for UNIX*).

See also "Creating a Recovery Application" on page 528 of the Objectivity/C++ programmer's guide

ooIndexMode

global type

Type for specifying how a transaction updates indexes after creating or modifying objects of an indexed class.

Constants

`oocInsensitive`

All applicable indexes are updated automatically when the transaction commits.

`oocSensitive`

All applicable indexes are updated automatically when the next predicate scan is performed in the transaction or, if no scans are performed, when the transaction commits. This allows you to change indexed objects and then scan them in the same transaction using any applicable index.

`oocExplicitUpdate`

Applicable indexes are updated only by explicit calls to the `ooUpdateIndexes` function.

See also `ooUpdateIndexes`
`ooTrans::start`

ooInit

global function

Initializes Objectivity/DB in a single-threaded or multithreaded application.

```
ooStatus ooInit(
     uint32 nFiles = 12,
     uint32 nPages = 200,
     uint32 nMaxPages = 500,
     ooBoolean installSigHandler = oocTrue);
```

Parameters

nFiles

Number of file descriptors to be reserved for Objectivity / DB. The reserved file descriptors are a subset of those allocated for the entire process. *nFiles* limits the number of files that Objectivity / DB can have open concurrently in an Objectivity context. When this limit is reached, Objectivity / DB must close files before opening additional files. (Note that applications have no explicit way to control when files are closed.) In a multithreaded application, the *nFiles* limit applies to *each* Objectivity context created by the application.

nPages

Initial number of buffer pages to be allocated for each buffer pool in the Objectivity / DB cache. This parameter affects the cache in the main thread; the cache sizes in additional Objectivity contexts are set by the `ooContext` constructor.

nMaxPages

Maximum number of buffer pages that can be allocated for each buffer pool in the Objectivity / DB cache. This number is limited by the amount of available swap space. If you specify 0, the cache may grow as needed up to the amount of available swap space. This parameter affects the cache in the main thread; the cache sizes in additional Objectivity contexts are set by the `ooContext` constructor.

installSigHandler

Specifies whether to register the predefined Objectivity / DB signal handler. If you omit this parameter, the predefined signal handler is registered. The signal handler is global to the process; the same signal handler is used in every thread.

Discussion

Your application must call `ooInit` early in `main()` and before using any Objectivity / DB services. If your application has multiple threads, it must call `ooInit` in the main thread (the thread that starts implicitly when you start the application) before starting any other threads. You should call `ooInit` only once in an application. Objectivity / DB ignores subsequent calls to this function. Do not call `ooInit` before `main()`—for example, do not call it in a global constructor.

The `ooInit` function implicitly calls the `ooInitThread` function for the main thread of the application, initializing that thread and creating an Objectivity context for it. A single-threaded application has just this single Objectivity context; a multithreaded application normally creates additional Objectivity contexts.

Every Objectivity context has an Objectivity / DB cache, which is a portion of the process' virtual memory that is reserved for caching persistent objects. The *nPages* and *nMaxPages* parameters of the `ooInit` function affect the size of the

Objectivity / DB cache in the Objectivity context of the main thread. The cache sizes in other, subsequently-created Objectivity contexts are set by similar parameters of the `ooContext` constructor.

An Objectivity / DB cache consists of *buffer pages*, which are the same size as the storage pages in the federated database opened by the application. Within the cache, buffer pages are organized into two *buffer pools*, plus a memory pool:

- The small-object buffer pool of pages containing small objects (objects that are a page in size or smaller).
- The large-object buffer pool of header pages for large objects (objects that span multiple pages).
- The large-object memory pool of dynamically allocated memory blocks for large objects.

The *`nPages`* and *`nMaxPages`* parameters determine the minimum and maximum sizes of each of the two buffer pools within the cache. If no large objects are opened, the maximum size of the Objectivity / DB cache is the maximum size of the two buffer pools, given by the formula: `2 * nMaxPages * pageSize` bytes, where *`pageSize`* is the number of bytes per storage page in the federated database. When large objects are opened, the cache increases by the size of the large-object memory pool, whose limit is controlled by the `ooSetLargeObjectMemoryLimit` function.

The default values for *`nPages`* and *`nMaxPages`* are usually sufficient for early versions of an application. However, because cache size affects performance, you may want to try different values during performance tuning.

Do not call the `ooInit` function in applications that call `ooCleanup`, `ooGetActiveTrans`, or `ooGetResourceOwners`.

Returns `oocSuccess` if successful; otherwise `oocError`.

See also `ooSetLargeObjectMemoryLimit`
`ooInitThread`

ooInitThread

global function

Initializes the current thread so that it can execute Objectivity / DB operations.

```
void ooInitThread (ooContext *context = new ooContext());
```

Parameters *`context`*

Objectivity context to be set for the thread. If you omit this parameter, a new Objectivity context is created for the thread. If you specify 0, a null Objectivity context is set for the thread; you must set a nonnull context explicitly before invoking any Objectivity / DB operations in that thread. If

you specify an existing Objectivity context, that context is set unless it is already the context for another thread; in this case, an error is signaled.

Discussion

Every thread that invokes Objectivity / DB operations must call this function before any of those operations are invoked.

It is *not* necessary to call this function:

- In the process' main thread. This thread is initialized automatically by the `ooInit` function, which implicitly calls the `ooInitThread` function.
- In threads that do not invoke Objectivity / DB operations.

See also

`ooTermThread`
`ooContext::setCurrent`

ooKey

global type

Type for specifying the key structure for a keyed object.

```
struct ooKey {
     ooKeyType type;
     uint32 offset;
     uint32 size;
     void *value;
};
```

Members

type

Type of the key field in the keyed object.

offset

Offset of the key field (in bytes) from the beginning of the keyed object's class.

size

Size of the key field (in bytes) in the keyed object's class.

value

Pointer to the value for which you want to search.

Discussion

A key structure provides Objectivity / DB with information about a specific key field, including the value for which you want to search.

When *type* is `oocString`, *value* must be a character array ending in a null terminator. If you specify a *value* that is shorter than *size*, it is padded with null characters. If you specify a *value* that is longer than *size*, an error results. When *type* is `oocCharArray`, the character array *value* has length *size* and may include null characters.

See also
ooKeyType
ooNewKey
ooGetMemberOffset
ooGetMemberSize

ooKeyType

global type

Type for specifying the data type of a key field within a keyed object. Constants of this type are used in creating key structures.

Constants

`oocUint16`

Objectivity / C++ type `uint16`.

`oocUint32`

Objectivity / C++ type `uint32`.

`oocInt16`

Objectivity / C++ type `int16`.

`oocInt32`

Objectivity / C++ type `int32`.

`oocFloat32`

Objectivity / C++ type `float32`.

`oocFloat64`

Objectivity / C++ type `float64`.

`oocString`

A fixed array of characters (type `char *`). In a key structure, a value of this type must have a null terminator. If you specify a value containing multiple null characters, only the characters up to and including the first null character are used.

`oocCharArray`

Character array with no particular terminator character. In a key structure, a value of this type may include null characters. The length of the character array is determined by the size of the key field, which is specified in the key structure.

See also
ooKey

ooLockMode

global type

Type for specifying how to lock an Objectivity / DB object.

Constants

`oocLockRead`

The object is locked for read. Other processes may also obtain read locks on the object. If all read locks on an object were obtained within multiple reader, one writer (MROW) transactions, another process may obtain an update lock; however, if a read lock on an object was obtained within a standard (non-MROW) transaction, no other process may obtain an update lock.

`oocLockUpdate`

The object is locked for update. No other process may obtain an update lock on the object. A process running an MROW transaction may obtain a read lock on the object; a process running a standard (non-MROW) transaction may not obtain a read lock.

ooMode

global type

Type specifying the intended level of access to an Objectivity / DB object or for specifying the concurrent access policy for a transaction.

Constants

`oocNoOpen`

(Used by various open, lookup, and scan operations.) A reference is set to the object without opening or locking it.

`oocRead`

(Used by various open, lookup, and scan operations.) The object is opened only for read.

`oocUpdate`

(Used by various open, lookup, and scan operations.) The object is opened for update (read and write).

`oocNoMROW`

(Used by `ooTrans::start`.) The standard concurrent access policy is activated for the transaction being started.

`oocMROW`

(Used by `ooTrans::start`.) The multiple reader, one writer (MROW) concurrent access policy is activated for the transaction being started.

ooMsgHandlerPtr

global type

Function-pointer type for application-defined message handler functions.

```
typedef void (*ooMsgHandlerPtr)(
     char *message);
```

Parameters

message

String containing the message to be displayed.

Discussion

A registered message handler is a function that is called by the Objectivity / DB-defined error handler to write a signaled error's message to an output device. Objectivity / DB provides a predefined message handler that is automatically registered when you start your application. This predefined message handler prints error messages to `stderr`. You can replace the predefined message handler by defining a custom message handler and using the `ooRegMsgHandler` function to register it.

Your custom message handler must be able to accept the argument passed to it by the predefined error handler. Therefore, your message handler must conform to the calling interface defined by this function-pointer type.

See also

`ooRegMsgHandler`

ooNameHashFuncPtr

global type

Function-pointer type for an application-defined hash function to be used by instances of `ooMap`.

```
typedef uint32 (*ooNameHashFuncPtr)(
     const char *name,
     uint32 size);
```

Parameters

name

String name to be hashed.

size

Number of bins in the hash table.

See also

`ooMap::set_nameHashFunction`

ooNewConts

global macro

Creates a batch of containers in the specified database. This macro provides better performance than calling the new operator repeatedly.

```
void ooNewConts(
     className,
     uint32 numberOfConts,
     const ooHandle(ooObj) &near,
     uint32 hash,
     uint32 initPages,
     uint32 percentGrowth,
     ooBoolean open,
     ooHandle(ooObj) *pHandle);
```

Parameters

className

Class name of the containers to be created. The class must have a default constructor.

numberOfConts

Number of containers to be created. This number is limited to 32766.

near

Clustering directive that specifies where to locate the new containers:

- If *near* is a handle to a database, the new containers are created in that database.
- If *near* is a handle to a container or basic object, the new containers are created in the database that contains the referenced container or basic object.

An error is signalled if *near* is 0.

hash

Determines whether to create hashed containers. You must create hashed containers if you intend to use them or any objects in them as scopes for naming objects, or if you intend to create keyed objects in the containers.

- Specify 0 to create nonhashed containers.
- Specify 1 or greater to create hashed containers.

  The number you specify is the clustering factor for any keyed objects created in the containers. A clustering factor is the number of sequentially keyed objects to be placed onto a page. A clustering factor of 1 maximally distributes keyed objects across pages. A higher number means fewer pages need to be read when finding sequences of keyed objects.

initPages

Initial number of logical pages to be allocated for each container. Specify 0 to use the system default value (4 pages for hashed containers, and 2 pages for nonhashed containers). The maximum value is 65535.

percentGrowth

Amount by which each container may grow, expressed as a percentage of its current size. Specify 0 to use the system default value (10%).

open

Specifies whether to open or close the batch containers after creation. Specify `oocTrue` to leave all the containers open; specify `oocFalse` to close all the containers. It is better to close all the containers if they will not be used immediately.

pHandle

Pointer to an array of container handles that, on return, stores the handles to the newly created containers. The number of handles in this array should be greater than or equal to *numberOfConts*.

Discussion

You can use the first handle in the handle array for error checking—if the operation is successful, the first handle is not 0; otherwise, the first handle is 0.

This macro does not provide a way to give a system name to each container. You may not use this macro as an expression—for example, in an assignment statement.

See also

`ooContObj::operator new`

ooNewKey

global macro

Creates a keyed object for the specified class.

```
ooHandle(ooObj) ooNewKey(
     className,
     (initializer),
     const ooHandle(ooObj) &contH,
     ooKey keyStruct);
```

Parameters

className

Name of the class for which a keyed object is to be created. The class must be a basic object class.

initializer

List of arguments to be passed to the constructor for class *className*.

contH

Handle to the container in which the keyed object is to be created. The container must be a hashed container. If you specify the handle to a database, the keyed object is created in the default container for the database.

keyStruct

The key structure with which the keyed object is created.

Returns Handle to a newly created keyed object.

See also ooKey
ooGetMemberOffset
ooGetMemberSize
ooRefHandle(ooObj)::lookupObj

ooNoLock

global function

Disables the Objectivity / DB locking facilities, removing concurrent access protection.

```
ooStatus ooNoLock();
```

Returns oocSuccess if successful; otherwise oocError.

Discussion If your application is guaranteed exclusive access to a federated database and it requires maximum performance, you may consider disabling locking to remove the runtime overhead associated with managing locks. However, if another process has access to the same data as your application, unpredictable results may occur, including corruption of your data. Therefore, use ooNoLock with caution. For most applications, benefits such as data integrity and concurrent access far outweigh the slight performance gain obtained by disabling locking.

The ooNoLock function is called after ooInit, but before any transactions are started.

In a multithreaded application, this function disables locking for transactions in the current Objectivity context.

ooOfflineMode

global type

(*FTO*) Type for specifying how an application should respond to the offline status of all autonomous partitions.

Constants oocIgnore

The application ignores the offline status of autonomous partitions.

`oocEnforce`

The application enforces the offline status of autonomous partitions.

See also `ooGetOfflineMode`
`ooSetOfflineMode`

ooPurgeAps

global function

(*FTO*) Purges the specified autonomous partitions from the federated database.

```
ooStatus ooPurgeAps(char **apNames, int numOfAps);
```

Parameters

apNames

An array of strings, where each string is the system name of an autonomous partition to be purged.

numOfAps

The number of system names in *apNames*.

Returns `oocSuccess` if successful; otherwise `oocError`.

Discussion

You use this function only in special-purpose recovery tools. You must call this function from within an update transaction (that is, after starting a transaction and opening the federated database for update).

This function is called to remove permanently inaccessible partitions from the federated database. Purging a partition is similar to deleting it; the difference is that you can purge partitions that are not accessible, but you cannot delete an inaccessible partition.

If any currently unavailable partition is *not* specified for purging, this function returns `oocSuccess`; however, the transaction cannot commit because it can't propagate the catalog changes to the unavailable partition.

If a quorum of images for a particular database still exists after removal of all the unavailable partitions, this function removes any catalog reference to that database's images and / or tie breakers in the purged partitions. If a quorum of images for a particular database is no longer available, this function deletes that database from the federated-database catalog. The database files, however, are not deleted.

WARNING To use this function safely, you must be *absolutely certain* that no process is active in any of the partitions to be purged. Purging partitions cannot be undone.

---

ooQueryOperatorPtr

global type

Function-pointer type for application-defined relational-operator functions.

```
typedef ooBoolean (*ooQueryOperatorPtr)(
     const void *lPtr,
     const void *rPtr,
     ooDataType lAType,
     ooDataType rAType);
```

Parameters

lPtr

Pointer to the left operand of the operator.

rPtr

Pointer to the right operand of the operator.

lAType

Indicates the data type of the left operand.

rAType

Indicates the data type of the right operand.

Returns

`oocTrue` if the relation expressed by the operator is satisfied by the operands; otherwise `oocFalse`.

Discussion

An application can define its own relational operators for use in predicates. To do this, the application defines an operator function that tests whether two operands are of an appropriate type and whether the desired relation holds between them. When registered along with a token, the operator function is called by the Objectivity / DB predicate-query mechanism each time the token is encountered in a predicate.

The operator function must conform to the calling interface of this function-pointer type so that the predicate-query mechanism can pass it the operands to be compared and a pair of constants indicating the operands' data types. 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`. For information about the types that are recognized by the predicate query mechanism, see the Objectivity / C++ programmer's guide.

See also

`ooDataType` global type
`ooUserDefinedOperators` global variable
`ooOperatorSet::registerOperator`

ooRegErrorHandler

global macro

Registers the specified error handler function with Objectivity / DB, replacing the previously registered error handler.

```
ooErrorHandlerPtr ooRegErrorHandler(
     ooErrorHandlerPtr handlerName);
```

Parameters

handlerName

Name of the error handler function to be registered.

Returns

A function pointer to the previously registered error handler.

Discussion

A registered error handler is a function that is invoked whenever the `ooSignal` function signals an error. Objectivity / DB provides a predefined error handler that is automatically registered when you start your application. You can replace the predefined error handler by defining a custom error handler and using `ooRegErrorHandler` to register it.

In a multithreaded application, the Objectivity / DB-defined error handler is automatically registered in each Objectivity context that your application creates. You can register a custom error handler in a thread's Objectivity context by invoking `ooRegErrorHandler` in that thread.

`ooRegErrorHandler` returns a pointer to the replaced error handler so that you can register it again when desired. When an error is signaled, the most recently registered error handler is invoked.

See also

Chapter 23, "Error Handling," of the Objectivity / C++ programmer's guide

ooRegMsgHandler

global macro

Registers the specified message handler function with Objectivity / DB.

```
ooMsgHandlerPtr ooRegMsgHandler(ooMsgHandlerPtr msgHandler);
```

Parameters

msgHandler

Pointer to the message handler function to be registered.

Returns

A function pointer to the previously registered message handler.

Discussion

A registered message handler is a function that is called by the Objectivity / DB-defined error handler to write a signaled error's message to an output device. Objectivity / DB provides a predefined message handler that is automatically registered when you start your application. This predefined message handler prints error messages to `stderr`. You can replace the

predefined message handler by defining a custom message handler and using `ooRegMsgHandler` to register it.

In a multithreaded application, the Objectivity / DB-defined message handler is automatically registered in each Objectivity context that your application creates. You can register a custom message handler in a thread's Objectivity context by invoking `ooRegMsgHandler` in that thread.

`ooRegMsgHandler` returns a pointer to the replaced message handler so that you can register it again when desired. The Objectivity / DB-defined error handler invokes the most recently registered message handler.

See also Chapter 23, "Error Handling," of the Objectivity / C++ programmer's guide

ooRegTwoMachineHandler

global function

(*DRO*) Registers the specified two-machine handler function with Objectivity / DB.

```
ooTwoMachineHandlerPtr ooRegTwoMachineHandler(
     ooTwoMachineHandlerPtr twoMachineHandler);
```

Parameters *twoMachineHandler*

Pointer to the two-machine handler function to be registered.

Returns A function pointer to the previously registered two-machine handler.

Discussion You can call this function after calling `ooInit` in a single-threaded application or after you call `ooInitThread` in each thread of a multithreaded application.

A two-machine handler is relevant only in a two-machine hot-failover configuration in which a hard link between the two machines enables applications on each machine to check on the status of the other machine.

An application's two-machine handler function is called when only one of two equally weighted database images is available. The function determines whether the application can write to the available database image (typically a local image). If the function returns `oocTrue`, the application can proceed as if the local database image by itself constituted a quorum; if it returns `oocFalse`, 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 `oocFalse`.

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 database. 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 continue.

It is your responsibility to 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.

NOTE In any given network failure, if at least one application on one machine has a two-machine handler function that returns `oocTrue`, then no application on the other machine should have a two-machine handler function that returns `oocTrue`.

---

ooReplace

global macro

Creates a database with the specified name in the specified federated database, replacing (deleting) any existing database with the same name.

```
ooHandle(ooDBObj) ooReplace(
     ooDBObj,
     (char *dbSysName,
     uint32 defaultContInitPages = 0,
     uint32 defaultContPercentGrow = 0,
     const char *hostName = 0,
     const char *pathName = 0,
     uint32 weight = 1),
     const ooHandle(ooFDObj) &fdH);
```

Parameters

dbSysName

System name of the database to create. Any existing database with this system name is removed before the new database is created. *dbSysName* follows the same naming rules as files of your operating system, and must be unique within the federated database.

defaultContInitPages

Initial number of logical pages to allocate for the default container in the new database. The maximum value is 65535. If you omit this parameter or specify 0, the system default value (4) is used.

defaultContPercentGrow

Amount by which the default container may grow, expressed as a percentage of its current size. If you omit this parameter or specify 0, the system default value (10%) is used.

hostName

Name of the host system on which to create the database file. If you specify a *hostName*, you must also specify the *pathName* parameter. If you omit the *hostName* parameter or specify 0, the database file is created on the same host as the federated database's system-database file.

pathName

Full pathname of the directory in which to create the database file. If you specify a *pathName*, you must also specify the *hostName* parameter. If you omit the *pathName* parameter or specify 0, the database file is created in the same directory as the federated database's system-database file. The *pathName* you specify may, but need not, include the filename for the database file.

weight

(*DRO*) Weight of the first database image. *weight* must be an integer greater than zero. If you omit this parameter, the weight is 1.

fdH

Handle to the federated database in which to create the new database.

(*FTO*) In a partitioned federated database, the new database is created in the federated database's initial partition.

Returns

If successful, a handle to the newly created database; otherwise, a null handle.

Discussion

This macro is generally used to clean up a federated database for repeated tests.

WARNING The removed database is *not* recovered if the transaction is aborted.

Note that all parameters except the first and last correspond to parameters of the `ooDBObj` constructor.

Example

The syntax for using `ooReplace` is somewhat unusual. This example shows how to use `ooReplace` to replace a database whose system name is `testDB`. The new database is created in the same location as the federated database's system-database file.

```
ooHandle(ooFDObj) fdH;
ooHandle(ooDBObj) dbH;
... // Start transaction, open FD, initialize fdH
dbH = ooReplace(ooDBObj, ("testDB"), fdH);
```

ooResetError

global macro

Clears Objectivity / DB error flags.

```
void ooResetError();
```

Discussion

This function sets the following global variables:

- `oovLastErrorLevel = oocNoError`
- `oovLastError = 0`

In a multithreaded application, `ooResetError` resets the error flags in the current Objectivity context.

See also

Chapter 23, “Error Handling,” in the Objectivity / C++ programmer’s guide.

ooResource

global type

Type for representing information about a locked resource (typically a container).

```
struct ooResource {
    ...
};
```

Members

…

Members contain information about the locked resource, an Objectivity / DB object. These members are subject to change from release to release. See the header file `ooRecover.h` for the current definition of this structure.

Discussion

If a transaction is waiting for a lock on an Objectivity / DB object (typically a container), a structure of this type, called a *resource-information structure*, identifies the object and its lock status.

You should not create structures of this type. Instead, you obtain a resource-information structure by calling `ooGetResourceOwners`; you must include the Objectivity / C++ header file `ooRecover.h`.

ooRunStatus

global function

Prints a summary of Objectivity / DB internal statistics to `stdout`; used primarily for performance tuning.

```
void ooRunStatus();
```

Discussion

In a multithreaded application, this function obtains statistics that pertain to the current Objectivity context.

See also

Chapter 24, “Performance,” in the Objectivity / C++ programmer’s guide

ooSetAMSUsage

global function

(*administration*) Sets the application's policy for using the Advanced Multithreaded Server (AMS).

```
void ooSetAMSUsage(ooAMSUsage amsUsage = oocAMSPreferred);
```

Parameters

amsUsage

Policy for using AMS. If you omit this parameter, the application uses AMS whenever possible. See ooAMSUsage for other values you can specify.

Discussion

The system default is for an application to use AMS whenever possible (that is, the policy is implicitly `oocAMSPreferred`). If you choose to invoke this function, you should call it after `ooInit` and before any other Objectivity / DB operation. An error is signaled if you specify `oocAMSOnly` and AMS is unavailable on the host that contains the requested data. The policy set by this function applies to the entire application, including all of its threads.

ooSetErrorFile

global function

Sets the error message output file.

```
ooStatus ooSetErrorFile(char *errorFileName);
```

Parameters

errorFileName

Name of the error message output file.

Discussion

In a multithreaded application, this function sets the error message output file for the current Objectivity context.

ooSetHotMode

global function

Enables or disables *hot mode*, which controls the timing of certain internal overhead operations in the Objectivity / DB cache, for purposes of improving performance.

```
void ooSetHotMode(ooBoolean hotMode = oocTrue);
```

Parameters

hotMode

Specifies whether to enable or disable hot mode. If you omit this parameter, hot mode is enabled.

Discussion

You can call this function anytime after calling `ooInit` and as often as desired to enable or disable hot mode. In a multithreaded application, this function affects only the Objectivity / DB cache in the current Objectivity context. The system

default is for hot mode to be disabled in an Objectivity context unless you explicitly enable it.

Hot mode can improve the performance of an application that repeatedly opens, closes, and reopens persistent objects that were created by applications running on other architectures. For a discussion of performance tradeoffs, see "Use Hot Mode" in Chapter 24 of the Objectivity / C++ programmer's guide.

ooSetLargeObjectMemoryLimit

global function

Sets the maximum amount of dynamically allocated memory that is available for caching large persistent objects.

```
void ooSetLargeObjectMemoryLimit(uint32 size);
```

Parameters

size

Number of bytes in the Objectivity / DB large-object memory pool.

Discussion

The Objectivity / DB cache has a both a buffer pool and a memory pool for handling *large objects* (persistent objects that span multiple storage pages). When a large object is created or opened, Objectivity / DB:

- Reads the object's header page into the large-object buffer pool. This page contains overhead information about the object.
- Reads the object's storage pages into the large-object memory pool. These pages form a single dynamically allocated block of contiguous buffer pages pointed to by the header page.

The `ooSetLargeObjectMemoryLimit` function sets the suggested limit on the total number of bytes in the large-object memory pool. When this limit is reached, Objectivity / DB attempts to swap out the pages of closed large objects before opening additional large objects. However, if Objectivity / DB cannot find enough closed large objects to swap out, it will ignore the specified limit and allocate additional pages as needed. Thus, the limit you specify is a soft limit that affects the amount of swapping performed on behalf of large objects.

If you do not call the `ooSetLargeObjectMemoryLimit` function, the default limit on the large-object memory pool the equal to the maximum size of the large-object buffer pool (see `ooInit`).

You can call the `ooSetLargeObjectMemoryLimit` function anytime after calling `ooInit` and as often as needed to increase or decrease the limit on the large-object memory pool. In a multithreaded application, this function sets the limit for the current Objectivity context.

ooSetLockWait

global function

Sets the default lock-waiting option for a series of transactions.

```
void ooSetLockWait(int32 waitOption = oocNoWait);
```

Parameters

waitOption

Specifies whether transactions are to wait to obtain locks, and if so, for how long:

- Specify `oocNoWait` or 0 to turn off lock waiting.
- Specify `oocWait` to cause transactions to wait indefinitely for locks.
- Specify a timeout period of *n* seconds, where *n* is an integer in the range 1 <= *n* <= 14400. If *n* is less than 0 or greater than 14400, it is treated as `oocWait`.

Discussion

By default, the transactions in an application do not wait for locks. You can override the default behavior on a per-transaction basis by setting the *waitOption* parameter of the `ooTrans::start` member function. Alternatively, you can change the default behavior by using `ooSetLockWait`.

Calling `ooSetLockWait` within a transaction overrides any lock-waiting option you specified through the `ooTrans::start` member function when you started that transaction. The option you set with `ooSetLockWait` becomes the new default, which subsequent transactions can choose to use or override.

Lock waiting does not apply to MROW read transactions. Therefore, the lock-waiting option you specify through `ooSetLockWait` is ignored by such transactions.

In a multithreaded application, this function sets the default lock-waiting option for the current Objectivity context.

ooSetOfflineMode

global function

(*FTO*) Sets the offline mode for the current process.

```
ooStatus ooSetOfflineMode(ooOfflineMode offlineMode);
```

Parameters

offlineMode

Offline mode to be set. Specify `oocIgnore` to ignore the offline status of autonomous partitions. Specify `oocEnforce` to enforce the offline status.

Returns

`oocSuccess` if successful; otherwise `oocError`.

Discussion Offline mode determines whether a process ignores or enforces the offline status of all autonomous partitions. If you do not call this function, offline status is enforced.

ooSetRpcTimeout

global function

Sets how long an application is to wait for an Objectivity server to respond before signaling a timeout error.

```
void ooSetRpcTimeout(long seconds);
```

Parameters *seconds*

Number of seconds to wait before signaling a timeout error.

Discussion By default, an application waits 25 seconds for the lock server or AMS to respond to a request. However, an Objectivity server running on a busy host machine may need more time to respond. If your application consistently signals a lock server or AMS timeout error, you can call this function to increase the timeout period; alternatively, you can consider running the Objectivity server on a less congested host.

In a multithreaded application, this function sets the timeout period for the current Objectivity context.

ooSignal

global function

Signals the specified error and reports additional error information.

```
1.   ooStatus ooSignal(
       const ooErrorLevel errorLevel,
       const ooError &errorID,
       const ooHandle(ooObj) *relevantObj,
       ...);
2.   ooStatus ooSignal(
       const ooErrorLevel errorLevel,
       const ooError &errorID,
       const ooHandle(ooObj) relevantObj,
       ...);
```

Parameters *errorLevel*

Error level for this event.

errorID

Error identifier for this event. Error identifiers must be declared and initialized in an error message header file, which is included in your

application source file. An error identifier consists of an error number and an error message.

relevantObj

Handle or pointer to the handle of an Objectivity / DB object that may help the system error handler to pinpoint the situation in which the error condition occurred. Set this parameter to 0 if there is no relevant Objectivity / DB object.

...

Variable number of parameters required by the format string stored in the message part of the error identifier. These parameters provide information about the situation in which the error condition occurred. They are substituted for conversion specifications (such as %s or %d) in the format string to construct the error message.

Returns `oocSuccess` if successful; otherwise `oocError`.

Discussion `ooSignal` signals the specified error, invokes the currently registered error handler, and returns the result from the error handler. It also sets Objectivity / C++ global variables as follows:

- `oovLastErrorLevel` is set to *`errorLevel`*.
- `oovLastError` is set with a pointer to *`errorID`*.
- `oovNError` is incremented.

In a multithreaded application, this function signals an error (and invokes the registered error handler) in the current Objectivity context.

ooStartInternalLS

global function

(*IPLS*) Starts an in-process lock server within an application.

```
ooStatus ooStartInternalLS(
     const char *fdName = 0
     void (*threadFn)() = 0);
```

Parameters *`fdName`*

Pathname of the boot file of the federated database or autonomous partition that the application intends to open. This parameter controls the conditions under which an in-process lock server is started:

- If you specify 0 (the default), an in-process lock server is started whether or not it will be used by the calling application.
- If you specify a boot file, an in-process lock server is started only if the calling application will use it—that is, only if the lock server host shown

in the boot file is the current host. If the lock server host is *not* the current host, an error is signalled and an in-process lock server is not started.

threadFn

Pointer to an application-defined function with no arguments that will be called from the in-process lock server's listener thread. A listener thread is created to service lock requests from external applications.

The specified function is called after the listener thread has initialized its Objectivity context and before it begins servicing requests. The function could be used to customize the behavior of the thread, such as adjusting its priority or installing an application-defined error handler to direct how error messages originating in that thread will be handled.

Returns

`oocSuccess` if the in-process lock server was successfully started; otherwise, `oocError`—for example, if a separate lock server process is already running on the same host machine, or if some operating-system resource could not be obtained.

If *`fdName`* is nonzero, `oocSuccess` indicates the in-process lock server is started and will be used by the application; `oocError` indicates that the boot file specified by *`fdName`* cannot be opened or specifies a lock server host that is different from the current host.

Discussion

This function must be called after `ooInit` is called and before the federated database is opened for the first time.

An application cannot start an in-process lock server if another lock server process is already running on the same host. If the application chooses to continue running, it will use the lock server that is already running on that host. An application can call the `ooCheckLS` function to test for a running lock server.

When an application successfully starts an in-process lock server:

- The application becomes the lock server process for the workstation on which it is running. If a federated database names that workstation as its lock server host, all applications accessing the federated database will send their lock requests to the application running the in-process lock server. The in-process lock server uses a separate *listener thread* to service requests from external applications.
- The application uses its own in-process lock server only if the opened federated database names the application's host as the lock server host. That is, an application always uses the lock server specified by the federated database, whether or not the application is running an in-process lock server.

  You can find out whether an application will use its in-process lock server by specifying the *`fdName`* parameter to `ooStartInternalLS`.

An in-process lock server improves performance only if most or all of the lock requests for a given federated database originate from a single multithreaded application process. The in-process lock server can then coordinate locking through simple function calls instead of servicing lock requests over the network.

See also `ooCheckLS`
`ooStopInternalLS`

ooStatus

global type

General return type for Objectivity/C++ global functions and member functions.

Constants `oocSuccess`

Indicates a successful outcome. Its value is a nonzero integer.

`oocError`

Indicates that an error occurred. Its value is the integer 0.

ooStopInternalLS

global function

(*IPLS*) Shuts down an in-process lock server within an application.

```
ooStatus ooStopInternalLS(
     int wait = INT_MAX,
     ooBoolean force = oocFalse);
```

Parameters *wait*

Number of seconds to wait for active transactions to terminate. The default value, `INT_MAX`, means there is no time limit, so this function will wait indefinitely. If active transactions terminate within the specified wait period, the in-process lock server is shut down. Otherwise, if transactions are still active when the specified wait period expires, this function takes the action specified by *force*.

force

Specifies whether to shut down the in-process lock server if active transactions have not yet terminated by the end of the *wait* period:

- If you omit this parameter or specify `oocFalse`, the in-process lock server continues running and this function returns `oocError`.
- If you specify `oocTrue`, the in-process lock server is shut down, even if transactions are active, and this function returns `oocSuccess`.

Returns `oocSuccess` if the in-process lock server is successfully shut down, or if it was not running; otherwise `oocError`.

Discussion

This function safely shuts down an in-process lock server so that you can terminate the application in which it is running without harming any other applications that may be using its in-process lock server.

This function should be called at the end of the application that is running the in-process lock server, after committing or aborting any transactions, and before calling `ooExitCleanup` or `exit`.

This function causes the in-process lock server to refuse any new transactions from external client applications, and to wait for any active transactions to finish during the period specified by *`wait`*. During the wait period, the in-process lock server may accept new client connections (for example, to allow administration tools to run), but continues to disallow new transactions. When all active transactions have finished, the in-process lock server is shut down. At this point, the application may safely exit.

If active transactions do not finish and the wait period expires, this function either shuts down the in-process lock server or allows it to continue running, according to the *`force`* parameter. If the in-process lock server continues running, the application can repeat the attempted shutdown by calling this function again later.

See also `ooStartInternalLS`

ooTermThread

global function

Terminates the current thread's ability to invoke Objectivity / DB operations.

```
ooStatus ooTermThread();
```

Discussion

Threads initialized with the `ooInitThread` or `ooInit` function must call the `ooTermThread` function after completing their Objectivity / DB operations and before termination. It is not, however, necessary to call `ooTermThread` in threads that are terminated due to process termination. So, for example, the main thread need invoke `ooTermThread` only if it terminates before the process terminates.

You may not reinitialize a thread after calling `ooTermThread`. You may not invoke `ooTermThread` more than once per thread.

The `ooTermThread` function deletes the thread's current Objectivity context, unless this context is the null context. Because Objectivity contexts are usually allocated dynamically, `ooTermThread` invokes the `delete` operator to delete the current context. Consequently, if the current Objectivity context is either statically allocated or a dynamically allocated member of a larger object, you must set the current context to 0 before using `ooTermThread`, and then arrange to delete the Objectivity context as appropriate.

See also ooInitThread
ooContext::setCurrent

ooTransId

global type

Unique identifier for a transaction.

Constants oocInValidTransId

An invalid transaction identifier. This value is used as the identifier in a transaction-information structure that signals the end of an array of transaction-information structures.

Discussion Every transaction has a *transaction 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. Administration tools such as oolockmon and oolistwait display transaction identifiers in their output.

You can get a transaction identifier of an active transaction by calling the getID member function on a transaction object. Alternatively, you can obtain the transaction identifier for a particular transaction from a transaction-information structure of type ooTransInfo that describes the transaction; you must include the Objectivity/C++ header file ooRecover.h.

See also ooTrans::getID

ooTransInfo

global type

Type for representing information about a transaction.

```
struct ooTransInfo {
     ooTransId tid;
     …
};
```

Members *tid*

Transaction identifier for the described transaction.

…

Additional members contain information about the described transaction. These members are subject to change from release to release. See the header file ooRecover.h for the current definition of this structure.

Discussion A structure of this type, called a *transaction-information structure*, describes a particular transaction. You should not create structures of this type. Instead, you obtain a transaction-information structure by calling recovery functions; you must include the Objectivity/C++ header file ooRecover.h.

See also ooCleanup
ooGetActiveTrans

ooTwoMachineHandlerPtr

global type

(*DRO*) Function-pointer type for application-defined message handler functions.

```
typedef ooBoolean (*ooTwoMachineHandlerPtr)();
```

Returns oocTrue if this application can safely access a database even though only one of two equally weighted images is available; otherwise, oocFalse.

Discussion A registered two-machine handler is a function that is called when only one of two equally weighted database images is available. The function determines whether the application can write to the available database image (typically a local image). If the function returns oocTrue, the application can proceed as if the local database image by itself constituted a quorum; if it returns oocFalse, 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 oocFalse.

See also ooRegTwoMachineHandler

ooTypeN

global macro

Gets the type number of the specified class of Objectivity / DB objects.

```
ooTypeNumber ooTypeN(className);
```

Parameters *className*

Name of the class whose type number is to be returned. The specified class must be either ooObj or derived from ooObj—that is, a basic-object class, a container class, the database class (ooDBObj), the federated-database class (ooFDObj), or the autonomous-partition class (ooAPObj).

Returns Type number of the class.

Discussion Every class in the ooObj inheritance hierarchy has a unique type number that identifies the class within the federated-database schema. You can use this number to determine whether an object is an instance of a particular class.

An error is signalled if the specified class is not in the ooObj inheritance hierarchy.

Because ooTypeN is expanded into a variable name, you cannot use it as a label in a switch statement; to do so causes a compiler error. However, you can use ooTypeN as part of a conditional expression in an if-else statement.

Example

This example obtains the type number of a referenced object and tests it against the type number of several different classes:

```
ooHandle(Fruit) fruitH;
ooTypeNumber typeNum;
fruitH = ...     // Set fruitH to reference some kind of fruit
// Set typeNum to the type number of the referenced fruit
typeNum = fruitH.typeN();
if (typeNum == ooTypeN(Apple)) {
   ... // perform operation for apples
}
else if (typeNum == ooTypeN(Orange)) {
   ... // perform operation for oranges
}
else if (fruitH->ooIsKindOf(ooTypeN(Berry))) {
   ... // perform operation for berries
}
```

See also

ooTypeNumber
ooObj::ooIsKindOf
ooRefHandle(classname)::typeN

ooTypeNumber

global type

Type number of a class in the ooObj inheritance hierarchy.

Discussion

The ooObj class and every class that derives from it has a unique type number that identifies the class within the federated-database schema. You can use this number to determine whether an object is an instance of a particular class.

See also

ooTypeN
ooObj::ooIsKindOf
ooRefHandle(*classname*)::typeN

ooUpdateIndexes

global function

Explicitly updates all applicable indexes to reflect a new or modified object.

```
ooStatus ooUpdateIndexes(ooHandle(ooObj) &pHandle);
```

Parameters

pHandle

Handle to the new or modified object for which indexes are to be updated.

Returns `oocSuccess` if successful; otherwise `oocError`.

Discussion Each transaction has an index mode that determines when indexes are to be updated for objects that are created or changed during the transaction. If a transaction's index mode is set to `oocExplicitUpdate`, indexes are updated only when you call the `ooUpdateIndexes` function. You must call this function once for each indexed object that is created or modified during the transaction—for example, after you have created a new object of an indexed class and initialized all its key fields, or after you have changed any of an existing object's key-field values. Explicit updates are recommended for update-intensive applications that use indexes over database and federated-database scopes.

See also `ooIndexMode`

ooUseIndex

global function

Enables or disables the use of indexes during a predicate query.

```
void ooUseIndex(ooBoolean useIndex = oocTrue);
```

Parameters *useIndex*

Specifies whether the `scan` member function of an iterator can use indexes during a predicate query. Specify `oocTrue` to use indexes during scan; specify `oocFalse` to disable the use of indexes.

Discussion Indexes are used in the optimization of certain kinds of predicate queries. Disabling the use of indexes may be desirable when:

- A query is going to scan the entire range of the type and sorting is not necessary. In such a case, indexes do not speed up your query.
- The data to be scanned has been created or modified since the last time indexes were updated.

If you do not call this function, the use of indexes is disabled. In a multithreaded application, this function controls the use of indexes in transactions in the current Objectivity context.

ooUserDefinedOperators

global variable

Operator set consulted by the predicate query mechanism to resolve application-defined relational operators used in a predicate.

```
ooOperatorSet *ooUserDefinedOperators;
```

Discussion When an application defines its own relational operators, it registers them with an operator set. The operator set must be assigned to this variable to make the

registered relational operators available to the Objectivity/DB predicate query mechanism. By default, this variable points to an initial operator set that is created by Objectivity/DB when the application starts.

In a multithreaded application, the operator set to which this variable refers may be consulted for predicate queries in any Objectivity context. This variable is not thread-safe, however, so your application must ensure that:

- Only one thread updates the global operator set at a time.
- If a thread is updating the global operator set, no other thread can be making a predicate query at the same time.

See also `ooQueryOperatorPtr` function-pointer type
`ooOperatorSet` class

ooVersMode

global type

Type for representing the versioning behavior of a basic object.

Constants

`oocNoVers`

The versioning behavior of the object is disabled—that is, no new version can be created from it.

`oocLinearVers`

Linear versioning is enabled, which allows exactly one new version to be created from the object. When a new version is created, the versioning status of the original object is automatically set to `oocNoVers` so no further versions can be made from it. The new version's versioning status is `oocLinearVers`, allowing it to be the source of the next new version.

`oocBranchVers`

Branch versioning is enabled, which allows any number of new versions to be created from the object.

See also *`ooRefHandle`*`(ooObj)::setVersStatus`

oovLastError

context variable

Pointer to the error identifier structure for the most recent error condition.

```
ooError *oovLastError;
```

Discussion In a multithreaded application, this variable refers to the most recent error condition in the current Objectivity context.

oovLastErrorLevel

context variable

Severity level of the most recent error condition.

```
ooErrorLevel oovLastErrorLevel;
```

Discussion

In a multithreaded application, this variable refers to the severity level of the most recent error condition in the current Objectivity context.

oovNError

context variable

Count of the total number of errors (not including warnings) that have occurred so far.

```
uint32 oovNError;
```

Discussion

In a multithreaded application, this variable refers to the number of errors that have occurred in the current Objectivity context.

ooVoidFuncPtr

global type

Function-pointer type for a function that has no parameters and that returns no result.

```
typedef void (*ooVoidFuncPtr)(void);
```

uint8

global type

Objectivity / C++ 8-bit unsigned integer type. Values of this primitive type may range from 0 to +255. This type is portable across all architectures supported by Objectivity / C++. This type is also called `ooUInt8`.

uint16

global type

Objectivity / C++ 16-bit unsigned integer type. Values of this primitive type may range from 0 to +65,535. This type is portable across all architectures supported by Objectivity / C++. This type is also called `ooUInt16`.

uint32

global type

Objectivity / C++ 32-bit unsigned integer type. Values of this primitive type may range from 0 to +4,294,967,295. This type is portable across all architectures supported by Objectivity / C++. This type is also called `ooUInt32`.

uint64

global type

Objectivity / C++ 64-bit unsigned integer type. Values of this primitive type may range from 0 to $+2^{64}-1$. This type is portable across all architectures supported by Objectivity / C++. This type is also called `ooUInt64`.

Wait Options

Integer constants that specify whether to wait for locks when starting a transaction.

Constants

`oocNoWait`

Turns off lock waiting for a transaction.

`oocTransNoWait`

Causes a transaction to use the default lock-waiting option currently in effect for the Objectivity context.

`oocWait`

Causes a transaction to wait indefinitely for locks.

See also

`ooSetLockWait`
`ooTrans::start`

appClass Class

Inheritance: **ooObj->*appClass***

Inheritance: **ooObj->ooContObj->*appClass***

Handle Class: `ooHandle(appClass)`

Object-Reference Class: `ooRef(appClass)`

Every application-defined persistence-capable class *appClass* represents a particular kind of basic object or container. *appClass* and its corresponding handle and object-reference classes together define the behavior for *appClass* instances.

See:

- "Reference Summary" on page 84 for an overview of the member functions generated for *appClass*
- "Reference Index" on page 85 for a list of the generated member functions

For operations performed through a handle or object reference, see:

- "Reference Summary" on page 478

About Application-Defined Classes

An application that wants to save objects in a federated database must define a persistence-capable class *appClass* for each kind of object to be saved. An application can define its own:

- Basic-object classes, which are derived from `ooObj` (but not from `ooContObj`).
- Container classes, which are derived from `ooContObj`.

Every *appClass* normally defines data members representing application-specific attributes and associations, as well as member functions that provide application-specific behavior. In addition, an *appClass* can optionally override various member functions inherited from ooObj to customize behavior:

- Basic-object and container classes can override ooValidate if they need a way to perform tests that validate instances.
- Basic-object classes can override ooCopyInit, ooPreMoveInit, ooPostMoveInit, or ooVersInit if they need to customize copy, move, or versioning operations.

Every *appClass* is defined using the Objectivity/C++ Data Definition Language (DDL), and is added to the federated-database schema by the DDL processor. Besides updating the schema, the DDL processor produces C++ header and implementation files containing:

- Classes for working with *appClass* instances—specifically, a handle class (ooHandle(*appClass*)), an object-reference class (ooRef(*appClass*)), and an iterator class (ooItr(*appClass*)). If *appClass* is a basic-object class, a short object-reference class (ooShortRef(*appClass*)) is generated as well.
- The definition of *appClass* with the following additions:
  - Member functions that redefine some of the functions inherited from ooObj or ooContObj (see "Redefinitions of Inherited Member Functions" on page 82).
  - Member functions for creating, deleting, and accessing each association defined by the class (see "Association Member Functions" on page 83).

An application must include the DDL-generated header files before it can use the generated classes and functions; the application must also be compiled with the corresponding implementation file. See the Objectivity/C++ Data Definition Language book for complete information about persistence-capable class definitions, the DDL processor, and the generated files.

Redefinitions of Inherited Member Functions

The DDL processor generates definitions in *appClass* for various member functions that are defined in ooObj:

- The function ooThis is redefined with *appClass*-specific parameter types.
- The virtual functions ooGetTypeN, ooGetTypeName, and ooIsKindOf are overridden for internal purposes; their interfaces and behavior are unchanged, so they can be regarded as inherited from ooObj.
- The functions operator new and operator delete are overloaded for internal purposes; their interfaces and behavior are unchanged, so they can be regarded as inherited from ooObj.

If *appClass* is a container class, the container-specific variants of `operator new` are also overloaded for internal purposes and can be regarded as inherited from in `ooContObj`.

appClass also inherits member functions that are not redefined by generated definitions:

- If *appClass* is a basic-object or container class, it inherits the `ooUpdate` member function, and can optionally override the inherited virtual member function `ooValidate`.
- If *appClass* is a basic-object class, it inherits the member functions for managing advanced versioning associations, and can optionally override the inherited virtual member functions `ooCopyInit`, `ooPreMoveInit`, `ooPostMoveInit`, or `ooVersInit`.

Association Member Functions

The DDL processor generates a set of member functions in *appClass* for each association defined in the class. These member functions enable you to create, delete, and access the corresponding association. The generated member-function names are constructed from the association's unique name. Various parameters of the generated functions accept object references, handles, or iterators to objects of the association's destination class.

A slightly different set of member functions is generated, depending on whether the association is *to-one* (that is, one-to-one or many-to-one) or *to-many* (that is, one-to-many or many-to-many).

The descriptions in this chapter use the following conventions:

linkName	Name of an association defined in *appClass*.
className	Name of the destination class specified in the *linkName* association definition—the class whose instances are to be associated with instances of *appClass*.

Working With *appClass* Instances

An application works with instances of *appClass* as described in "Working With Basic Objects" on page 432 and "Working With Containers" on page 209. In brief, after the application includes the appropriate DDL-generated header files, it can:

- Create a new persistent instance using the application-defined (or default) *appClass* constructor and the inherited `operator new`.

- Reference an instance of *appClass* through a handle of class ooHandle(*appClass*) or an object reference of class ooRef(*appClass*).
- Perform Objectivity / DB operations on a referenced instance of *appClass* by calling various member functions on the referencing handle or object reference.
- Perform various operations on a referenced instance of *appClass* by accessing the instance's defined, generated, or inherited members. These members can be accessed directly from within a member function of *appClass* or indirectly through the indirect member-access operator (->) on an *appClass* handle or object reference.

Reference Summary

The following table lists the member functions generated by the DDL processor in the definition of each application-defined persistence-capable class *appClass*. Member functions indicated as *(inherited)* are redefined in *appClass* with no change in behavior. They are documented with the base class, along with the other inherited member functions not listed here; see the ooObj class (page 431) or the ooContObj class (page 207).

Creating and Deleting an *appClass* Object	operator new *(inherited)* operator new *(inherited, containers only)* operator delete *(inherited)*
Managing To-One Associations	*linkName* exist_*linkName* set_*linkName* del_*linkName*
Managing To-Many Associations	*linkName* exist_*linkName* add_*linkName* sub_*linkName* del_*linkName*
Working With This *appClass* Object	ooGetTypeN *(inherited)* ooGetTypeName *(inherited)* ooIsKindOf *(inherited)* ooThis

Reference Index

Generated Member Functions

add_*linkName*

Creates a *linkName* association from this object to the specified destination object.

```
ooStatus add_linkName(const ooHandle(className) &objH);
```

Parameters

objH

Handle to the destination object to be linked to this object. The destination object must be an instance of the destination class *className* or any of its derived classes. If *objH* is a null handle, no action is performed.

Returns

oocSuccess if successful; otherwise oocError.

Discussion

This member function is generated for the to-many association *linkName* defined by this object's class.

The application must be able to obtain an update lock for this object.

If *linkName* is a bidirectional association, this operation also creates the inverse association from the specified destination object to this object. In this case, the application must be able to obtain update locks on both objects.

No error is signaled if a *linkName* association already exists between this object and the specified destination object. That is, you can create duplicate associations between the two objects (even though it could be semantically meaningless to do so).

del_*linkName*

Deletes all *linkName* associations that exist for this object.

```
ooStatus del_linkName();
```

Returns `oocSuccess` if successful; otherwise `oocError`.

Discussion This member function is generated for the to-one or to-many association *linkName* defined by this object's class.

The application must be able to obtain an update lock for this object.

If *linkName* is a bidirectional association, this operation also deletes the inverse association to this object from each of the former destination objects. In this case, the application must be able to obtain update locks on all of the affected objects.

You should call the `exist_`*linkName* member function to test whether *linkName* associations exist before you try to delete them.

exist_*linkName*

Tests whether this object has a *linkName* association to the specified destination object.

```
ooBoolean exist_linkName(
     const ooHandle(className) &objH) const;
```

Parameters *objH*

Handle to the destination object to be tested for association. The destination object must be an instance of the destination class *className* or any of its derived classes. You can specify 0 to test whether this object has any *linkName* association(s) at all.

Returns `oocTrue` if an association exists, otherwise `oocFalse`.

Discussion This member function is generated for the to-one or to-many association *linkName* defined by this object's class.

linkName

Finds, and optionally opens, the destination object linked to this object by the to-one association *linkName*.

```
1.  ooHandle(className) linkName(
      const ooMode openMode = oocNoOpen) const;
2.  ooRef(className) &linkName(
      ooRef(className) &object,
      const ooMode openMode = oocNoOpen) const;
3.  ooHandle(className) &linkName(
      ooHandle(className) &object,
      const ooMode openMode = oocNoOpen) const;
```

Parameters

openMode

Intended level of access to the destination object:

- Specify `oocNoOpen` (the default) to set the returned object reference or handle to the destination object without opening it.
- Specify `oocRead` to open the destination object for read.
- Specify `oocUpdate` to open the destination object for update.

object

Object reference or handle to be set to the destination object, which is an instance of the destination class *className* or one of its derived classes.

Returns

Object reference or handle to the destination object. A null handle is returned if no *linkName* association exists.

Discussion

This member function is generated for the to-one association *linkName* defined by this object's class.

When called without an *object* parameter, this member function allocates a new *className* handle and returns it. Otherwise, this member function returns the object reference or handle that is passed to it.

linkName

Initializes an iterator to find, and optionally open, all destination objects that are linked to this object by the to-many association *linkName*, and that satisfy any specified selection criteria.

```
1.  ooStatus linkName(
      ooItr(className) &iterator,
      const ooMode openMode = oocNoOpen) const;
```

```
2.  ooStatus linkName(
      ooItr(className) &iterator,
      const char *predicate);

3.  ooStatus linkName(
      ooItr(className) &iterator,
      const ooMode openMode,
      const ooAccessMode access,
      const char *predicate);
```

Parameters

iterator

Iterator for finding the destination objects, which are instances of the destination class *className* or any of its derived classes.

openMode

Intended level of access to each destination object found by the iterator's `next` member function:

- `oocNoOpen` (the default in variant 1) causes `next` to set the iterator to the next destination object without opening it.
- `oocRead` causes `next` to open the next destination object for read.
- `oocUpdate` causes `next` to open the next destination object for update.

predicate

String expression in predicate query language. Specifies the condition to be met by the found destination objects. The iterator is initialized to find only those destination objects that match *predicate*.

access

Level of access control of the data members that *predicate* can test:

- Specify `oocPublic` to permit the predicate to test only public data members, preserving encapsulation.
- Specify `oocAll` to permit the predicate to test any data member. To preserve encapsulation, you should use this mode only within member functions of the class you are querying.

Returns

`oocSuccess` if successful; otherwise `oocError`.

Discussion

This member function is generated for the to-many association *linkName* defined by this object's class.

Variant 2 finds destination objects without opening them; the other variants may open the found objects, as specified by the *openMode* parameter.

If no *linkName* associations currently exist from this object, the iterator is set to null, and the *linkName* member function returns `oocSuccess`. (The iterator's `next` member function will return `oocFalse`, however.)

An error is signaled if *predicate* tries to test a non-existent data member, or if *predicate* tries to test a protected or private data member when the *access* parameter is ooPublic.

The *linkName* member function does *not* use indexes to optimize a search, even if the *predicate* parameter is specified.

ooThis

Sets an object reference or handle to reference this *appClass* object.

```
1.  ooHandle(appClass) ooThis() const;
2.  ooRef(appClass) &ooThis(
      ooRef(appClass) &object) const;
3.  ooHandle(appClass) &ooThis(
      ooHandle(appClass) &object) const;
```

Parameters

object

Object reference or handle to be set to this object.

Returns

Object reference or handle to this object.

Discussion

You normally use ooThis in a member function of *appClass*; when such a member function is called on an *appClass* object, ooThis provides the member function with an *appClass* handle or object reference to the object. The member function can then perform operations on the object that are available only through a handle or object reference.

When called without an *object* parameter, ooThis allocates a new handle and returns it. Otherwise, ooThis returns the object reference or handle that is passed to it.

Calling ooThis in a member function of a persistence-capable class serves the same purpose as using the C++ this keyword in a member function of a non-persistence-capable class—both ooThis and this enable you to access the object on which the member function is called. However, ooThis and this differ in the following ways:

- The this keyword is the pointer to the object. Syntactically, the this keyword is a name, and can be used in expressions such as this->get();
- The ooThis member function returns an object reference or handle to the object. Syntactically, ooThis is a function and can be used in expressions such as ooThis()->get();

Member functions of persistence-capable classes should use `ooThis` (and not `this`) because the only safe way to operate on a persistent object is through an object reference or handle.

WARNING `ooThis` is the only safe way to obtain an object reference or handle within a member function; do *not* attempt to initialize an object reference or handle by assigning the `this` pointer to it.

---

set_*linkName*

Creates a *linkName* association from this object to the specified destination object.

```
ooStatus set_linkName(
     const ooHandle(className) &objH);
```

Parameters

objH

Handle to the destination object to be associated. The destination object must be an instance of the destination class *className* or any of its derived classes. If *objH* is a null handle, any existing *linkName* association from this object is deleted.

Returns

`oocSuccess` if successful; otherwise `oocError`.

Discussion

This member function is generated for the to-one association *linkName* defined by this object's class.

The application must be able to obtain an update lock for this object.

If *linkName* is a bidirectional association, this operation also creates the inverse association from the specified destination object to this object. In this case, the application must be able to obtain update locks on both objects.

Because this member function creates to-one associations, an error is signaled if this object already has a *linkName* association to some destination object.

sub_*linkName*

Deletes one or more *linkName* associations from this object to the specified destination object.

```
1.   ooStatus sub_linkName(
       const ooHandle(className) &objH);
```

```
2.  ooStatus sub_linkName(
      const ooHandle(className) &objH,
      const unit32 number = 1);
```

Parameters

objH

Handle to the destination object. The destination object must be an instance of the destination class *className* or any of its derived classes. If *objH* is a null handle, no action is performed.

number

Number of *linkName* associations to delete between this object and the destination object:

- If you specify 0, all such associations are deleted.
- If you specify 1 (the default), the first or only such association is deleted.
- If you specify a number greater than 1, this member function deletes the first *number* associations encountered.

Do not use this parameter if *linkName* is a one-to-many bidirectional association.

Returns

`oocSuccess` if successful; otherwise `oocError`. `oocSuccess` is returned even if no association exists to be deleted or if *number* exceeds the number of existing associations. `oocError` results from internal errors or locking errors.

Discussion

Only a single variant of this member function is generated for any given *linkName* association:

- Variant 1 is generated if *linkName* is a one-to-many bidirectional association. Only one *linkName* association can exist from the source object to a given destination object, so there is no need for the second parameter (*number*).
- Variant 2 is generated if *linkName* is a one-to-many unidirectional association and for a many-to-many bidirectional association. It is possible for multiple *linkName* associations to exist from the source object to a given destination object. In that situation, you can specify the *number* parameter to delete some or all of those associations.

The application must be able to obtain an update lock for this object. If *linkName* is a bidirectional association, this operation also deletes the inverse association(s) from the specified destination object to this object. In this case, the application must be able to obtain update locks on both objects.

See also

del_*linkName*

d_Database Class

Inheritance: **d_Database**

The non-persistence-capable ODMG class `d_Database` represents an *ODMG database*, which is the ODMG equivalent of an Objectivity / DB federated database.

See:

- "Reference Summary" on page 94 for an overview of member functions
- "Reference Index" on page 94 for a list of member functions

To add this and other ODMG types and definitions to your application, you must run the DDL processor and your C++ compiler with the `-DOO_ODMG` flag.

About ODMG Databases

In the storage hierarchy, an ODMG database is equivalent to an *Objectivity/DB federated database*. Therefore, an ODMG-compliant application uses an instance of `d_Database` to refer to and manipulate the top-level storage object, whereas a standard Objectivity / C++ application uses a federated-database handle (an instance of *`ooRefHandle`*`(ooFDObj)`).

Although analogous in purpose, `d_Database` and *`ooRefHandle`*`(ooFDObj)` define different behavior. As specified by ODMG, `d_Database` member functions enable you to open, close, and find objects named in the scope of an ODMG database (Objectivity / DB federated database). See the documentation for `ooRefHandle(ooFDObj)` for a list of operations on federated-database handles.

NOTE Do not confuse an *ODMG database* with an *Objectivity/DB database*; they refer to different storage levels.

The ODMG standard has no storage level equivalent to an Objectivity / DB database. Consequently, a strictly ODMG-compliant application should avoid manipulating Objectivity / DB databases explicitly. To do this:

1. Before you start your application, create a default Objectivity / DB database:
   a. Use the `oonewdb` tool create an Objectivity / DB database.
   b. Set the system name of the new Objectivity / DB database as the value of the environment variable `OO_DB_NAME`.
2. In your application, create persistent objects by calling `operator new` with a `d_Database` object as the clustering directive. This directive causes new objects to be placed in the Objectivity / DB database specified by the `OO_DB_NAME` environment variable.

NOTE If the `OO_DB_NAME` environment variable is not set, an Objectivity / DB database named `default_odmg_db` is automatically created and used.

Reference Summary

Accessing an ODMG Database	`access_status`
	`close`
	`open`

Working With Scope Names	`get_object_name`
	`lookup_object`
	`rename_object`
	`set_object_name`

Creating Transient Objects	`transient_memory`

Reference Index

lookup_object	(ODMG) Finds the persistent object with the specified name in the scope of this ODMG database.
open	(ODMG) Opens an ODMG database.
rename_object	(ODMG) Changes the name of a persistent object within the scope of this ODMG database.
set_object_name	(ODMG) Assigns a scope name to a persistent object within the scope of this ODMG database.
transient_memory	(ODMG) Clustering directive for creating a transient object from a persistence-capable class.

Types and Constants

access_status

(*ODMG*) Type specifying the intended level of access to an ODMG database.

Constants

`not_open`

Equal to the constant `oocNoOpen` of the Objectivity / C++ global type `ooMode`.

`read_write`

Equal to the constant `oocUpdate` of the Objectivity / C++ global type `ooMode`.

`read_only`

Equal to the constant `oocRead` of the Objectivity / C++ global type `ooMode`.

`exclusive`

Reserved for future development.

Discussion

This type corresponds to the global type `ooMode`.

transient_memory

(*ODMG*) Clustering directive for creating a transient object from a persistence-capable class.

```
static const d_Database *const transient_memory;
```

Discussion

This static data member is set to 0 by Objectivity / DB. You can specify it as a clustering directive to `operator new` to create a transient object from a persistence-capable class.

Member Functions

close

(*ODMG*) Closes an ODMG database.

```
void close();
```

Discussion This member function calls the `close` member function on a federated-database handle.

get_object_name

(*ODMG*) Gets the name defined in the scope of this ODMG database for the specified object.

```
const char *get_object_name(const d_Ref_Any &object) const;
```

Parameters *object*

An ODMG generic reference to the object whose scope name you want to get.

Returns Pointer to a string containing the scope name. If the referenced object does not have a scope name in the specified scope, the returned pointer is null.

Discussion The string is statically allocated by the member function and overwritten with each invocation. You should make a local copy of the returned string if you intend to use it later in the application.

The application must be able to get a read lock on the hashed container used by this ODMG database for storing scope names.

lookup_object

(*ODMG*) Finds the persistent object with the specified name in the scope of this ODMG database.

```
d_Ref_Any lookup_object(const char *name) const;
```

Parameters *name*

Scope name to look up in this ODMG database.

Returns An ODMG generic reference to the found object; a null generic reference is returned if no object with the specified scope name is found in the ODMG database.

Discussion The application must be able to obtain a read lock on the hashed container used by the ODMG database for storing scope names.

open

(*ODMG*) Opens an ODMG database.

```
void open(
     const char *databaseName,
     access_status status = read_write);
```

Parameters *databaseName*

Path to the boot file of the ODMG database to be opened. You can omit this parameter if you set the `OO_FD_BOOT` environment variable to the path. You can specify this path with or without a host name. If you specify it as a host path, use the format *`host`*`::`*`path`*.

status

Intended level of access to the ODMG database:

- Specify `read_write` (the default) to open the ODMG database for update and to designate the transaction as an update transaction.
- Specify `read_only` to open the ODMG database for read and to designate the transaction as a read transaction.

Discussion This member function calls the open member function on a federated-database handle (*without* performing local automatic recovery).

rename_object

(*ODMG*) Changes the name of a persistent object within the scope of this ODMG database.

```
void rename_object(
     const char *oldName,
     const char *newName);
```

Parameters *oldName*

Scope name of the object to be renamed.

newName

New scope name to assign. This name:

- Must be a null-terminated string that can contain any nonnull character.
- Must be unique within the name scope defined by this ODMG database.
- May contain up to 500 characters.

set_object_name

(*ODMG*) Assigns a scope name to a persistent object within the scope of this ODMG database.

```
void set_object_name(
     const d_Ref_Any &object,
     const char *name);
```

Parameters

object

An ODMG generic reference to the object to be named.

name

Scope name to assign. This name:

- Must be a null-terminated string that can contain any nonnull character.
- Must be unique within the name scope defined by this ODMG database.
- May contain up to 500 characters.

d_Date Class

Inheritance: **d_Date**

The non-persistence-capable ODMG class `d_Date` represents a *date*, which consists of three components—a year, a month, and a day.

See:

- "Reference Summary" on page 99 for an overview of member functions
- "Reference Index" on page 100 for a list of member functions

To use this class, your application must include the `ooTime.h` header file. No extra linking is required.

Reference Summary

Creating a Date	`d_Date`
Date Components	`Month` `Weekday`

Setting a Date	`next`
	`operator=`
	`previous`

Getting Information	current day day_of_week day_of_year days_in_month days_in_year is_between is_leap_year is_valid_date month year
Comparing Dates	::operator== ::operator!= ::operator< ::operator<= ::operator> ::operator>=
Arithmetic Operations on Dates	::operator+ operator++ operator+= ::operator- operator-- operator-=
Determining Whether Time Periods Overlap	::overlaps

Reference Index

current	(ODMG) Returns the current date.
day	(ODMG) Returns this date's day of the month.
day_of_week	(ODMG) Returns this date's day of the week.
day_of_year	(ODMG) Returns this date's day of the year.
days_in_month	(ODMG) Returns the number of days in this date's month or in the specified month.
days_in_year	(ODMG) Returns the number of days in this date's year or in the specified year.
d_Date	(ODMG) Default constructor that constructs a new date set to the current date.

d_Date	(ODMG) Constructs a new date from the specified date or timestamp.
d_Date	(ODMG) Constructs a new date from the specified information.
is_between	(ODMG) Returns 1 if this date is within the specified period.
is_leap_year	(ODMG) Returns 1 if this date's year or the specified year is a leap year.
is_valid_date	(ODMG) Returns 1 if the specified components make a valid date.
Month	(ODMG) Enumerated type for the months of the year.
month	(ODMG) Returns this date's month.
next	(ODMG) Advances this date to the next occurrence of the specified day of type `Weekday`.
::operator+	(ODMG) Addition operator; allocates a new date that is set to the sum of the specified date and interval of days.
operator++	(ODMG) Increment operator; increments this date by a day.
operator+=	(ODMG) Increment operator; adds the specified interval or number of days to this date.
::operator-	(ODMG) Subtraction operator; allocates a new date that is set to the specified date minus the specified interval of days.
operator--	(ODMG) Decrement operator; decrements this date by a day.
operator-=	(ODMG) Decrement operator; subtracts the specified interval or number of days from this date.
operator=	(ODMG) Assignment operator; sets this date to the specified date or timestamp.
::operator==	(ODMG) Equality operator; returns 1 if every component of one date matches the corresponding component of the other.
::operator!=	(ODMG) Inequality operator; returns 1 if any component of one date differs from the corresponding component of the other.
::operator<	(ODMG) Less-than operator; returns 1 if one date is less than another.
::operator<=	(ODMG) Less-than-or-equal-to operator; returns 1 if one date is less than or equal to another.
::operator>	(ODMG) Greater-than operator; returns 1 if one date is greater than another.

::operator>=	(ODMG) Greater-than-or-equal-to operator; returns 1 if one date is greater than or equal to another.
::overlaps	(ODMG) Returns 1 if two time periods overlap, where one or both periods are specified using dates.
previous	(ODMG) Moves this date to the previous occurrence of the specified day of type Weekday.
Weekday	(ODMG) Enumerated type for the days of the week.
year	(ODMG) Returns this date's year.

Types

Month

(*ODMG*) Enumerated type for the months of the year.

```
enum Month {
     January = 1,
     February = 2,
     March = 3,
     April = 4,
     May = 5,
     June = 6,
     July = 7,
     August = 8,
     September = 9,
     October = 10,
     November = 11,
     December = 12};
```

Weekday

(*ODMG*) Enumerated type for the days of the week.

```
enum Weekday {
     Sunday = 0,
     Monday = 1,
     Tuesday = 2,
     Wednesday = 3,
     Thursday = 4,
     Friday = 5,
     Saturday = 6};
```

Constructors

d_Date

(*ODMG*) Default constructor that constructs a new date set to the current date.

```
d_Date();
```

d_Date

(*ODMG*) Constructs a new date from the specified date or timestamp.

```
1.   d_Date(const d_Date &date);
2.   d_Date(const d_Timestamp &timeStamp);
```

d_Date

(*ODMG*) Constructs a new date from the specified information.

```
1.   d_Date(
       unsigned short year,
       unsigned short dayOfYear);
2.   d_Date(
       unsigned short year,
       unsigned short month = 1,
       unsigned short day = 1);
```

Parameters

year

Year component of the new date.

dayOfYear

Sequence number of a day in a year. The maximum value is 365 (366 in a leap year). This value is used to calculate the month and day of the new date.

month

Month component of the new date. The maximum value is 12.

day

Day component of the new date. The maximum value is the number of days in the specified month for the specified year.

Operators

::operator+ global function

(*ODMG*) Addition operator; allocates a new date that is set to the sum of the specified date and interval of days.

```
1.   d_Date ::operator+(
       const d_Date &left,
       const d_Interval &right);
2.   d_Date ::operator+(
       const d_Interval &left,
       const d_Date &right);
```

WARNING Addition that causes the year component to exceed 65535 results in an invalid date.

operator++

(*ODMG*) Increment operator; increments this date by a day.

```
1.   d_Date &operator++();
2.   d_Date operator++(int n);
```

Parameters *n*

This parameter is not used in calling this operator; its presence in the function declaration specifies a postfix operator.

Returns (Variant 1) This date, incremented by a day. (Variant 2) A new date whose value is this date, before this date is incremented.

Discussion Variant 1 is the prefix increment operator, which increments this date and then returns it. Variant 2 is the postfix increment operator, which allocates and returns a new date set to this date, and then increments this date.

Example

```
d_Date d;
++d;            // Prefix
d++;            // Postfix
```

operator+=

(*ODMG*) Increment operator; adds the specified interval or number of days to this date.

```
1.   d_Date &operator+=(const d_Interval &);
2.   d_Date &operator+=(int ndays);
```

WARNING Addition that causes the year component to exceed 65535 results in an invalid date.

::operator- global function

(*ODMG*) Subtraction operator; allocates a new date that is set to the specified date minus the specified interval of days.

```
d_Date ::operator-(
     const d_Date &left,
     const d_Interval &right);
```

operator--

(*ODMG*) Decrement operator; decrements this date by a day.

```
1.   d_Date &operator--();
2.   d_Date operator--(int);
```

Returns (Variant 1) This date, decremented by a day. (Variant 2) A new date whose value is this date, before this date is decremented.

Discussion Variant 1 is the prefix decrement operator, which decrements this date and then returns it. Variant 2 is the postfix decrement operator, which allocates and returns a new date set to this date, and then decrements this date.

operator-=

(*ODMG*) Decrement operator; subtracts the specified interval or number of days from this date.

```
1.   d_Date &operator-=(const d_Interval &);
2.   d_Date &operator-=(int ndays);
```

operator=

(*ODMG*) Assignment operator; sets this date to the specified date or timestamp.

```
1.   d_Date &operator=(const d_Date &date);
2.   d_Date &operator =(const d_Timestamp &timeStamp);
```

::operator== global function

(*ODMG*) Equality operator; returns 1 if every component of one date matches the corresponding component of the other.

```
int ::operator==(
     const d_Date &left,
     const d_Date &right);
```

::operator!= global function

(*ODMG*) Inequality operator; returns 1 if any component of one date differs from the corresponding component of the other.

```
int ::operator!= (
     const d_Date &left,
     const d_Date &right);
```

::operator< global function

(*ODMG*) Less-than operator; returns 1 if one date is less than another.

```
int ::operator< (
     const d_Date &left,
     const d_Date &right);
```

::operator<= global function

(*ODMG*) Less-than-or-equal-to operator; returns 1 if one date is less than or equal to another.

```
int ::operator<=(
     const d_Date &left,
     const d_Date &right);
```

::operator> global function

(*ODMG*) Greater-than operator; returns 1 if one date is greater than another.

```
int ::operator> (
     const d_Date &left,
     const d_Date &right);
```

::operator>= global function

(*ODMG*) Greater-than-or-equal-to operator; returns 1 if one date is greater than or equal to another.

```
int ::operator>=(
     const d_Date &left,
     const d_Date &right);
```

Member Functions

current

(*ODMG*) Returns the current date.

```
static d_Date current();
```

day

(*ODMG*) Returns this date's day of the month.

```
unsigned short day() const;
```

day_of_week

(*ODMG*) Returns this date's day of the week.

```
Weekday day_of_week() const;
```

day_of_year

(*ODMG*) Returns this date's day of the year.

```
unsigned short day_of_year() const;
```

days_in_month

(*ODMG*) Returns the number of days in this date's month or in the specified month.

```
1.  unsigned int days_in_month() const;
2.  static unsigned int days_in_month(
      unsigned short year,
      unsigned short month);
```

days_in_year

(*ODMG*) Returns the number of days in this date's year or in the specified year.

```
1.  unsigned int days_in_year() const;
2.  static unsigned int days_in_year(unsigned short year);
```

is_between

(*ODMG*) Returns 1 if this date is within the specified period.

```
int is_between(
     const d_Date &date,
     const d_Date &date) const;
```

is_leap_year

(*ODMG*) Returns 1 if this date's year or the specified year is a leap year.

```
1.  int is_leap_year() const;
2.  static int is_leap_year(unsigned short year);
```

is_valid_date

(*ODMG*) Returns 1 if the specified components make a valid date.

```
static int is_valid_date(
     unsigned short year,
     unsigned short month,
     unsigned short day);
```

month

(*ODMG*) Returns this date's month.

```
Month month() const;
```

next

(*ODMG*) Advances this date to the next occurrence of the specified day of type `Weekday`.

```
d_Date &next(Weekday day);
```

previous

(*ODMG*) Moves this date to the previous occurrence of the specified day of type `Weekday`.

```
d_Date &previous(Weekday day);
```

year

(*ODMG*) Returns this date's year.

```
unsigned short year() const;
```

Related Global Functions

::overlaps

(*ODMG*) Returns 1 if two time periods overlap, where one or both periods are specified using dates.

```
1.   int ::overlaps(
       const d_Date &startLeft,
       const d_Date &endLeft,
       const d_Date &startRight,
       const d_Date &endRight);
2.   int ::overlaps(
       const d_Timestamp &startLeft,
       const d_Timestamp &endLeft,
       const d_Date &startRight,
       const d_Date &endRight);
3.   int ::overlaps(
       const d_Date &startLeft,
       const d_Date &endLeft,
       const d_Timestamp &startRight,
       const d_Timestamp &endRight);
```

Discussion

Each time period is specified by a start and end time. You can specify the time periods using dates (variant 1), or using dates and timestamps (variants 2 and 3).

d_Interval Class

Inheritance: **d_Interval**

The non-persistence-capable ODMG time class `d_Interval` represents an *interval*, or duration of time, conforming to the day-time interval defined in the SQL standard.

See:

- "Reference Summary" on page 112 for an overview of member functions
- "Reference Index" on page 113 for a list of member functions

To use this class, your application must include the `ooTime.h` header file. No extra linking is required.

About d_Interval

A `d_Interval` object consists of four time components—days, hours, minutes, and seconds. The size of these components depends on the total duration to be represented:

- For intervals of less than a day, the `d_Interval` class accepts nonnormalized input, and then normalizes the time components when they are accessed. For example, you can construct an interval of 68 minutes; calling the `hour` function returns a value of 1, and calling the minute function returns a value of 8.
- For intervals of greater than a day, the `d_Interval` class requires that you partially normalize the input time components to specify an integer number of days, plus some (nonnormalized) number of hours, minutes, and seconds.

  For example, instead of specifying 28 hours, you could specify 1 day, 2 hours, and 120 minutes, which would be normalized on access to 1 day and 4 hours.

The Objectivity/C++ implementation of the `d_Interval` class stores each time component as an integer number of milliseconds. Within a `d_Interval` object:

- The total number of milliseconds for all time components must not exceed 2,147,483,648.
- The total number of milliseconds for the hour, day, and seconds time components must not exceed 86,400,000 (the number of milliseconds in a day).

For seconds, which are input as floating-point numbers, conversion to milliseconds may result in a loss of precision. For example, if you specify 25.532962 seconds, only 25532 milliseconds are stored, and the exact value returned by the `seconds` function (such as 25.532) is architecture-dependent.

Reference Summary

Creating an Interval	d_Interval
Setting an Interval	operator=
Getting Information	day hour is_zero minute second
Comparing Intervals	::operator== ::operator!= ::operator< ::operator<= ::operator> ::operator>=
Arithmetic Operations on Intervals	::operator+ operator+= operator- ::operator- operator-= ::operator* operator*= ::operator/ operator/=

Reference Index

::operator==	(ODMG) Equality operator; returns 1 if the specified intervals are of the same duration.
::operator!=	(ODMG) Inequality operator; returns 1 if the specified intervals are of different durations.
::operator<	(ODMG) Less-than operator; returns 1 if one interval is less than another.
::operator<=	(ODMG) Less-than-or-equal-to operator; returns 1 if one interval is less than or equal to another.
::operator>	(ODMG) Greater-than operator; returns 1 if one interval is greater than another.
::operator>=	(ODMG) Greater-than-or-equal-to operator; returns 1 if one interval is greater than or equal to another.

Constructors

d_Interval

(*ODMG*) Constructs a new interval of time from the specified components.

```
d_Interval(
     int day=0,
     int hour=0,
     int minute=0,
     float second=0.0);
```

Parameters

day

Number of whole days in the interval.

hour

Number of hours in the interval. This number must be less than 24; if the interval is longer than a day, you must represent the excess hours as a number of whole days.

minute

Number of minutes in the interval.

second

Number of seconds in the interval.

Discussion When combined, *hour*, *minute*, and *second* may not exceed 86,400,000 (the number of milliseconds in a day). For more information about permitted component values, see “About d_Interval” on page 111.

d_Interval

(*ODMG*) Copy constructor that constructs a new interval from the specified interval.

```
d_Interval(const d_Interval &interval);
```

Operators

::operator+ global function

(*ODMG*) Addition operator; allocates a new interval that is set to the sum of the specified intervals.

```
d_Interval ::operator+(
     const d_Interval &left,
     const d_Interval &right);
```

operator+=

(*ODMG*) Increment operator; adds the specified interval to this interval.

```
d_Interval &operator+=(const d_Interval &interval);
```

operator-

(*ODMG*) Unary minus operator; returns the negative of this interval.

```
d_Interval operator-() const;
```

::operator- global function

(*ODMG*) Subtraction operator; allocates a new interval that is set to the difference between the specified intervals.

```
d_Interval ::operator-(
     const d_Interval &left,
     const d_Interval &right);
```

operator-=

(*ODMG*) Decrement operator; subtracts the specified interval from this interval.

```
d_Interval &operator-=(const d_Interval &interval);
```

::operator* global function

(*ODMG*) Multiplication operator; multiplies the specified interval and an integer value, and then returns the result in a new interval.

```
1.   d_Interval ::operator*(
       const d_Interval &left,
       int right);
2.   d_Interval ::operator*(
       int left,
       const d_Interval &right);
```

operator*=

(*ODMG*) Multiplication operator; multiplies this interval by the specified integer value.

```
d_Interval &operator*=(int);
```

::operator/ global function

(*ODMG*) Division operator; divides the specified interval by an integer value.

```
d_Interval ::operator/ (
     const d_Interval &left,
     int right);
```

operator/=

(*ODMG*) Division operator; divides this interval by the specified integer value.

```
d_Interval &operator/=(int);
```

operator=

(*ODMG*) Assignment operator; sets this interval to the specified interval.

```
d_Interval &operator=(const d_Interval &interval);
```

::operator== global function

(*ODMG*) Equality operator; returns 1 if the specified intervals are of the same duration.

```
int ::operator==(
     const d_Interval &left,
     const d_Interval &right);
```

::operator!= global function

(*ODMG*) Inequality operator; returns 1 if the specified intervals are of different durations.

```
int ::operator!= (
     const d_Interval &left,
     const d_Interval &right);
```

::operator< global function

(*ODMG*) Less-than operator; returns 1 if one interval is less than another.

```
int ::operator< (
     const d_Interval &left,
     const d_Interval &right);
```

::operator<= global function

(*ODMG*) Less-than-or-equal-to operator; returns 1 if one interval is less than or equal to another.

```
int ::operator<=(
     const d_Interval &left,
     const d_Interval &right);
```

::operator> global function

(*ODMG*) Greater-than operator; returns 1 if one interval is greater than another.

```
int ::operator> (
     const d_Interval &left,
     const d_Interval &right);
```

::operator>= global function

(*ODMG*) Greater-than-or-equal-to operator; returns 1 if one interval is greater than or equal to another.

```
int ::operator>=(
     const d_Interval &left,
     const d_Interval &right);
```

Member Functions

day

(*ODMG*) Returns the normalized day component of this interval.

```
int day() const;
```

hour

(*ODMG*) Returns the normalized hour component of this interval.

```
int hour() const;
```

is_zero

(*ODMG*) Returns 1 if the duration of this interval is zero.

```
int is_zero() const;
```

minute

(*ODMG*) Returns the normalized minute component of this interval.

```
int minute() const;
```

second

(*ODMG*) Returns the normalized seconds component of this interval.

```
float second() const;
```

d_Iterator<*element_type*> Class

Inheritance: **d_Iterator<*element_type*>**

The non-persistence-capable ODMG class d_Iterator<*element_type*> represents a *VArray iterator*, which supports iteration over the elements of a VArray of the same element type.

See:

- "Reference Summary" on page 120 for an overview of member functions
- "Reference Index" on page 120 for a list of member functions

See also:

- d_Varray<element_type>
- ooVArrayT<element_type>

About VArray Iterators

You create an iterator to find elements of a particular VArray by calling the create_iterator member function on that VArray. The new iterator is initialized to point to the first element of that VArray. The initialized iterator pins the VArray in memory.

An initialized VArray iterator supports two alternative iteration styles for finding the VArray's elements:

- The next member function tests for a next element, sets a parameter to the current element, 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.

Reference Summary

Creating a VArray Iterator	`d_Iterator`
Assigning a VArray Iterator	`operator=`

Positioning a VArray Iterator	`advance`
	`next`
	`operator++`
	`operator--`
	`reset`

Finding a VArray Element	`get_element` `next`
Testing for the Next Element	`next` `not_done`

Reference Index

reset	(ODMG) Reinitializes this VArray iterator to the start of iteration for the same VArray.
operator++	(ODMG) Increment operator; advances this VArray iterator forward to the next element in the VArray.
operator--	(ODMG) Decrement operator; moves this VArray iterator backward to the previous element in the VArray.
operator=	(ODMG) Assignment operator; assigns the specified VArray iterator to this VArray iterator.

Constructors and Destructors

d_Iterator

(*ODMG*) Default constructor that constructs a new uninitialized VArray iterator.

```
d_Iterator();
```

Discussion

You initialize the new VArray iterator by assigning another (initialized) VArray iterator to it. (You normally obtain an initialized VArray iterator by calling the create_iterator member function on a VArray of the same *element_type*.)

d_Iterator

(*ODMG*) Copy constructor that constructs a new VArray iterator initialized with the specified VArray iterator.

```
d_Iterator(const d_Iterator<element_type> &iterator);
```

Parameters

iterator

VArray iterator of the same *element_type* as the VArray iterator you are constructing.

Operators

operator++

(*ODMG*) Increment operator; advances this VArray iterator forward to the next element in the VArray.

```
1.  d_Iterator<element_type> &operator++();
2.  d_Iterator<element_type> operator++(int n);
```

Parameters *n*

This parameter is not used in calling this operator; its presence in the function declaration specifies a postfix operator.

Returns (Variant 1) This VArray iterator, advanced to the next element.
(Variant 2) A new VArray iterator set to this iterator, before this iterator is advanced.

Discussion Variant 1 is the prefix increment operator, which advances this VArray iterator and then returns it.

Variant 2 is the postfix increment operator, which allocates and returns a new VArray iterator set to this iterator, and then advances this iterator.

An error is signaled if you attempt to advance the VArray iterator once it has already reached the end of the VArray.

The increment operator is equivalent to the `advance` member function.

operator--

(*ODMG*) Decrement operator; moves this VArray iterator backward to the previous element in the VArray.

```
1.  d_Iterator<element_type> &operator--();
2.  d_Iterator<element_type> operator--(int n);
```

Parameters *n*

This parameter is not used in calling this operator; its presence in the function declaration specifies a postfix operator.

Returns (Variant 1) This VArray iterator, moved to the previous element.
(Variant 2) A new VArray iterator set to this iterator, before this iterator is decremented.

Discussion Variant 1 is the prefix decrement operator, which moves this VArray iterator and then returns it.

Variant 2 is the postfix decrement operator, which allocates and returns a new VArray iterator set to this iterator, and then moves this iterator.

An error is signaled if you attempt to move the VArray iterator backward once it has already reached the first element of the VArray.

operator=

(*ODMG*) Assignment operator; assigns the specified VArray iterator to this VArray iterator.

```
d_Iterator<element_type> &operator=(
     const d_Iterator<element_type> &iterator);
```

Parameters *iterator*

VArray iterator of the same *element_type* as this VArray iterator.

Member Functions

advance

(*ODMG*) Advances this VArray iterator to the next element in the VArray.

```
void advance();
```

get_element

(*ODMG*) Returns the element at this VArray iterator's current position in the VArray.

```
element_type get_element() const;
```

Returns The current element in the VArray.

Discussion An error is signaled if there is no current element—for example, if the iteration is completed or if the VArray has no elements.

next

(*ODMG*) Tests whether to continue iterating and, if so, assigns the current element to the output parameter and advances this VArray iterator to the next element.

```
int next(element_type &found);
```

Parameters

found

Output parameter to which to assign the current element in the VArray.

Returns

1, if there is a next element in the VArray; 0, if iteration is complete.

Discussion

This member function is implemented using the `not_done`, `get_element`, and `advance` member functions.

not_done

(*ODMG*) Tests whether to continue iterating.

```
int not_done() const;
```

Returns

1, if there is a next element in the VArray; 0, if iteration is complete.

reset

(*ODMG*) Reinitializes this VArray iterator to the start of iteration for the same VArray.

```
void reset();
```

This member function positions the iterator just before the first element of the VArray. Consequently, after a reset, the `advance` member function positions the iterator at the first element. However, you do *not* need to advance the iterator before calling either `get_element` or `next`, because both of these member functions treat a reset iterator as if it started at the first element.

d_Ref_Any Class

Inheritance: **d_Ref_Any**

The non-persistence-capable ODMG class d_Ref_Any represents an *ODMG generic reference* to a persistent object of any type.

See:

- "Reference Summary" on page 125 for an overview of member functions
- "Reference Index" on page 126 for a list of member functions

About ODMG Generic References

An ODMG generic reference allows conversions among object references in the type hierarchy. An ODMG generic reference can be used as an intermediary between any two types d_Ref<X> and d_Ref<Y>, where X and Y are different types. An Objectivity/C++ object reference ooRef(*X*) can always be converted to or constructed from a d_Ref_Any; each object-reference class has an appropriate constructor and a conversion operator.

Reference Summary

Creating a Generic Reference	d_Ref_Any
Setting a Generic Reference	clear operator=
Getting Information	is_null

Comparing Generic References	::operator== ::operator!=
Deleting the Referenced Object	delete_object

Reference Index

Constructors and Destructors

d_Ref_Any

(*ODMG*) Default constructor that constructs a new null ODMG generic reference.

```
d_Ref_Any();
```

d_Ref_Any

(*ODMG*) Constructs a new ODMG generic reference that refers to the same persistent object as the specified ODMG generic reference or pointer.

```
1.  d_Ref_Any(const d_Ref_Any &);
2.  d_Ref_Any(d_Persistent_Object *);
```

Operators

operator=

(*ODMG*) Assignment operator; sets this ODMG generic reference to refer to the same persistent object as the specified ODMG generic reference or pointer.

```
1.  d_Ref_Any &operator=(const d_Ref_Any &);
2.  d_Ref_Any &operator=(d_Persistent_Object *);
```

::operator== global function

(*ODMG*) Equality operator; tests whether the same persistent object is referenced by both of the specified items.

```
1.  int ::operator==(
      const d_Ref_Any &left,
      const d_Ref_Any &right);
2.  int ::operator==(
      const d_Ref_Any &left,
      const d_Persistent_Object *right);
3.  int ::operator==(
      const d_Persistent_Object *left,
      const d_Ref_Any &right);
4.  int ::operator==(
      const ooRefHandle(ooObj) &left,
      const d_Ref_Any &right);
5.  int ::operator==(
      const d_Ref_Any &left,
      const ooRefHandle(ooObj) &right);
```

Returns 1, if *left* and *right* refer to the same persistent object; otherwise 0.

Discussion

You can compare two ODMG generic references (variant 1), an ODMG generic reference and a pointer to a persistent object (variants 2 and 3), or an ODMG generic reference and an object reference or handle to a persistent object (variants 4 and 5).

::operator!= global function

(*ODMG*) Inequality operator; tests whether different persistent objects are referenced by the two specified items.

```
1.  int ::operator!=(
      const d_Ref_Any &left,
      const d_Ref_Any &right);
2.  int ::operator!=(
      const d_Ref_Any &left,
      const d_Persistent_Object *right);
3.  int ::operator!=(
      const d_Persistent_Object *left,
      const d_Ref_Any &right);
4.  int ::operator!=(
      const ooRefHandle(ooObj) &left,
      const d_Ref_Any &right);
5.  int ::operator!=(
      const d_Ref_Any &left,
      const ooRefHandle(ooObj) &right);
```

Returns

1, if *left* and *right* do not refer to the same persistent object; otherwise 0.

Discussion

You can compare two ODMG generic references (variant 1), an ODMG generic reference and a pointer to a persistent object (variants 2 and 3), or an ODMG generic reference and an object reference or handle to a persistent object (variants 4 and 5).

Member Functions

clear

(*ODMG*) Sets this ODMG generic reference to null.

```
void clear();
```

delete_object

(*ODMG*) Deletes the persistent object referenced by this ODMG generic reference.

```
void delete_object();
```

is_null

(*ODMG*) Tests whether this ODMG generic reference is null.

```
int is_null() const;
```

Returns 1, if this generic reference is null; 0, if it is not null.

d_Time Class

Inheritance: **`d_Time`**

The non-persistence-capable ODMG class `d_Time` represents a specific *time value*, which is internally stored in Greenwich Mean Time (GMT).

See:

- "Reference Summary" on page 135 for an overview of member functions
- "Reference Index" on page 136 for a list of member functions

To use this class, your application must include the `ooTime.h` header file. No extra linking is required.

About Time Values

A time value represents an instant in time expressed as a number of hours, minutes, and seconds since midnight. All arithmetic on time values is done on a modulo 24-hour basis.

An application can create a time value from the current time or from a specified number of hours, minutes, and seconds. In either case, the number of hours is expressed locally to the time zone specified by the application. The hours, minutes, and seconds components are then normalized so that they are expressed as the number of milliseconds after midnight in Greenwich Mean Time (GMT), and the time value stores the difference in hours between the specified time zone and GMT.

An application can specify a time zone explicitly for each time value it creates. More typically, however, an application sets a default time zone and then creates its time values without explicitly specifying their time zones. Initially, the default time zone is set to GMT; the application uses a static member function to change its default time zone (for example, to the computer's local time zone). Because

the default time zone is static data, it is compiled into the application, but not stored in the database.

An application can specify a time zone as a constant of type `Time_Zone`. Each time zone is numbered according to the number of hours that must be added or subtracted from local time to get GMT. For example, `GMT6` indicates a time of 6 hours ahead of GMT, so 6 must be subtracted from it to get GMT. Conversely, `GMT_6` means that the time is 6 hours earlier than GMT (read the underscore as a minus), so 6 hours must be added to get GMT.

It is the application's responsibility to adjust for daylight savings time, if necessary; the `d_Time` class does not provide any mechanisms for doing so.

Examples

The following program is a prototype for simulating a simple roaming application and a simple billing application for cellular telephone calls. The first two transactions of the program each simulate the roaming application, which records the time and duration of a cellular telephone call. The third transaction simulates the centralized billing application that calculates the cost of each call.

```
// DDL file
class eventObject : public ooObj {
public:
   int32 event_num;               // unique event number
   d_Time event_time;              // beginning of a call
   int32 duration_in_seconds;     // duration of a call
};

// Application code
int main() {
   ooHandle(ooFBObj) fdH;
   ooHandle(ooDBObj) dbH;
   ooHandle(ooContObj) contH;
   ooHandle(eventObject) eventH;
   ooTrans trans;

   ooInit();

// The first transaction simulates the roaming application at
// a local receiving station, which records a call made by
// customer1

   trans.start();
   fdH.open("Example1", oocUpdate);
```

```
    if(!dbH.exist(fdH,"ROAMING_CALLS") ) {
       dbH = new(fdH) ooDBObj("ROAMING_CALLS");
    }
    dbH.open(fdH,"ROAMING_CALLS",oocUpdate);

    if(!contH.exist(dbH,"Customer1")) {
       contH = new ("Customer1",0,0,0,dbH) cont ("Customer1");
    }
    contH.open(dbH,"Customer1",oocUpdate);

    // Assume customer1 makes a call from zone GMT_5 (-5 hours),
    // starting at 11.30 p.m. and lasting 50 minutes.

    // Set the application's default time zone to GMT_5
    d_Time::set_default_Time_Zone(d_Time::Time_Zone::GMT_5);

    // Create a time value at the time of call activation
    // The new time value is local to GMT_5
    d_Time  call_N_1;

    // Store the time value in an event that represents the call
    eventH = new(contH) eventObject();
    eventH->event_num = 1;
    eventH->event_time = call_N_1;
    eventH->duration_in_seconds = 300;

    trans.commit();

// The second transaction simulates the roaming application at
// a second local receiving station, which records the second
// call made by customer

    trans.start();
    // open the federated database
    // open the ROAMING_CALLS database
    // open Customer1 container
    …

    // Assume customer1 travels for a day and makes a call
    // from zone GMT10 (+10 hours), starting at 11.50 a.m. and
    // lasting 30 minutes.

    // Set the application's default time zone to GMT10
    d_Time::set_default_Time_Zone(d_Time::Time_Zone::GMT10);
```

```
    // Create a time value at the time of call activation
    // The new time value is local to GMT10
    d_Time  call_N_2;

    // Store the time value in an event that represents the call
    eventH = new(contH) eventObject();
    eventH->event_num = 2;
    eventH->event_time = call_N_2;
    eventH->duration_in_seconds = 180;

    trans.commit();

// The third transaction simulates a central billing application,
// which calculates a statement for customer1

    trans.start();
    // open the federated database
    // open the ROAMING_CALLS database
    // open Customer1 container
    …

    ooItr(eventObject)  eventI;
    unsigned short local_hour_of_call, GMT_hour_of_call;
    short local_to_GMT_difference;

    // Find each call event for customer1 and calculate its cost
    eventI.scan(contH);
    while(eventI.next()) {
       // Get the call's time value expressed in GMT
       GMT_hour_of_call = eventI->event_time.hour();

       // Get the number of hours between the call's time zone
       // and GMT
       local_to_GMT_difference =
             eventI->event_time.tz_hour();

       // Reconstruct the time of the call in its time zone
       local_hour_of_call =
             GMT_hour_of_call + local_to_GMT_difference;

       // Use local_hour_of_call, the call's duration, and the
       // call's event_time->minute() value to determine whether
       // to use a peak-time rate, an off-peak rate, or a
       // combination of rates. Get the cost of the call by
       // calculating the number of minutes at each rate.
       …
    } // End while
```

```
   // Print a statement showing each call and its cost.
   …

   trans.commit();

   return 1;
} // End main
```

Reference Summary

Creating a Time Value	d_Time
Time Zones	Time_Zone
Setting a Time Value	operator=
Setting a Time Zone	set_default_Time_Zone set_default_Time_Zone_to_local
Getting Information	current hour minute second tz_hour tz_minute
Comparing Time Values	::operator== ::operator!= ::operator< ::operator<= ::operator> ::operator>=
Arithmetic Operations on Time Values	::operator+ operator+= ::operator- operator-=
Determining Whether Time Periods Overlap	::overlaps

Reference Index

::operator<=	(ODMG) Less-than-or-equal-to operator; returns 1 if one time value is less than or equal to another.
::operator>	(ODMG) Greater-than operator; returns 1 if one time value is greater than another.
::operator>=	(ODMG) Greater-than-or-equal-to operator; returns 1 if one time value is greater than or equal to another.
::overlaps	(ODMG) Returns 1 if two specified time periods overlap, where one or both periods are specified using time values.
second	(ODMG) Returns the seconds component of this time value.
set_default_Time_Zone	(ODMG) Sets the default time zone.
set_default_Time_Zone_to_local	(ODMG) Resets the default time zone to your computer's local time zone.
Time_Zone	(ODMG) Enumerated type used to denote a specific time zone.
tz_hour	(ODMG) Returns the number of hours between the time zone stored in this time value and GMT.
tz_minute	(ODMG) Returns the number of minutes between the time-zone minutes stored in this time value and GMT.

Types

Time_Zone

(*ODMG*) Enumerated type used to denote a specific time zone.

```
enum Time_Zone {
     GMT = 0,
     GMT12 = 12,
     GMT_12 = –12,
     GMT1 = 1,
     GMT_1 = –1,
     GMT2 = 2,
     GMT_2 = –2,
```

```
    GMT3 = 3,
    GMT_3 =  −3,
    GMT4 = 4,
    GMT_4 = −4,
    GMT5 = 5,
    GMT_5 = −5,
    GMT6 = 6,
    GMT_6 = −6,
    GMT7 = 7,
    GMT_7 = −7,
    GMT8 = 8,
    GMT_8 = −8,
    GMT9 = 9,
    GMT_9 = −9,
    GMT10 = 10,
    GMT_10 = −10,
    GMT11 = 11,
    GMT_11 = −11,
    USeastern = −5,
    UScentral = −6,
    USmountain = −7,
    USpacific = −8};
```

Discussion Time zones are numbered according to the number of hours that must be added or subtracted from the local time to get the time in Greenwich, England (GMT). For example, `GMT6` indicates a time of 6 hours greater than GMT, so 6 must be subtracted from it to get GMT. Conversely, `GMT_6` means that the time is 6 hours earlier than GMT (read the underscore as a minus).

Constructors

d_Time

(*ODMG*) Default constructor that constructs a new time value set to the current time in the default time zone.

```
d_Time();
```

d_Time

(*ODMG*) Constructs a new time value from the specified time value.

```
1.  d_Time(const d_Time &);
2.  d_Time(const d_Timestamp &);
```

d_Time

(*ODMG*) Constructs a new time value from the specified components.

```
1.   d_Time(
       unsigned short hour,
       unsigned short minute,
       float second);
2.   d_Time(
       unsigned short hour,
       unsigned short minute,
       float second,
       short tzhour,
       short tzminute);
```

Parameters

hour

Hours component of the new time value; the number of hours past midnight.

minute

Minutes component of the new time value; the number of minutes in any fractional hour remaining after whole hours are subtracted out.

second

Seconds component of the new time value; the number of seconds in any fractional minute remaining after whole minutes are subtracted out.

tzhour

The time zone for the new time value. You can specify the time zone using a constant of type Time_Zone. Alternatively, you can specify the integer number of hours to subtract from the hours component of the new time value to get the hours component of GMT:

- Specify a positive number if the new time value is later than GMT.
- Specify a negative number if the new time value is earlier than GMT.

tzminute

Number of minutes to subtract from the minutes component of the new time value to get the minutes component of GMT; normally 0.

Discussion

Variant 1 constructs a new time value that is local to the default time zone, which implicitly sets the *tzhour* and *tzminute* components.

Variant 2 allows you to construct a time value that is local to a nondefault time zone.

Operators

::operator+ global function

(*ODMG*) Addition operator; adds the specified time value and interval and returns the sum in a newly allocated time value.

```
1.   d_Time ::operator+(
       const d_Time &left,
       const d_Interval &right);
2.   d_Time ::operator+(
       const d_Interval &left,
       const d_Time &right);
```

operator+=

(*ODMG*) Increment operator; adds the specified interval to this time value.

```
d_Time &d_Time::operator+=(const d_Interval &interval);
```

::operator- global function

(*ODMG*) Subtraction operator; subtracts a time value or interval from a time value and returns the difference in a newly allocated time value.

```
1.   d_Interval ::operator-(
       const d_Time &left,
       const d_Time &right);
2.   d_Time ::operator -(
       const d_Time &left,
       const d_Interval &right);
```

operator-=

(*ODMG*) Decrement operator; subtracts the specified interval from this time value.

```
d_Time &operator-=(const d_Interval &interval);
```

operator=

(*ODMG*) Assignment operator; sets this time value to the specified time value.

```
1.   d_Time &operator=(const d_Time &time);
2.   d_Time operator=(const d_Timestamp &timeStamp);
```

::operator== global function

(*ODMG*) Equality operator; returns 1 if every component of one time value matches the corresponding component of the other.

```
int ::operator==(
     const d_Time &left,
     const d_Time &right);
```

::operator!= global function

(*ODMG*) Inequality operator; returns 1 if any component of one time value differs from the corresponding component of the other.

```
int ::operator!= (
     const d_Time &left,
     const d_Time &right);
```

::operator< global function

(*ODMG*) Less-than operator; returns 1 if one time value is less than another.

```
int ::operator< (
     const d_Time &left,
     const d_Time &right);
```

::operator<= global function

(*ODMG*) Less-than-or-equal-to operator; returns 1 if one time value is less than or equal to another.

```
int ::operator<=(
     const d_Time &left,
     const d_Time &right);
```

::operator> global function

(*ODMG*) Greater-than operator; returns 1 if one time value is greater than another.

```
int ::operator> (
     const d_Time &left,
     const d_Time &right);
```

::operator>= global function

(*ODMG*) Greater-than-or-equal-to operator; returns 1 if one time value is greater than or equal to another.

```
int ::operator>=(
     const d_Time &left,
     const d_Time &right);
```

Member Functions

current

(*ODMG*) Returns the current time in the default time zone.

```
static d_Time current();
```

hour

(*ODMG*) Returns the hours component of this time value, expressed in GMT.

```
unsigned short hour() const;
```

Discussion This member function expresses the hours component of this time value as a number of hours past midnight in GMT. To get the hours component expressed locally to the time value's time zone, you must add the value returned by `tz_hour`.

minute

(*ODMG*) Returns the minutes component of this time value.

```
unsigned short minute() const;
```

second

(*ODMG*) Returns the seconds component of this time value.

```
float second( ) const;
```

set_default_Time_Zone

(*ODMG*) Sets the default time zone.

```
static void set_default_Time_Zone(Time_Zone zone);
```

Discussion The default time zone value is initially set to GMT. (This behavior differs from the ODMG standard, in which the default time zone is initially set to your computer's local time.) Changing the default time zone affects subsequently created time values, but does not affect any existing time values.

set_default_Time_Zone_to_local

(*ODMG*) Resets the default time zone to your computer's local time zone.

```
static void set_default_Time_Zone_to_local( );
```

Discussion Changing the default time zone affects subsequently created time values, but does not affect any existing time values.

tz_hour

(*ODMG*) Returns the number of hours between the time zone stored in this time value and GMT.

```
short tz_hour( ) const;
```

Returns The number of hours add to the value returned by `hour` to get the hours component in this time value's time zone. A positive number indicates this time value is later than GMT; a negative number indicates this time value is earlier than GMT.

tz_minute

(*ODMG*) Returns the number of minutes between the time-zone minutes stored in this time value and GMT.

```
short tz_minute( ) const;
```

Returns The number of minutes to add to the value returned by `minute` to get the minutes component in this time value's time zone; normally 0.

Related Global Functions

::overlaps

(*ODMG*) Returns 1 if two specified time periods overlap, where one or both periods are specified using time values.

```
1.   int ::overlaps(
       const d_Time &startLeft,
       const d_Time &endLeft,
       const d_Time &startRight,
       const d_Time &endRight);

2.   int ::overlaps(
       const d_Timestamp &startLeft,
       const d_Timestamp &endLeft,
       const d_Time &startRight,
       const d_Time &endRight);

3.   int ::overlaps(
       const d_Time &startLeft,
       const d_Time &endLeft,
       const d_Timestamp &startRight,
       const d_Timestamp &endRight);
```

Discussion

Each time period is specified by a start and end time. Variant 1 allows you to specify the time periods using time values. Variants 2 and 3 allow you to specify time periods using time values and timestamps.

d_Timestamp Class

Inheritance: **d_Timestamp**

The non-persistence-capable ODMG class d_Timestamp represents a *timestamp*, which contains both a date (see d_Date) and a time value (see d_Time).

See:

- "Reference Summary" on page 145 for an overview of member functions
- "Reference Index" on page 146 for a list of member functions

To use this class, your application must include the ooTime.h header file. No extra linking is required.

Reference Summary

Creating a Timestamp	d_Timestamp
Setting a Timestamp	operator=
Getting Information	current date day hour minute month second time tz_hour tz_minute year

Comparing Timestamps	::operator== ::operator!= ::operator< ::operator<= ::operator> ::operator>=
Arithmetic Operations on Timestamps	::operator+ operator+= ::operator- operator-=
Determining Whether Time Periods Overlap	::overlaps

Reference Index

operator-=	(ODMG) Decrement operator; subtracts the specified interval from this timestamp.
operator=	(ODMG) Assignment operator; assigns the specified timestamp or date to this timestamp.
::operator==	(ODMG) Equality operator; returns 1 if every component of one timestamp matches the corresponding component of the other.
::operator!=	(ODMG) Inequality operator; returns 1 if any component of one timestamp differs from the corresponding component of the other.
::operator<	(ODMG) Less-than operator; returns 1 if one timestamp is less than another.
::operator<=	(ODMG) Less-than-or-equal-to operator; returns 1 if one timestamp is less than or equal to another.
::operator>	(ODMG) Greater-than operator; returns 1 if one timestamp is greater than another.
::operator>=	(ODMG) Greater-than-or-equal-to operator; returns 1 if one timestamp is greater than or equal to another.
::overlaps	(ODMG) Returns 1 if two time periods overlap, where one or both periods are specified using timestamps.
second	(ODMG) Returns the seconds component of this timestamp.
time	(ODMG) Returns the time component of this timestamp.
tz_hour	(ODMG) Returns the number of hours between the time zone stored in this timestamp and GMT.
tz_minute	(ODMG) Returns the number of minutes between the time-zone minutes stored in this timestamp and GMT.
year	(ODMG) Returns the year component of this timestamp.

Constructors

d_Timestamp

(*ODMG*) Default constructor that constructs a new timestamp set to the current date and time.

```
d_Timestamp();
```

d_Timestamp

(*ODMG*) Constructs a new timestamp from the specified object(s).

```
1.  d_Timestamp(const d_Date &);
2.  d_Timestamp(const d_Date &, const d_Time &);
3.  d_Timestamp(const d_Timestamp &);
```

Discussion

You can construct a timestamp from:

- A date plus the current time (variant 1)
- A date and a time value (variant 2)
- Another timestamp (variant 3)

d_Timestamp

(*ODMG*) Constructs a new timestamp with the specified components.

```
d_Timestamp(
     unsigned short year,
     unsigned short month= 1,
     unsigned short day = 1,
     unsigned short hour = 0,
     unsigned short minute = 0,
     float second = 0.0);
```

Parameters

year

Corresponds to the year component of a `d_Date`. Maximum value is 65535.

month

Corresponds to the month component of a `d_Date`.

day

Corresponds to the day component of a `d_Date`.

hour

Corresponds to the hours component of a `d_Time`.

minute

Corresponds to the minutes component of a `d_Time`.

second

Corresponds to the seconds component of a `d_Time`.

Discussion

The time value in the new timestamp is local to the application's default time zone, which is set by `d_Time::set_default_Time_Zone`. The timestamp stores

the time zone as the number of hours between the local time and Greenwich Mean Time (GMT).

Operators

::operator+

global function

(*ODMG*) Addition operator; allocates a new timestamp that is set to the sum of the specified timestamp and interval.

```
1.   d_Timestamp ::operator+(
       const d_Timestamp &left,
       const d_Interval &right);
2.   d_Timestamp ::operator+(
       const d_Interval &left,
       const d_Timestamp &right);
```

WARNING Addition that causes the year component to exceed 65535 results in an invalid timestamp.

operator+=

(*ODMG*) Increment operator; adds the specified interval to this timestamp.

```
d_Timestamp &operator+=(const d_Interval &);
```

WARNING Addition that causes the year component to exceed 65535 results in an invalid timestamp.

::operator-

global function

(*ODMG*) Subtraction operator; allocates a new timestamp that is set to the specified timestamp minus the specified interval.

```
d_Timestamp ::operator-(
     const d_Timestamp &left,
     const d_Interval &right);
```

operator-=

(*ODMG*) Decrement operator; subtracts the specified interval from this timestamp.

```
d_Timestamp &operator-=(const d_Interval &);
```

operator=

(*ODMG*) Assignment operator; assigns the specified timestamp or date to this timestamp.

```
1.   d_Timestamp &operator=(const d_Timestamp &);
2.   d_Timestamp &operator=(const d_Date &);
```

::operator== global function

(*ODMG*) Equality operator; returns 1 if every component of one timestamp matches the corresponding component of the other.

```
int ::operator==(
     const d_Timestamp &left,
     const d_Timestamp &right);
```

::operator!= global function

(*ODMG*) Inequality operator; returns 1 if any component of one timestamp differs from the corresponding component of the other.

```
int ::operator!= (
     const d_Timestamp &left,
     const d_Timestamp &right);
```

::operator< global function

(*ODMG*) Less-than operator; returns 1 if one timestamp is less than another.

```
int ::operator< (
     const d_Timestamp &left,
     const d_Timestamp &right);
```

::operator<= global function

(*ODMG*) Less-than-or-equal-to operator; returns 1 if one timestamp is less than or equal to another.

```
int ::operator<=(
     const d_Timestamp &left,
     const d_Timestamp &right);
```

::operator> global function

(*ODMG*) Greater-than operator; returns 1 if one timestamp is greater than another.

```
int ::operator> (
     const d_Timestamp &left,
     const d_Timestamp &right);
```

::operator>= global function

(*ODMG*) Greater-than-or-equal-to operator; returns 1 if one timestamp is greater than or equal to another.

```
int ::operator>=(
     const d_Timestamp &left,
     const d_Timestamp &right);
```

Member Functions

current

(*ODMG*) Returns a timestamp containing the current date and time in the default time zone.

```
static d_Timestamp current();
```

date

(*ODMG*) Returns the date component of this timestamp.

```
const d_Date  &date() const;
```

day

(*ODMG*) Returns the day component of this timestamp.

```
unsigned short day() const;
```

hour

(*ODMG*) Returns the hours component of this timestamp, expressed in GMT.

```
unsigned short hour() const;
```

Discussion

This member function expresses the hours component of this timestamp as a number of hours past midnight in GMT. To get the hours component expressed locally to the time value's time zone, you must add the value returned by `tz_hour`.

minute

(*ODMG*) Returns the minutes component of this timestamp.

```
unsigned short minute() const;
```

month

(*ODMG*) Returns the month component of this timestamp.

```
unsigned short month() const;
```

second

(*ODMG*) Returns the seconds component of this timestamp.

```
float second() const;
```

time

(*ODMG*) Returns the time component of this timestamp.

```
const d_Time  &time() const;
```

tz_hour

(*ODMG*) Returns the number of hours between the time zone stored in this timestamp and GMT.

```
short tz_hour() const;
```

Returns The number of hours to add to the value returned by hour to get the hours component in this timestamp's time zone. A positive number indicates this time value is later than GMT; a negative number indicates this time value is earlier than GMT.

tz_minute

(*ODMG*) Returns the number of minutes between the time-zone minutes stored in this timestamp and GMT.

```
short tz_minute() const;
```

Returns The number of minutes to add to the value returned by minute to get the minutes component in this timestamp's time zone; normally 0.

year

(*ODMG*) Returns the year component of this timestamp.

```
unsigned short year() const;
```

Related Global Functions

::overlaps

global function

(*ODMG*) Returns 1 if two time periods overlap, where one or both periods are specified using timestamps.

```
1.   int ::overlaps(
       const d_Timestamp &startLeft,
       const d_Timestamp &endLeft,
       const d_Timestamp &startRight,
       const d_Timestamp &endRight);
2.   int ::overlaps(
       const d_Timestamp &startLeft,
       const d_Timestamp &endLeft,
       const d_Date &startRight,
       const d_Date &endRight);
3.   int ::overlaps(
       const d_Date &startLeft,
       const d_Date &endLeft,
       const d_Timestamp &startRight,
       const d_Timestamp &endRight);
```

```
4.   int ::overlaps(
       const d_Timestamp &startLeft,
       const d_Timestamp &endLeft,
       const d_Time &startRight,
       const d_Time &endRight);

5.   int ::overlaps(
       const d_Time &startLeft,
       const d_Time &endLeft,
       const d_Timestamp &startRight,
       const d_Timestamp &endRight);
```

Discussion

Each time period is specified by a start and end time. You can specify the time periods using timestamps (variant 1), using timestamps and dates (variants 2 and 3), or using timestamps and time values (variants 4 and 5).

ooAdmin Class

Inheritance: **ooObj->ooAdmin**

Handle Class: `ooHandle(ooAdmin)`

Object-Reference Class: `ooRef(ooAdmin)`

The persistence-capable class `ooAdmin` is the abstract base class for all *administrator* classes.

To use this class, your application must include the `ooCollections.h` header file. For UNIX linking information, see *Installation and Platform Notes for UNIX*.

About Administrators

An administrator is an object of a concrete class derived from `ooAdmin`. Each collection derived from `ooCollection` has an administrator that manages the containers for the collection's internal objects. A collection's administrator is created when the collection itself is created.

Because the `ooAdmin` class is abstract, you never create instances of it; instead, you work with instances of its concrete derived classes:

- Ordered collections use tree administrators of the `ooTreeAdmin` class.
- Unordered collections use hash administrators of the `ooHashAdmin` class.

You should not create your own subclasses of this class.

Like other persistent objects, administrators are normally manipulated through handles or object references. You call a collection's `admin` member function to obtain an object reference to the collection's administrator.

ooAPObj Class

Inheritance: **ooObj->ooAPObj**

Handle Class: `ooHandle(ooAPObj)`

Object-Reference Class: `ooRef(ooAPObj)`

The persistence-capable class `ooAPObj` represents an *autonomous partition*. You use `ooAPObj` only for autonomous-partition creation; you work with an existing autonomous partition through a handle or object reference.

You can create and work with autonomous partitions only if you have bought and installed Objectivity / DB Fault Tolerant Option (Objectivity / FTO).

See:

- "Reference Index" on page 159 for a list of `ooAPObj` member functions

For operations performed through an autonomous-partition handle or object reference, see:

- "Reference Summary" on page 493

About 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 and containers. This capability is available through the Objectivity / FTO product.

Every autonomous partition has all the system resources necessary to run an Objectivity / DB application, including a boot file, a lock server, and a system-database file. The system-database file contains schema information and

a global catalog of all autonomous partitions, their locations, and the databases they contain.

Each autonomous partition has a system name, which is its logical name within the federated database. The system name of each autonomous partition must be unique among all the system names of databases and autonomous partitions in the federated database.

An autonomous partition also has a unique integer identifier that is assigned by Objectivity / DB when the partition is created.

See Chapter 27, "Autonomous Partitions," in the the Objectivity / C++ programmer's guide for additional information about autonomous partitions.

Working With Autonomous Partitions

Autonomous partitions can be created and deleted with administration tools or from within an application. An application:

- Creates an autonomous partition using the `ooAPObj` class constructor and `operator new`. The new autonomous partition is represented in memory as an instance of `ooAPObj`, which serves as a proxy for the actual system-database file on disk.
- Deletes an autonomous using the global function `ooDelete`; `operator delete` is not available on class `ooAPObj`.

To work with a new autonomous partition, an application must assign the result of `operator new` to an autonomous-partition handle or object reference (an instance of `ooHandle(ooAPObj)` or `ooRef(ooAPObj)`). Similarly, to work with an existing autonomous partition, the application must open it through an autonomous-partition handle or object reference; multiple handles and object references can be set to reference the same autonomous partition. The application then operates on the referenced autonomous partition by:

- Calling a member function on an appropriate handle or object reference.
- Passing an appropriate handle or object reference to a global function.

Member functions for operating on existing autonomous partitions are defined in the *`ooRefHandle`*`(ooAPObj)` classes, not in the `ooAPObj` class itself. Although `ooAPObj` inherits member functions from `ooObj`, these generally do not apply to autonomous partitions; in fact, the inherited member functions cannot be called, because there is no indirect member-access operator (`->`) on the *`ooRefHandle`*`(ooAPObj)` classes.

You may not derive classes from `ooAPObj`.

Reference Index

Constructors

ooAPObj

Constructs a new autonomous partition with the specified system name, lock server host, and file locations.

```
ooAPObj(
     const char *apSysName,
     const char *lockServer,
     const char *apFileHost,
     const char *apFilePath,
     const char *bootFileHost = 0,
     const char *bootFilePath = 0,
     const char *jnlDirHost = 0,
     const char *jnlDirPath = 0);
```

Parameters

apSysName

System name of the autonomous partition to create. The specified name:

- Must follow the same naming rules as files of your operating system.
- Must be unique among all the system names of databases and autonomous partitions in the federated database.

The system name implicitly specifies the name of the autonomous-partition boot file.

lockServer

Name of the host that runs the lock server for the new autonomous partition.

apFileHost

Name of the data server host on which to create the autonomous partition's system-database file.

apFilePath

Fully qualified pathname (including the filename) of the autonomous partition's system-database file on *apFileHost*. By convention, you construct the filename from the system name with the extension .AP.

bootFileHost

Name of the data server host on which to create the autonomous partition's boot file. If you specify 0 (the default) and *bootFilePath* has a nondefault value, the boot file is created on the current host. If you specify 0 for both *bootFileHost* and *bootFilePath*, the boot file is created on *apFileHost*.

bootFilePath

Fully qualified pathname for the directory on *bootFileHost* in which to create the autonomous partition's boot file. The filename for the boot file is *apSysName*.boot. If you specify 0 (the default), the boot file is created in the same directory as the autonomous partition's system-database file on *apFileHost*.

jnlDirHost

Name of the data server host on which to create the journal directory for the autonomous partition. If you specify 0 (the default) and *jnlDirPath* has a nondefault value, the journal directory is created on the current host. If you specify 0 for both *jnlDirHost* and *jnlDirPath*, the journal directory is created on *apFileHost*.

jnlDirPath

Fully qualified pathname for the autonomous partition's journal directory on *jnlDirHost*. If you specify 0 (the default), the journal directory is created in the directory that contains the autonomous partition's system-database file on *apFileHost*.

Operators

operator new

Creates a new autonomous partition in the currently open federated database.

```
void *operator new(
     size_t,
     const ooRefHandle(ooFDObj) &containingFd);
```

Parameters *size_t*

Do not specify; this parameter is automatically initialized by the compiler with the size of the class type in bytes.

containingFd

Federated database in which to create the new autonomous partition. Because an autonomous partition can be created only in the currently open federated database, you can omit this parameter, or, for explicitness, you can specify an object reference or handle to the open federated database.

Returns

Memory pointer to the new autonomous partition. This pointer is null if an error occurs during the creation of the autonomous partition or if an autonomous partition with the specified system name already exists.

Discussion

This operator must be used in an update transaction. When you commit or checkpoint the transaction, the new autonomous partition is made permanent on disk. If the transaction is aborted, the autonomous partition is not created.

Expressions containing this operator are of the following form:

`new(`*`containingFD`*`) ooAPObj(`*`initializers`*`)`

The *`initializers`* you specify are a list of values to be passed as parameters to the `ooAPObj` constructor. At a minimum, you must specify the system name, lock server host, and file location for the new autonomous partition.

You normally assign the result of `operator new` directly to an autonomous partition handle. You can verify the creation of the autonomous partition by checking whether the handle is null.

ooBTree Class

Inheritance: **ooObj->ooCollection->ooBTree**

Handle Class: `ooHandle(ooBTree)`

Object-Reference Class: `ooRef(ooBTree)`

The persistence-capable class `ooBTree` is the abstract base class for classes that represent *ordered collections* containing persistent objects.

See:

- "Reference Summary" on page 164 for an overview of member functions
- "Reference Index" on page 164 for a list of member functions

To use this class, your application must include the `ooCollections.h` header file. For UNIX linking information, see *Installation and Platform Notes for UNIX*.

About Ordered Collections

Ordered collections are *scalable*, that is, they can increase in size with minimal performance degradation. They are implemented using a B-tree data structure so that elements can be added, deleted, and retrieved efficiently. Each node in the B-tree has a corresponding array. The array for a non-leaf node contains references to the nodes at the leaf level of the B-tree; the array for a leaf node contains references to elements of the collection whose indexes are within a particular range. A newly created B-tree consists of a single node (the root of the B-tree) and its array. As the collection grows, additional nodes are created as necessary. When each node is created, its corresponding array is also created.

Concrete classes derived from `ooBTree` represent more specific kinds of ordered collections:

- `ooTreeSet` represents sorted sets of persistent objects.

- ooTreeList represents lists of persistent objects.
- ooTreeMap represents sorted object maps.

Because the ooBTree class is abstract, you never create instances of it; instead, you work with instances of its concrete derived classes. You should not create your own subclasses of this class.

Like other persistent objects, ordered collections are normally manipulated through handles and object references.

For additional information about the the various persistence-capable collection classes, see Chapter 11, "Persistent Collections," in the Objectivity/C++ programmer's guide.

Reference Summary

Adding and Removing Elements	remove
Getting Elements	first get iterator last
Getting Indexes	indexOf lastIndexOf
Getting Information	depth size
Testing	contains isEmpty
Maintaining the B-Tree	compact
Viewing in an MROW Transaction	refresh

Reference Index

depth	Gets the depth of this ordered collection's B-tree.
first	Finds the first element (or key) in this ordered collection.
get	Finds the specified element of this ordered collection.
indexOf	Searches this ordered collection for the first element (or key) that is equal to the specified object.
isEmpty	Tests whether this ordered collection is empty.
iterator	Initializes a scalable-collection iterator to find the elements of this ordered collection.
last	Finds the last element (or key) in this ordered collection.
lastIndexOf	Searches this ordered collection for the last element (or key) that is equal to the specified object.
refresh	Refreshes each container used internally by this ordered collection, except for the container in which the collection itself is stored.
remove	Removes the first occurrence of the specified object from this ordered collection.
size	Gets the size of this ordered collection.

Member Functions

compact

Minimizes the number of nodes in this ordered collection's B-tree.

```
virtual void compact();
```

Discussion

After you have added all elements that you expect this collection to have, you can call this member function to minimize the number of nodes in its B-tree. Doing so saves space and improves read performance. Indexes of elements within the collection remain unchanged.

If you call this member function before all elements have been added, insert (add) performance will not necessarily improve. After the B-tree has been compacted, adding an element will very likely cause one or more nodes to be added to the B-tree.

contains

Tests whether this ordered collection contains the specified element (or key).

```
1.   virtual ooBoolean contains(
       const ooHandle(ooObj) &objH) const;
2.   ooBoolean contains(
       const void *lookupVal) const;
```

Parameters

objH

Handle to the object to be tested for containment in this ordered collection.

lookupVal

Pointer to data that identifies the object to be tested for containment in this ordered collection.

Returns

`oocTrue` if this ordered collection contains an element (or key) equal to the specified object; otherwise, `oocFalse`.

Discussion

Variant 2 tests whether any element (or key) of this ordered collection is "equal" to the specified lookup data, as determined by the comparator for this collection. It is useful if this ordered collection has an application-defined comparator that can identify an element (or key) based on class-specific data. Because a list has no comparator, variant 2 is not relevant for lists.

depth

Gets the depth of this ordered collection's B-tree.

```
int depth() const;
```

Returns

The number of nodes between the root and a leaf node in this ordered collection's B-tree.

first

Finds the first element (or key) in this ordered collection.

```
virtual ooRef(ooObj) first() const;
```

Returns

If the elements of this ordered collection are persistent objects, an object reference to the first element; if the elements are key-value pairs, an object reference to the key of the first element.

See also

`last`
`get`

get

Finds the specified element of this ordered collection.

```
1.   ooRef(ooObj) get(const ooInt32 index) const;
2.   virtual ooRef(ooObj) get(const void *lookupVal) const = 0;
```

Parameters

index

The zero-based index of the desired element.

lookupVal

Pointer to data that identifies the desired element (or key).

Returns

(Variant 1) Finds the element at the specified index.

- If the elements of this ordered collection are persistent objects, variant 1 returns an object reference to the element whose index is *index*.
- If the elements are key-value pairs, variant 1 returns an object reference to the key of the element whose index is *index*.

(Variant 2) Finds the element (or key) that is equal to the specified lookup data, as determined by the comparator for this ordered collection.

- If the elements of this ordered collection are persistent objects, variant 2 returns an object reference to the element that is equal to *lookupVal*, or a null object reference if this ordered collection does not contain such an element.
- If the elements are key-value pairs, variant 2 returns an object reference to the value of the element whose key is equal to *lookupVal*, or a null object reference if this ordered collection does not contain such an element.

Discussion

Variant 2 is useful if this ordered collection has an application-defined comparator that can identify an element (or key) based on class-specific data. Because a list has no comparator, variant 2 is not relevant for lists.

See also

first
last

indexOf

Searches this ordered collection for the first element (or key) that is equal to the specified object.

```
virtual int indexOf(
     const ooHandle(ooObj) &objH,
     const int index = 0) const;
```

Parameters *objH*

Handle to the object whose index is to be found.

index

The zero-based index at which search should start.

Returns The index of the first element in this ordered collection, at or after the starting position, that is equal to (or whose key is equal to) the specified object. If no such element is found, this member function returns -1.

Discussion This member function searches forward in the collection, starting the search at the element whose index is *index*.

If elements of this ordered collection are persistent objects, this member function compares each element with the specified object; if elements are key-value pairs, this member function compares the key of each element with the specified object.

Search stops when a matching element is found; if more than one element matches, this member function finds the one closest to the beginning of the collection (but at or after the starting index).

See also `lastIndexOf`

isEmpty

Tests whether this ordered collection is empty.

```
virtual ooBoolean isEmpty() const;
```

Returns `oocTrue` if this ordered collection has no elements; otherwise, `oocFalse`.

iterator

Initializes a scalable-collection iterator to find the elements of this ordered collection.

```
virtual ooCollectionIterator *iterator() const;
```

Returns A pointer to a scalable-collection iterator for finding the elements of this ordered collection; the caller is responsible for deleting the iterator when it is no longer needed.

Discussion The returned iterator finds the elements as ordered in the collection.

You must delete the iterator when you have finished using it.

last

Finds the last element (or key) in this ordered collection.

```
ooRef(ooObj) last() const;
```

Returns — If the elements of this ordered collection are persistent objects, an object reference to the last element; if the elements are key-value pairs, an object reference to the key of the last element.

See also — first
get

lastIndexOf

Searches this ordered collection for the last element (or key) that is equal to the specified object.

```
1.   int lastIndexOf(
       const ooHandle(ooObj) &objH) const;
2.   int lastIndexOf(
       const ooHandle(ooObj) &objH,
       const int index) const;
```

Parameters

objH

Handle to the object whose index is to be found.

index

The zero-based index at which search should start.

Returns — The index of the last element in this ordered collection, at or before the starting position, that is equal to (or whose key is equal to) the specified object. If no such element is found, this member function returns -1.

Discussion — This member function searches backward in the collection. Variant 1 starts searching at the last element; variant 2 starts searching at the element whose index is *index.*

If elements of this ordered collection are persistent objects, this member function compares each element with the specified object; if elements are key-value pairs, this member function compares the key of each element with the specified object.

Search stops when a matching element is found; if more than one element matches, this member function finds the one closest to the end of the collection (but at or before the starting index).

See also — indexOf

refresh

Refreshes each container used internally by this ordered collection, except for the container in which the collection itself is stored.

```
virtual ooStatus refresh(ooMode &openMode) const;
```

Parameters *openMode*

Intended level of access to each refreshed container:

- Specify `oocRead` to open the container for read. This implicitly requests a read lock on the container.
- Specify `oocUpdate` to open the container for update (read and write). This implicitly requests an update lock on the container.

Returns `oocSuccess` if every container can be refreshed; otherwise, `oocError`.

Discussion You typically call this member function when you need to refresh your view of an ordered collection that you are reading in an MROW transaction. This member function calls `refreshOpen` on each container that is used internally by the ordered collection—that is, on the administrator, node, and array containers maintained by the ordered collection. This member function does not refresh the container in which the ordered collection itself is stored, nor does it necessarily refresh the containers that store the collection's elements.

remove

Removes the first occurrence of the specified object from this ordered collection.

```
virtual ooBoolean remove(
     const ooHandle(ooObj) &objH);
```

Parameters *objH*

Handle to the object to be removed.

Returns `oocTrue` if an element was removed; otherwise, `oocFalse`.

Discussion If the elements of this ordered collection are persistent objects, this member function removes the first element (if any) that is equal to the specified object. If the elements are key-value pairs, this member function removes the element (if any) whose key is the specified object.

size

Gets the size of this ordered collection.

```
virtual int size() const;
```

Returns The number of elements in this ordered collection

ooCollection Class

Inheritance: **ooObj->ooCollection**

Handle Class: `ooHandle(ooCollection)`

Object-Reference Class: `ooRef(ooCollection)`

The persistence-capable class `ooCollection` is the abstract base class for classes that represent *scalable collections* containing persistent objects.

See:

- "Reference Summary" on page 174 for an overview of member functions
- "Reference Index" on page 175 for a list of member functions

To use this class, your application must include the `ooCollections.h` header file. For UNIX linking information, see *Installation and Platform Notes for UNIX*.

About Scalable Collections

A *collection* is an aggregate that can contain a variable number of elements. A scalable collection can increase in size with minimal performance degradation.

Concrete classes derived from this class represent more specific kinds of collections of persistent objects and collections of key-value pairs. A collection of key-value pairs in which both the key and the value are persistent objects is called an *object map*.

- `ooTreeList` represents lists of persistent objects.
- `ooHashSet` represents unordered sets of persistent objects.
- `ooTreeSet` represents sorted sets of persistent objects.
- `ooHashMap` represents unordered object maps.
- `ooTreeMap` represents sorted object maps.

Because the ooCollection class is abstract, you never create instances of it; instead, you work with instances of its concrete derived classes. You should not create your own subclasses of this class.

Like other persistent objects, collections are normally manipulated through handles and object references.

Related Classes

Every collection derived from ooCollection uses a persistent object, called an *administrator*, an instance of a concrete class derived from ooAdmin. A collection's administrator manages the collections used by the collection's interal objects.

An instance of any derived class except ooTreeList uses a persistent object, called a *comparator*, an instance of a concrete class derived from ooCompare.

Objectivity/C++ includes one additional persistence-capable collection class that is not derived from ooCollection. ooMap represents name maps; a *name map* is a collection of key-value pairs in which the key is a string and the value is a persistent object.

For additional information about the difference among the various persistence-capable collection classes, see Chapter 11, "Persistent Collections," in the Objectivity/C++ programmer's guide.

Reference Summary

Adding and Removing Elements	add addAll clear remove removeAll removeAllDeleted retainAll
Getting Elements	get iterator keyIterator valueIterator
Getting Information	size

Finding Auxiliary Objects	admin comparator
Testing	contains containsAll isEmpty
Viewing in an MROW Transaction	refresh

Reference Index

add	Adds the specified object to this collection.
addAll	Adds all elements (or keys) in the specified collection to this collection.
admin	Finds the administrator for this collection.
clear	Sets this collection to an empty collection, removing any elements it contains.
comparator	Finds the comparator for this collection.
contains	Tests whether this collection contains the specified object.
containsAll	Tests whether this collection contains all elements (or keys) in the specified collection.
get	Finds the element of this collection that is equal to the specified lookup data, as determined by the comparator for this collection.
isEmpty	Tests whether this collection is empty.
iterator	Initializes a scalable-collection iterator to find the elements of this collection.
keyIterator	Initializes a scalable-collection iterator to find the keys or elements of this collection.
refresh	Refreshes each container used internally by this collection, except for the container in which the collection itself is stored.
remove	Removes the first occurrence of the specified object from this collection.
removeAll	Removes all elements (or keys) of the specified collection from this collection.

removeAllDeleted	Removes from this collection all persistent objects that have been deleted from the federated database.
retainAll	Retains all elements of this collection that are also in the specified collection, removing all other elements.
size	Gets the size of this collection.
valueIterator	Initializes a scalable-collection iterator to find the values or elements of this collection.

Member Functions

add

Adds the specified object to this collection.

```
virtual ooBoolean add(
     const ooHandle(ooObj) &objH) = 0;
```

Parameters

objH

Handle to the object to be added.

Returns

`oocTrue` if an element was added; otherwise, `oocFalse`.

See also

addAll
remove

addAll

Adds all elements (or keys) in the specified collection to this collection.

```
virtual ooBoolean addAll(
     const ooHandle(ooCollection) &collectionH);
```

Parameters

collectionH

Handle to the collection whose elements are to be added to this collection. If *collectionH* is the handle to a collection of key-value pairs and this is a collection of objects, only the keys of the specified collection are added to this collection.

Returns

`oocTrue` if any elements were added; otherwise, `oocFalse`.

See also

add
removeAll

admin

Finds the administrator for this collection.

```
virtual ooRef(ooAdmin) admin() const = 0;
```

Returns Object reference to the administrator for this collection.

clear

Sets this collection to an empty collection, removing any elements it contains.

```
virtual void clear();
```

See also `remove`
`removeAll`
`retainAll`

comparator

Finds the comparator for this collection.

```
virtual ooRef(ooCompare) comparator() const = 0;
```

Returns Object reference to the comparator for this collection, or null if this collection has a default comparator.

contains

Tests whether this collection contains the specified object.

```
1.   virtual ooBoolean contains(
       const ooHandle(ooObj) &objH) const = 0;
2.   virtual ooBoolean contains(
       const void *lookupVal) const;
```

Parameters *objH*

Handle to the element to be tested for containment in this collection.

lookupVal

Pointer to data that identifies the element to be tested for containment in this collection.

Returns `oocTrue` if this collection contains an element equal to the specified object; otherwise, `oocFalse`.

Discussion Variant 2 tests whether any element (or key) of this collection is "equal" to the specified lookup data, as determined by the comparator for this collection. It is useful if this collection has an application-defined comparator that can identify an element (or key) based on class-specific data.

See also `containsAll`

containsAll

Tests whether this collection contains all elements (or keys) in the specified collection.

```
virtual ooBoolean containsAll(
    const ooHandle(ooCollection) &collectionH) const;
```

Parameters *collectionH*

Handle to the collection whose elements (or keys) are to be tested for containment in this collection. Two elements (or keys) are equal if they are references for the same persistent object.

Returns `oocTrue` if this collection contains an element (or key) equal to each element (or key) of the specified collection; otherwise, `oocFalse`.

Discussion The meaning of this member function depends on whether the elements of the collections being compared are persistent objects or key-value pairs.

Elements of This Collection	Elements of Other Collection	Meaning
Persistent objects	Persistent objects	Tests whether this collection contains all elements of the specified collection
Persistent objects	Key-value pairs	Tests whether this collection contains all keys of the specified collection
Key-value pairs	Persistent objects	Tests whether all elements of the specified collection are keys of this collection
Key-value pairs	Key-value pairs	Tests whether all keys of the specified collection are also keys of this collection

See also `contains`

get

Finds the element of this collection that is equal to the specified lookup data, as determined by the comparator for this collection.

```
virtual ooRef(ooObj) get(const void *lookupVal) const = 0;
```

Parameters

lookupVal

Pointer to data that identifies the desired element.

Returns

If the elements of this collection are persistent objects, this member function returns an object reference to the element that is equal to *lookupVal*, or a null object reference if this collection does not contain such an element.

If the elements are key-value pairs, this member function returns an object reference to the value of the element whose key is equal to *lookupVal*, or a null object reference if this collection does not contain such an element.

Discussion

This member function is useful if this collection has an application-defined comparator that can identify an element (or key) based on class-specific data.

isEmpty

Tests whether this collection is empty.

```
virtual ooBoolean isEmpty() const = 0;
```

Returns

`oocTrue` if this collection has no elements; otherwise, `oocFalse`.

iterator

Initializes a scalable-collection iterator to find the elements of this collection.

```
virtual ooCollectionIterator *iterator() const = 0;
```

Returns

A pointer to a scalable-collection iterator for finding the elements of this collection; the caller is responsible for deleting the iterator when it is no longer needed.

Discussion

If this collection is ordered, the returned iterator finds the elements as ordered in the collection; if this collection is unordered, the iterator finds the elements in an undefined order.

You must delete the iterator when you have finished using it.

See also

`keyIterator`
`valueIterator`

keyIterator

Initializes a scalable-collection iterator to find the keys or elements of this collection.

```
virtual ooCollectionIterator *keyIterator() const;
```

Returns

A pointer to a scalable-collection iterator for finding the keys or elements of this collection; the caller is responsible for deleting the iterator when it is no longer needed.

Discussion

If the elements of this collection are key-value pairs, the returned iterator finds the keys of this collection; if the elements are persistent objects, the iterator finds this collection's elements.

If this collection is ordered, the returned iterator finds the keys or elements as ordered in the collection; if the collection is unordered, the iterator finds the keys or elements in an undefined order.

You must delete the iterator when you have finished using it.

See also

`iterator`
`valueIterator`

refresh

Refreshes each container used internally by this collection, except for the container in which the collection itself is stored.

```
virtual ooStatus refresh(ooMode &openMode) const = 0;
```

Parameters

openMode

Intended level of access to each refreshed container:

- Specify `oocRead` to open the container for read. This implicitly requests a read lock on the container.
- Specify `oocUpdate` to open the container for update (read and write). This implicitly requests an update lock on the container.

Returns

`oocSuccess` if every container can be refreshed; otherwise, `oocError`.

Discussion

You typically call this member function when you need to refresh your view of a collection that you are reading in an MROW transaction. This member function calls `refreshOpen` on each container that is used internally by the collection—that is, on the administrator, node, and array containers maintained by an ordered collection, or the administrator and hash-bucket containers maintained by an unordered collection. This member function does not refresh

the container in which the collection itself is stored, nor does it necessarily refresh the containers that store the collection's elements.

remove

Removes the first occurrence of the specified object from this collection.

```
virtual ooBoolean remove(
     const ooHandle(ooObj) &objH) = 0;
```

Parameters

objH

Handle to the object to be removed.

Returns

`oocTrue` if an element was removed; otherwise, `oocFalse`.

Discussion

If the elements of this collection are persistent objects, this member function removes the first element (if any) that is equal to the specified object. If the elements are key-value pairs, this member function removes the element (if any) whose key is the specified object.

See also

`add`
`clear`
`removeAll`
`retainAll`

removeAll

Removes all elements (or keys) of the specified collection from this collection.

```
virtual ooBoolean removeAll(
     ooHandle(ooCollection) &collectionH);
```

Parameters

collectionH

Handle to the collection whose elements are to be removed from this collection.

Returns

`oocTrue` if any elements were removed; otherwise, `oocFalse`.

Discussion Which elements are removed depends on whether elements of the two collections are persistent objects or key-value pairs.

Elements of This Collection	Elements of Other Collection	Removes From This Collection
Persistent objects	Persistent objects	All elements that are also elements of the specified collection
Persistent objects	Key-value pairs	All elements that are keys of the specified collection
Key-value pairs	Persistent objects	All elements whose keys are elements of the specified collection
Key-value pairs	Key-value pairs	All elements whose keys are also keys of the specified collection

See also
`addAll`
`clear`
`remove`
`removeAllDeleted`
`retainAll`

removeAllDeleted

Removes from this collection all persistent objects that have been deleted from the federated database.

```
virtual void removeAllDeleted();
```

Discussion You can call this member function to restore this collection's referential integrity.

See also
`clear`
`remove`
`removeAll`
`retainAll`

retainAll

Retains all elements of this collection that are also in the specified collection, removing all other elements.

```
virtual ooBoolean retainAll(
    ooHandle(ooCollection) &collectionH);
```

Parameters *collectionH*

Handle to the collection whose elements are to be retained in this collection.

Returns `oocTrue` if any elements were removed; otherwise, `oocFalse`.

Discussion Which elements are removed depends on whether elements of the two collections are persistent objects or key-value pairs.

Elements of This Collection	Elements of Other Collection	Removes From This Collection
Persistent objects	Persistent objects	All elements that are not also elements of the specified collection
Persistent objects	Key-value pairs	All elements that are not keys of the specified collection
Key-value pairs	Persistent objects	All elements whose keys are not elements of the specified collection
Key-value pairs	Key-value pairs	All elements whose keys are not also keys of the specified collection

See also `clear`
`remove`
`removeAll`

size

Gets the size of this collection.

```
virtual int size() const = 0;
```

Returns The number of elements in this collection.

valueIterator

Initializes a scalable-collection iterator to find the values or elements of this collection.

```
virtual ooCollectionIterator *valueIterator() const;
```

Returns A pointer to a scalable-collection iterator for finding the values or elements of this collection; the caller is responsible for deleting the iterator when it is no longer needed.

Discussion

If the elements of this collection are key-value pairs, the returned iterator finds the values of the collection; if the elements are persistent objects, the iterator finds the collection's elements.

If this collection is ordered, the iterator finds the values or elements as ordered in the collection; if the collection is unordered, the iterator finds the values or elements in an undefined order.

You must delete the iterator when you have finished using it.

See also

`iterator`
`keyIterator`

ooCollectionIterator Class

Inheritance: **`ooCollectionIterator`**

The non-persistence-capable class `ooCollectionIterator` is the abstract base class for *scalable-collection iterators*, which step through the objects in scalable persistent collections.

See:

- "Reference Summary" on page 187 for an overview of member functions
- "Reference Index" on page 188 for a list of member functions

To use this class, your application must include the `ooCollections.h` header file. For UNIX linking information, see *Installation and Platform Notes for UNIX*.

About Scalable-Collection Iterators

A scalable-collection iterator finds elements, keys, or values in a particular scalable collection. This group of objects is called 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 element* 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. That is, its index is -1 and it has no current element.

Working With a Scalable-Collection Iterator

Because the `ooCollectionIterator` class is abstract, you never create instances of it; instead, you work with instances of its concrete derived classes. You should not create your own subclasses of this class.

Obtaining a Scalable-Collection Iterator

During a transaction, you obtain a pointer to a scalable-collection iterator by calling a member function of a collection.

- Call the `iterator` member function of a collection of objects (list, unordered set, or sorted set) to obtain a scalable-collection iterator that finds the elements of that collection. If the collection is ordered, the iterator finds elements as ordered by the collection; if the collection is unordered, the iterator finds the elements in an undefined order.
- Call the `keyIterator` member function of a collection of key-value pairs (unordered object map or sorted object map) to obtain a scalable-collection iterator that finds the keys of that collection. If the collection is a sorted object map, the iterator finds the keys in their sorted order; if it is an unordered object map, the iterator finds the keys in an undefined order.
- Call the `valueIterator` member function of a collection of key-value pairs to obtain a scalable-collection iterator that finds the values of that collection. If the collection is a sorted object map, the iterator finds the value in the order in which their keys are sorted; if it is an unordered object map, the iterator finds the keys in an undefined order. In either case, the iterator returned by `valueIterator` finds the elements of the collection in the same order as does the iterator returned by `keyIterator`.

Using 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 step through the iteration set from beginning to end, call the `hasNext` member function for loop control to test whether additional elements remain. Within the loop, you make successive calls to the iterator's `next` member function to find each element. As you step through the iteration set from beginning to end, the current index increases until the scalable-collection iterator is positioned after the last element. At that point, the current index is the size of the iteration set and the scalable-collection iterator has no current element.

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. Within the loop, you make successive calls to the iterator's `previous` member function to find 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. At that point, the current index is back to -1 and the iterator has no current element.

Various member functions, such as `next` and `previous`, find persistent objects in the iterator's corresponding collection. These functions return type-independent object references or handles, which you can then cast to the appropriate type.

Modifying the Collection

Methods of the scalable-collection iterator allow you to modify the corresponding 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 element of the iteration set from the corresponding collection; the `set` member function replaces the current element with a specified object.

NOTE 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.

Reference Summary

Testing	`hasNext` `hasPrevious`

Finding Elements	`next`
	`current`
	`previous`
	`goToIndex`

Getting Indexes	`currentIndex`
	`nextIndex`
	`previousIndex`

Repositioning the Iterator	`goTo`
Modifying the Collection	`remove` `set`

Moving an Element	moveCurrentTo
Getting Information	collection currentValue

Reference Index

Member Functions

collection

Finds this scalable-collection iterator's corresponding collection.

```
virtual ooHandle(ooCollection) collection() = 0;
```

Returns Handle to the collection whose elements, keys, or values this scalable-collection iterator finds.

current

Finds the current object in this scalable-collection iterator's iteration set.

```
ooRef(ooObj) current() const;
```

Returns Object reference to this scalable-collection iterator's current element, or a null object reference if there is no current element (because the iterator is positioned before the first element or after the last element).

currentValue

Finds the value object paired with the current (key) object in this scalable-collection iterator's iteration set.

```
virtual ooRef(ooObj) currentValue() const;
```

Returns Object reference to the value of this scalable-collection iterator's current element, or a null object reference if there is no current element (because the iterator is positioned before the first element or after the last element).

Discussion You typically use this member function only if you are iterating over the keys of an unordered object map or a sorted object map, and you want to find the value that is paired with the current key. This function is equivalent to the `current` member function if you are iterating over a set or list, whose elements are not key-value pairs.

currentIndex

Gets the current index of this scalable-collection iterator.

```
virtual int currentIndex() = 0;
```

Returns The index in the iteration set at which this iterator is currently positioned.

goTo

Positions this scalable-collection iterator at the first occurrence of the specified object.

```
1.   virtual ooBoolean goTo(const ooHandle(ooObj) &objH) = 0;
2.   virtual ooBoolean goTo(const void *lookupVal) = 0;
```

Parameters

objH

Handle to the object to find in the iteration set.

lookupVal

Pointer to data that identifies the object to find in the iteration set.

Returns

`oocTrue` if the object was found, otherwise, `oocFalse`.

Discussion

You typically use this member function only if you are iterating over the elements of a sorted set or the keys of a sorted object map. In those cases, you are assured that the iteration set contains at most one occurrence of the specified object; in addition, the position of that object is meaningful because the elements of the iteration set are sorted. For example, if a sorted set has a comparator that sorts objects by a `name` data member, you might want to iterate starting with the objects whose names begin with the letter C. To do so, you could create an object with the name `"C"`, use that object as the parameter to `goTo`, then iterate from the new position.

Variant 2 finds the element that is equal to the specified lookup data, as determined by the comparator for the collection. It is useful if the collection has an application-defined comparator that can identify an element (or key) based on class-specific data. Because a list has no comparator, variant 2 is not relevant when iterating through the elements of a list.

If the specified object is found, this scalable-collection iterator is positioned at that object (as if the object had just been returned by `next`). To iterate starting with the specified object, you need to call `current` followed by successive calls to `next`.

If the specified object is not found, this scalable-collection iterator is positioned at the index just past where that object belongs in the sorted collection. In the preceding example, if the set contained two successive elements with names `"Byrnes"` and `"Cabbot"`, the current index would be set to the index of the element named `"Cabbot"`.

goToIndex

Positions this scalable-collection iterator at the specified index and finds the object at that position in the iteration set.

```
ooRef(ooObj) goToIndex(int index);
```

Parameters

index

Zero-based index at which to position this scalable-collection iterator.

Returns

Object reference to the object at the specified index in the iteration set, or a null object reference if *index* is out of bounds.

Discussion

You typically use this member function only if this scalable-collection iterator is initialized to find objects in an ordered collection; indexes are not meaningful when iterating through an unordered collection.

If the specified index is valid, it becomes this iterator's new current index. If the specified index is less than -1, this iterator is positioned before the first element. If the specified index is too large, this iterator is positioned after the last element.

This member function can be inefficient because it requires traversing the collection to find the indicated index. It locks all the collection's B-tree nodes up to and including the one containing the specified index. Whenever possible, you should use `goTo` instead of this member function.

hasNext

Tests whether this scalable-collection iterator has any elements after its current location.

```
virtual ooBoolean hasNext() const = 0;
```

Returns

`oocTrue` if the iteration set has at least one element after the current location; otherwise, `oocFalse`.

See also

`next`
`hasPrevious`

hasPrevious

Tests whether this scalable-collection iterator has any elements before its current location.

```
virtual ooBoolean hasPrevious();
```

Returns

`oocTrue` if the iteration set has at least one element before the current location; otherwise, `oocFalse`.

See also `previous`
`hasNext`

moveCurrentTo

Moves the current element of this scalable-collection iterator, clustering it with the specified object.

```
virtual ooStatus moveCurrentTo(const ooHandle(ooObj) &objH) = 0;
```

Parameters *objH*

A handle to the object with which to cluster the current element; must be a database, a persistent container, or a persistent basic object.

- If the clustering object is a database, the current element is moved to the default container of that database.
- If the clustering object is a persistent container, the current element is moved to that container.
- If the clustering object is a persistent basic object, the current element is moved to the container in which that object is stored. If possible, it will be stored on the same page as the clustering object or on a page close to the clustering object.

Returns `oocSuccess` if successful; otherwise, `oocError`.

Discussion The current element must be a basic object, not a container. This member function changes the object identifier of the current element. If the collection has a default comparator, this member function first removes the current element from the collection; then it moves that object and adds it back to the collection.

You should not call this member function in either of the following conditions:

- The current element is an element, key, or value of another persistent collection.
- The collection has an application-specific comparator that uses the object identifiers of the elements or keys.

next

Finds the next object in this scalable-collection iterator's iteration set.

```
ooRef(ooObj) next();
```

Returns Object reference to the next element in the iteration set, or a null object reference if this scalable-collection iterator is positioned after the last element in the iteration set.

Discussion If a next element exists, this member function returns a type-independent object reference to that element, which you must then cast to the appropriate type. It also increments this scalable-collection iterator's index.

See also `hasNext`
`nextIndex`
`previous`

nextIndex

Gets the index of this scalable-collection iterator's next element.

```
virtual int nextIndex();
```

Returns The index of the element that would be returned by the subsequent call to `next`.

Discussion The returned index is not necessarily a valid index for the iteration set. If this scalable-collection iterator is currently positioned at the last element of the iteration set, this member function returns the number of elements in the iteration set.

See also `next`
`hasNext`
`previousIndex`

previous

Finds the previous object in this scalable-collection iterator's iteration set.

```
ooRef(ooObj) previous();
```

Returns Object reference to the previous element in the iteration set, or a null object reference if this scalable-collection iterator is positioned before the first element in the iteration set.

Discussion If a previous element exists, this member function returns a type-independent object reference to that element, which you must then cast to the appropriate type. It also decrements this scalable-collection iterator's index.

See also `hasPrevious`
`previousIndex`
`next`

previousIndex

Gets the index of this scalable-collection iterator's previous element.

```
virtual int previousIndex();
```

Returns The index of the element that would be returned by the subsequent call to `previous`.

Discussion The returned index is not necessarily a valid index for the iteration set. If this scalable-collection iterator is currently positioned at the first element of the iteration set, this member function returns -1.

See also `previous`
`hasPrevious`
`nextIndex`

remove

Removes the current element from this scalable-collection iterator's corresponding collection.

```
virtual ooStatus remove() = 0;
```

Returns `oocSuccess` if successful; otherwise, `oocError`.

Discussion If this scalable-collection iterator is positioned before the first element or after the last element of the iteration set, this member function does nothing.

If this scalable-collection iterator is positioned at an element of the iteration set, the operation of this member function is as follows:

- If iterating through the elements of a collection of objects, remove the current element from the collection and position the iterator at the previous element. (If you removed the first element, the iterator is then positioned before the new first element.)
- If iterating through the keys of an object map, remove the element of the object map whose key is the current element of the iteration set. Then position the iterator at the previous element of the iteration set.
- If iterating through the values of an object map, set the value to null in the key-value pair whose value is the current element of the iteration set. Leave this iterator positioned at this same element.

set

Modifies this scalable-collection iterator's corresponding collection, replacing the current element with the specified object.

```
virtual ooStatus set(const ooHandle(ooObj) &objH);
```

Parameters

objH

Handle to the object that is to replace the current element.

Returns

`oocSuccess` if successful; otherwise, `oocError`.

Discussion

If this scalable-collection iterator is positioned before the first element or after the last element of the iteration set, this member function does nothing.

If this scalable-collection iterator is positioned at an element of the iteration set, the operation of this member function is as follows.

- If iterating through the elements of a list, replace the current element of the collection with the specified object.
- If iterating through the values of an object map, set the value to the specified object in the key-value pair whose value is the current element of the iteration set.
- If iterating through the elements of a set, the `set` operation is not supported; any call to `set` returns `oocError`.
- If iterating through the keys of an object map, the `set` operation is not supported; any call to `set` returns `oocError`.

ooCompare Class

Inheritance: **ooObj->ooCompare**

Handle Class: `ooHandle(ooCompare)`

Object-Reference Class: `ooRef(ooCompare)`

The persistence-capable class `ooCompare` is the abstract base class for all *comparator* classes.

See:

- "Reference Index" on page 199 for list of member functions

To use this class, your application must include the `ooCollections.h` header file. For UNIX linking information, see *Installation and Platform Notes for UNIX*.

About Comparators

A comparator is an object of a concrete derived class of `ooCompare`. It provides a comparison function for ordering elements of sorted collections and a hashing function for computing the hash codes for elements of unordered collections.

Every sorted or unordered collection uses the default comparator unless you assign an application-defined comparator to it; see "Working With a Comparator" on page 198. All application-defined comparators must be classes derived from `ooCompare`.

Comparators for Sorted Collections

Every sorted set and every sorted object map has a comparator. The comparator of a sorted collection defines a total ordering used by the underlying B-tree. Its `compare` member function is used to compare two persistent objects and indicate

their relative position in the total ordering. A sorted collection with a default comparator sorts persistent objects by their object identifiers (OIDs).

If elements of the sorted collection are persistent objects, the elements themselves are compared; if elements are key-value pairs, the elements' keys are compared.

For additional information about defining a comparate class for sorted collections, see "Comparator Class for Sorted Collections" on page 255 in the Objectivity/C++ programmer's guide.

Comparators for Unordered Collections

Every unordered set and every unordered object map has a comparator. The comparator of a scalable unordered collection supplies the hash function used by the underlying extendible hash table. Its `hash` member function computes the hash value for a persistent object; its `compare` member function tests two persistent objects for equality. An unordered collection with a default comparator computes hash values for persistent objects from their OIDs and compares objects for equality by comparing their OIDs.

If elements of the unordered collection are persistent objects, hash values are computed from the elements themselves and elements are compared for equality. If elements are key-value pairs, hash values are computed from the elements' keys and the keys are compared for equality.

For additional information about defining a comparate class for sorted collections, see "Comparator Class for Unordered Collections" on page 260 in the Objectivity/C++ programmer's guide.

Unique Identification of Collection Elements

An application-defined comparator class can optionally provide the ability to identify an element (or key) based on class-specific data. For additional information, see "Supporting Content-Based Lookup in a Sorted Collection" on page 258 and "Supporting Content-Based Lookup in an Unordered Collection" on page 264 in the Objectivity/C++ programmer's guide.

Working With a Comparator

Because the `ooCompare` class is abstract, you never create instances of it; instead, you work with instances of its concrete derived classes.

You may define your own comparator class with custom comparator and hashing functions. All application-defined comparator classes must derive from the `ooCompare` class.

If your application uses an application-defined comparator class, you create an instance of that class with a call to the new operator. 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*. The clustering directive in new operator specifies where in the federated database to store the new comparator.

A comparator is locked whenever you access its corresponding collection. To avoid locking conflicts, you typically cluster the comparator in a separate container. If the comparator is stored in the same container as the collection, applications may fail to get the necessary read lock on the comparator when another process is updating the collection.

After creating a comparator, you assign it to a collection by passing a handle to the comparator as a parameter to the constructor that creates the collection. Typically, an application creates a comparator for each collection; you should not assign the same comparator to more than one collection.

NOTE The persistent data for a persistent collection includes an object reference to its comparator. Your application should not explicitly save object references to any comparator. For example, you should not add a comparator to a collection or save an object reference to a comparator in a persistent data member of any persistent object.

Like other persistent objects, comparators are normally manipulated through handles or object references.

Reference Index

This class overloads `operator new` and `operator delete`, which behave as described for the `ooObj` class (page 431).

`compare`	Compares two persistent objects.
`hash`	Computes the hash value for a persistent object in a persistent collection that uses this comparator.

Member Functions

compare

Compares two persistent objects.

```
1.   virtual int compare(
       const ooHandle(ooObj) &obj1H,
       const ooHandle(ooObj) &obj2H) const;
2.   virtual int compare(
       const ooHandle(ooObj) &obj1H,
       const void *&lookupVal) const;
```

Parameters

obj1H

Handle to the first of the two persistent objects to be compared. This object is an element (or key) of a persistent collection that uses this comparator.

obj2H

Handle to the second of the two persistent objects to be compared.

lookupVal

Pointer to data that identifies an element (or key) of the persistent collection containing *obj1H*; the identified element (or key) is the second of the two persistent objects to be compared.

Returns

A negative integer if the first object is less than (sorts before) the second; zero if the two objects are equal; a positive integer if the first object is greater than (sorts after) the second.

Discussion

Variant 1 compares the two specified persistent objects; the default implementation of variant 1 compares their OIDs.

Variant 2 is called by member functions that accept arbitrary data to identify an element. The default implementation of variant 2 casts its `void *&` parameter to a handle of type `ooHandle(ooObj)` and compares the OIDs of the two objects.

hash

Computes the hash value for a persistent object in a persistent collection that uses this comparator.

```
1.   virtual int hash(
       const ooHandle(ooObj) &objH) const;
2.   virtual int hash(
       const void *&lookupVal) const;
```

Parameters

objH

Handle to the persistent object from which to calculate the hash value. This object is an element (or key) of a persistent collection that uses this comparator.

lookupVal

Pointer to data that identifies a persistent object whose hash value is to be computed.

Returns

The hash value for the specified object.

Discussion

Variant 1 computes the hash value for the specified persistent object. The default implementation of this variant computes the hash value from the object's OID.

Variant 2 is called by member functions that accept arbitrary data to identify an element. The specified data identifies the element whose hash value is to be computed. The default implementation of variant 1 casts its `void *&` parameter to a handle of type `ooHandle(ooObj)` and computes the hash value from the object's OID.

ooContext Class

Inheritance: **ooContext**

The non-persistence-capable class ooContext represents an *Objectivity context*—the operating context that enables a thread to execute a series of transactions with an Objectivity / DB federated database.

See:

- "Reference Summary" on page 204 for an overview of member functions
- "Reference Index" on page 204 for a list of member functions

About Objectivity Contexts

An Objectivity context is a complete set of Objectivity / DB-managed data and memory resources, including an Objectivity / DB cache, the current values of Objectivity / C++ global variables, current error and message handlers, and so on. Every Objectivity / DB application has at least one Objectivity context, which belongs to the main thread (the thread that starts implicitly when you start the application). A multithreaded application typically provides an Objectivity context for each additional thread that executes Objectivity / DB transactions, although it is possible for the same Objectivity context to be reused by several threads in turn. Objectivity contexts are the only kind of transient Objectivity / DB object that you can pass between threads.

The ooContext class provides a public constructor and destructor for creating and destroying Objectivity contexts. Instances of the class are also created and destroyed implicitly by the ooInitThread and ooTermThread global functions, respectively.

Every thread that is initialized for Objectivity / DB has a *current Objectivity context*, which is the context that Objectivity / DB uses when the thread executes database operations. The ooInitThread global function sets a thread's current Objectivity context for the first time; the ooContext class provides an interface

for changing it. An uninitialized thread has no current Objectivity context associated with it.

Objectivity contexts are typically referenced through pointers. A null `ooContext` pointer is called a *null Objectivity context*. A thread's current Objectivity context may be null or nonnull; however, a thread with a null context cannot execute Objectivity/DB operations other than context operations and destructors.

For more information about Objectivity contexts, see Chapter 3, "Objectivity/DB Initialization," and Chapter 5, "Multithreaded Objectivity/C++ Applications," in the Objectivity/C++ programmer's guide.

Reference Summary

Creating and Destroying an Objectivity Context	`ooContext` `~ooContext`

Accessing an Objectivity Context	`current`
	`setCurrent`
	`setCurrentShared`

Reference Index

`ooContext`	Constructs a new Objectivity context whose Objectivity/DB cache has the specified initial and maximum numbers of buffer pages.
`~ooContext`	Destructor for the class `ooContext`.
`current`	Returns a pointer to the current Objectivity context for the executing thread; returns 0 if this thread has a null Objectivity context.
`setCurrent`	Sets the current Objectivity context for the executing thread.
`setCurrentShared`	For experienced users only. Sets the current Objectivity context for the executing thread, allowing multiple threads to share the same Objectivity context.

Constructors and Destructors

ooContext

Constructs a new Objectivity context whose Objectivity / DB cache has the specified initial and maximum numbers of buffer pages.

```
ooContext(uint32 nPages = 200, uint32 nMaxPages = 500);
```

Parameters

npages

Initial number of buffer pages to be allocated for each buffer pool in the Objectivity / DB cache.

nMaxPages

Maximum number of buffer pages that can be allocated for each buffer pool in the Objectivity / DB cache. This number is limited by the amount of available swap space. Specifying 0 allows the cache to grow as needed, up to the amount of available swap space.

Discussion

You use the `ooInit` global function to specify nondefault cache sizes for the Objectivity context in the main thread; you use the `ooContext` constructor to specify nondefault cache sizes for any additional Objectivity context. See `ooInit` for more information about initial and maximum Objectivity / DB cache sizes.

~ooContext

Destructor for the class `ooContext`.

```
~ooContext();
```

Discussion

An error is signaled if you call this destructor for an Objectivity context that is the current context for any thread.

Member Functions

current

Returns a pointer to the current Objectivity context for the executing thread; returns 0 if this thread has a null Objectivity context.

```
static ooContext *current();
```

setCurrent

Sets the current Objectivity context for the executing thread.

```
static void setCurrent(ooContext *context);
```

Parameters *context*

Pointer to the Objectivity context to be set. You can specify 0 to set the thread's current Objectivity context to the null context.

Discussion An error is signaled if the specified Objectivity context is the current context for another thread.

A thread may not invoke Objectivity / DB operations (other than context operations and destructors) until a nonnull context is set.

setCurrentShared

For experienced users only. Sets the current Objectivity context for the executing thread, allowing multiple threads to share the same Objectivity context.

```
static void setCurrentShared(ooContext *context);
```

Parameters *context*

Pointer to the Objectivity context to be set.

Discussion This member function is the same as `setCurrent`, except that the specified Objectivity context may be the current context for another thread.

WARNING Allowing multiple threads to share the same Objectivity context is a dangerous operation. To prevent data corruption, you must guarantee that no two threads execute Objectivity / C++ operations (including constructors and destructors) in the same Objectivity context at the same time. For example, you can use mutexes to achieve this.

---

ooContObj Class

Inheritance: **ooObj->ooContObj**

Handle Class: ooHandle(ooContObj)

Object-Reference Class: ooRef(ooContObj)

The persistence-capable class ooContObj represents a standard Objectivity / DB *container*, and serves as the base class for all container classes. Together, the ooContObj class and its corresponding handle and object-reference classes define the behavior of containers.

See:

- "Reference Summary" on page 211 for an overview of ooContObj member functions
- "Reference Index" on page 211 for a list of ooContObj member functions

For operations performed through a container handle or object reference, see:

- "Reference Summary" on page 515

About Containers

A container is the third highest level in the Objectivity / DB storage hierarchy; every database contains one or more containers, and every container can contain one or more basic objects.

Containers serve a number of purposes. They are used:

- To group basic objects. Basic objects within a container are physically clustered together in memory and on disk, so access to 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 the container are also locked. This enables the lock server to manage relatively few container-level locks rather than potentially millions or billions of object-level locks.
- Optionally, to maintain application-specific data.

Containers function both as *storage objects* (because basic objects can be clustered in them) and as *persistent objects* (for example, because they can be referenced in attributes, elements of a persistent collection, scope-named, and so on).

A container is physically maintained within a database file, and may optionally have a system name in addition to any number of scope names; a container's system name cannot be changed.

(*FTO*) Every container is controlled by an autonomous partition—usually the partition in which the container's database resides. If an application moves the control of a container to a different partition, the container is physically moved to the system database file in that partition.

Kinds of Container

An application normally creates *standard containers* to control the clustering and locking of basic objects. Standard containers are created as instances of class `ooContObj`.

An application can also derive its own persistence-capable container classes from `ooContObj`. Instances of such classes, called *application-defined containers*, can have associations and attributes for persistent data). Like any persistence-capable class, an application-defined container class must be defined in a DDL file and its definition must be processed by the DDL processor.

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`.

Exactly one *default container* is created automatically by Objectivity / DB for each database; default containers are created as instances of class `ooDefaultContObj`.

Every container is created as either *hashed* or *nonhashed*. If a container is hashed, the container (and any object in it) can be used as a scope for naming objects. A non-hashed container occupies less storage than a hashed container, but does not support scope naming.

Container Structure

A container consists of a *container object* that manages a particular set of *storage pages* allocated within a particular database file. The storage pages contain the basic objects that are clustered in the container:

- Every small basic object resides entirely within a single storage page, possibly along with other small basic objects.
- Every large basic object spans multiple storage pages; the first of these pages (called a header page) contains certain housekeeping information and the remaining pages contain the object's persistent data.

A storage page is considered a *logical page* if it contains one or more small objects *or* the header information for a large object. Within each container, the logical pages are numbered; the object identifier of a basic object includes the number of the logical page on which the object resides.

The container object has various responsibilities, which include:

- Maintaining a *page map* that records the physical location of each logical page. The page map enables the container object to use object identifiers to quickly locate basic objects in the container.
- Managing the allocation of pages according to the initial number and growth characteristics you specify when the container is created.
- Maintaining a hash table, if the container is hashed for scope naming.
- Storing persistent data, if the container is an instance of an application-defined container class that defines attributes or associations.

Working With Containers

Standard, application-defined, and garbage-collectible containers are created and deleted from within an application. An application:

- Creates a container using the constructor and `operator new` defined by the appropriate container class—namely, `ooContObj`, the application-defined class *`appClass`*, or `ooGCContObj`.

  The resulting container-class instance represents the container object, which, in turn, manages the container's pages in the database.
- Deletes a container by calling the `ooDelete` global function.

As is the case for any persistent object, you specify whether a container is to be transient or persistent when you create it. Containers should normally be persistent; a clustering directive on `operator new` specifies where to locate a new persistent container in the federated database. A transient container is

simply the container object without any allocated storage pages; it can have application-specific persistent data, but basic objects cannot be clustered in it.

To work with a new container, an application *must* assign the result of `operator new` to a handle. For example, a new standard container is normally assigned to an instance of `ooHandle(ooContObj)`; a new application-defined container of class *`appClass`* is normally assigned to an instance of the corresponding handle class `ooHandle(`*`appClass`*`)`.

Similarly, to identify and work with an existing container, the application must open it through an appropriate container handle or object reference; multiple handles and object references can be set to reference the same container. Note that opening a container reads just the container object into memory; pages managed by the container are fetched only when the basic objects on those pages are opened.

The application operates on an open container by:

- Calling various member functions on any of the referencing handles or object references.
- Passing any of the referencing handles or object references to various global functions or member functions of other classes.

The general handle and object-reference classes for containers (*`ooRefHandle`*`(ooContObj)`) provide the primary interface for operating on existing containers, including member functions for opening, locking, getting information, and finding objects from a referenced container. The type-specific handle and object-reference classes *`ooRefHandle`*`(`*`appClass`*`)` inherit these member functions from *`ooRefHandle`*`(ooContObj)`, redefining them wherever type-specific behavior or parameters are required.

In addition, every application-defined container class *`appClass`* has:

- DDL-generated member functions for creating, deleting, and accessing any associations defined by the class.
- An `operator new` defined by `ooContObj`.
- The member functions defined by `ooObj` that apply to containers (see “Reference Summary” on page 211). These include member functions for identifying a container’s type, for obtaining a handle to the container, or for opening the container for update.
- Data members and member functions specific to *`appClass`*.

You can call these members directly from within a member function of *`appClass`* or indirectly through the indirect member-access operator (`->`) on an *`appClass`* handle or object reference.

NOTE ooObj also defines member functions for moving, copying, and versioning basic objects. These member functions are *not* available on containers, which cannot be moved, copied, or versioned.

Reference Summary

In the following table, the member functions indicated as *(inherited)* are defined by ooObj and are available for containers; they are documented with the ooObj class (page 431). They are the *only* ooObj member functions that apply to containers.

Creating and Deleting a Container	ooContObj operator new operator delete *(inherited)*
Working With the Container	ooGetTypeN *(inherited)* ooGetTypeName *(inherited)* ooIsKindOf *(inherited)* ooThis ooUpdate *(inherited)* ooValidate *(inherited)*
ODMG Interface	operator new

Reference Index

ooContObj	Default constructor that constructs a new standard container.
ooThis	Sets an object reference or handle to reference this container.
operator new	Creates a new standard container.

Constructors

ooContObj

Default constructor that constructs a new standard container.

```
ooContObj();
```

Operators

operator new

Creates a new standard container.

1.
```
void *operator new(
  size_t);
```

2.
```
void *operator new(
  size_t,
  const ooRefHandle(ooObj) &near);
```

3.
```
void *operator new(
  size_t,
  const ooObj *near);
```

(ODMG) 4.
```
void *operator new(
  size_t,
  ooRefHandle(ooObj) near,
  const char *type);
```

(ODMG) 5.
```
void *operator new(
  size_t,
  d_Database *near,
  const char *type = 0);
```

6.
```
void *operator new(
  size_t,
  const char *contSysName,
  const uint32 hash,
  const uint32 initPages,
  const uint32 percentGrowth);
```

```
7.  void *operator new(
      size_t,
      const char *contSysName,
      const uint32 hash,
      const uint32 initPages,
      const uint32 percentGrowth,
      const ooRefHandle(ooObj) &near);
8.  void *operator new(
      size_t,
      const char *contSysName,
      const uint32 hash,
      const uint32 initPages,
      const uint32 percentGrowth,
      const ooObj *near);
```

Parameters

size_t

Do not specify; this parameter is automatically initialized by the compiler with the size of the class type in bytes.

near

Specifies whether the new container is to be persistent or transient. If the new container is persistent, *near* is the clustering directive that specifies the container's location in the federated database.

To create a transient container, specify 0 or a pointer to a transient object.

To create a persistent container, choose one of the following alternatives as the clustering directive:

- Omit *near* (variants 1 and 6)
- Specify an object reference, handle, or pointer to a database, container, or persistent basic object (variants 2, 3, 4, 7, and 8). *near* may not reference a federated database or an autonomous partition.
- (*ODMG*) Specify a pointer to a `d_Database` object that references the federated database (variant 5).

When you create a persistent container:

- If you omit *near*, the new container is created in the most recently opened or created database. An error is signalled if no such database exists.
- If *near* references a database, the new container is created in that database.
- If *near* references a container or basic object, the new container is created in the database that contains the referenced container or basic object.

- (*ODMG*) If *near* specifies a valid `d_Database` object, the new container is created in the most recently opened or created database in the federation. If no such database exists, a database called `default_odmg_db` is used; if necessary, this database is created.

type

(*ODMG*) This parameter is ignored. By convention it is the name of the class you are instantiating.

contSysName

System name of the new container. Specify 0 to omit the system name. This name must be unique within the database.

hash

Determines whether to create a hashed container. You must create a hashed container if you intend to use it or any object in it as a scope for naming objects, or if you intend to create keyed objects in the container.

- Specify 0 to create a nonhashed container.
- Specify 1 or greater to create a hashed container.

  The number you specify is the clustering factor for any keyed objects created in the container. A clustering factor is the number of sequentially keyed objects to be placed onto a page. A clustering factor of 1 maximally distributes keyed objects across pages. A higher number means fewer pages need to be read when finding sequences of keyed objects.

initPages

Initial number of logical pages to allocate for the new container. Specify 0 to use the system default value (4 pages for a hashed container, and 2 pages for a nonhashed container). The maximum value is 65535.

percentGrowth

Amount by which the new container may grow when it needs to accommodate more basic objects, expressed as a percentage of its current size. Specify 0 to use the system default value (10%).

Returns

Memory pointer to the new container. This pointer is null if an error occurs during the creation of the container.

Discussion

Variants 1 through 5 each create an unnamed, nonhashed container with an initial size of 2 pages and a growth factor of 10%—as if parameters *contSysName*, *hash*, *initPages*, and *percentGrowth* were each set to 0.

When you create a new persistent container, you must:

- Use `operator new` in an update transaction. The new container is made permanent on disk when the transaction commits or is checkpointed. If the transaction is aborted, the container is not created.

- Assign the result of `operator new` directly to a handle. You can verify the creation of the container by checking whether the handle is null. (If an object reference is desired, you can then assign the handle to an object reference.)

WARNING 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.

See also `ooNewConts` global macro

Member Functions

ooThis

Sets an object reference or handle to reference this container.

```
1.  ooHandle(ooContObj) ooThis() const;
2.  ooRef(ooContObj) &ooThis(
      ooRef(ooContObj) &object) const;
3.  ooHandle(ooContObj) &ooThis(
      ooHandle(ooContObj) &object) const;
```

Parameters *object*

Object reference or handle to be set to this object.

Returns Object reference or handle to this object.

Discussion You normally use `ooThis` in a member function of an application-defined container class; when such a member function is called on a container of the class, `ooThis` provides the member function with an object reference or handle to the container. The member function can then perform operations on the container that are available only through an object reference or handle.

When called without a *container* parameter, `ooThis` allocates a new handle and returns it. Otherwise, `ooThis` returns the object reference or handle that is passed to it.

See also *appClass*::`ooThis`

ooConvertInObject Class

Inheritance: **ooConvertInObject**

The non-persistence-capable class `ooConvertInObject` represents an *unconverted object*—an object that is affected by schema evolution and that is about to be converted to its new shape.

See:

- "Reference Summary" on page 218 for an overview of member functions
- "Reference Index" on page 218 for a list of member functions

About ooConvertInObject Instances

The `ooConvertInObject` class enables you to get the values of primitive data members in unconverted objects. You use instances of this class in any *conversion functions* you create to augment the object conversion process. When a conversion function is called during object conversion, two objects are passed to it:

- A representation of an existing, unconverted object (an instance of `ooConvertInObject`)
- A representation of the object after it has been converted (an instance of `ooConvertInOutObject`)

You use member functions of `ooConvertInObject` to get values from the unconverted object's primitive data members. From these pre-conversion values, you can compute data-member values to be set after the object has been converted. You use member functions of `ooConvertInOutObject` to set data-member values in the converted object.

The `ooConvertInObject` class provides access to primitive data members, but not to variable-length data members (VArrays), associations, or object references. You can access primitive data members that are defined:

- Directly in the class whose objects are being converted (see the `get`*Type* member functions).
- In a base class of the class whose objects are being converted (see the `getOldBaseClass` member function).
- In an embedded class of the class whose objects are being converted (see the `getOldDataMember` member function).

See Chapter 19, "Object Conversion," of the Objectivity/C++ programmer's guide for an example that uses this class.

Reference Summary

Accessing Primitive Data Members	`getFloat32`
	`getFloat64`
	`getInt8`
	`getInt16`
	`getInt32`
	`getInt64`
	`getUInt8`
	`getUInt16`
	`getUInt32`
	`getUInt64`

Accessing a Base or Embedded Class	`getOldBaseClass` `getOldDataMember`

Reference Index

`getFloat32`	Gets the value of the specified `float32` data member in an unconverted object.
`getFloat64`	Gets the value of the specified `float64` data member in an unconverted object.
`getInt8`	Gets the value of the specified `int8` data member in an unconverted object.

getInt16	Gets the value of the specified int16 data member in an unconverted object.
getInt32	Gets the value of the specified int32 data member in an unconverted object.
getInt64	Gets the value of the specified int64 data member in an unconverted object.
getOldBaseClass	Gets the part of an unconverted object that is inherited from the specified base class.
getOldDataMember	Gets the part of an unconverted object that belongs to the specified embedded object.
getUInt8	Gets the value of the specified uint8 data member in an unconverted object.
getUInt16	Gets the value of the specified uint16 data member in an unconverted object.
getUInt32	Gets the value of the specified uint32 data member in an unconverted object.
getUInt64	Gets the value of the specified uint64 data member in an unconverted object.

Member Functions

getFloat32

Gets the value of the specified float32 data member in an unconverted object.

```
ooStatus getFloat32(
     const char *memberName,
     float32 &value);
```

Parameters

memberName

Name of the data member whose value is to be obtained.

value

Name of the float32 variable in which the value is to be returned.

Returns

oocSuccess if successful, or oocError if the data-member name is not valid.

getFloat64

Gets the value of the specified `float64` data member in an unconverted object.

```
ooStatus getFloat64(
     const char *memberName,
     float64 &value);
```

Parameters

memberName

Name of the data member whose value is to be obtained.

value

Name of the `float64` variable in which the value is to be returned.

Returns

`oocSuccess` if successful, or `oocError` if the data-member name is not valid.

getInt8

Gets the value of the specified `int8` data member in an unconverted object.

```
ooStatus getInt8(
     const char *memberName,
     int8 &value);
```

Parameters

memberName

Name of the data member whose value is to be obtained.

value

Name of the `int8` variable in which the value is to be returned.

Returns

`oocSuccess` if successful, or `oocError` if the data-member name is not valid.

getInt16

Gets the value of the specified `int16` data member in an unconverted object.

```
ooStatus getInt16(
     const char *memberName,
     int16 &value);
```

Parameters

memberName

Name of the data member whose value is to be obtained.

value

Name of the `int16` variable in which the value is to be returned.

Returns

`oocSuccess` if successful, or `oocError` if the data-member name is not valid.

getInt32

Gets the value of the specified `int32` data member in an unconverted object.

```
ooStatus getInt32(
     const char *memberName,
     int32 &value);
```

Parameters

memberName

Name of the data member whose value is to be obtained.

value

Name of the `int32` variable in which the value is to be returned.

Returns

`oocSuccess` if successful, or `oocError` if the data-member name is not valid.

getInt64

Gets the value of the specified `int64` data member in an unconverted object.

```
ooStatus getInt64(
     const char *memberName,
     int64 &value);
```

Parameters

memberName

Name of the data member whose value is to be obtained.

value

Name of the `int64` variable in which the value is to be returned.

Returns

`oocSuccess` if successful, or `oocError` if the data-member name is not valid.

getOldBaseClass

Gets the part of an unconverted object that is inherited from the specified base class.

```
ooStatus getOldBaseClass(
     const char *baseClassName,
     ooConvertInObject &basePartOfExistObj);
```

Parameters

baseClassName

Name of a base class of the class whose objects are being converted.

basePartOfExistObj

Name of the `ooConvertInObject` variable in which the inherited part is to be returned.

Returns `oocSuccess` if successful, or `oocError` if the path is not valid.

Discussion You use this member function when you need to get the value of an inherited primitive data member of an unconverted object. First, you use this member function to get the part of the object that is inherited from the specified base class. Then, you use the appropriate `get`*`Type`* member function on the inherited part (accessed through the *`basePartOfExistObj`* variable) to get the value of the desired data member.

getOldDataMember

Gets the part of an unconverted object that belongs to the specified embedded object.

```
ooStatus getOldDataMember(
     const char *memberName,
     ooConvertInObject &embeddedPartOfExistObj);
```

Parameters *`memberName`*

Name of the data member that specifies the desired embedded object.

`embeddedPartOfExistObj`

Name of the `ooConvertInObject` variable in which the embedded part is to be returned.

Returns `oocSuccess` if successful, or `oocError` if the *`memberName`* is not valid.

Discussion You use this member function when you need to get the value of a primitive data member of an object that is embedded as a data member of an unconverted object. First, you use this member function to get the desired embedded part of the unconverted object. Then, you use the appropriate `get`*`Type`* member function on the embedded part (accessed through the *`embeddedPartOfExistObj`* variable) to get the value of the desired primitive data member.

getUInt8

Gets the value of the specified `uint8` data member in an unconverted object.

```
ooStatus getUInt8(
     const char *memberName,
     uint8 &value);
```

Parameters *`memberName`*

Name of the data member whose value is to be obtained.

value

Name of the uint8 variable in which the value is to be returned.

Returns oocSuccess if successful, or oocError if the data-member name is not valid.

getUInt16

Gets the value of the specified uint16 data member in an unconverted object.

```
ooStatus getUInt16(
     const char *memberName,
     uint16 &value);
```

Parameters *memberName*

Name of the data member whose value is to be obtained.

value

Name of the uint16 variable in which the value is to be returned.

Returns oocSuccess if successful, or oocError if the data-member name is not valid.

getUInt32

Gets the value of the specified uint32 data member in an unconverted object.

```
ooStatus getUInt32(
     const char *memberName,
     uint32 &value);
```

Parameters *memberName*

Name of the data member whose value is to be obtained.

value

Name of the uint32 variable in which the value is to be returned.

Returns oocSuccess if successful, or oocError if the data-member name is not valid.

getUInt64

Gets the value of the specified uint64 data member in an unconverted object.

```
ooStatus getUInt64(
     const char *memberName,
     uint64 &value);
```

Parameters *memberName*

Name of the data member whose value is to be obtained.

value

Name of the `uint64` variable in which the value is to be returned.

Returns `oocSuccess` if successful, or `oocError` if the data-member name is not valid.

ooConvertInOutObject Class

Inheritance: **ooConvertInObject->ooConvertInOutObject**

The non-persistence-capable class `ooConvertInOutObject` represents a *converted object*—an object that is affected by schema evolution and that has been converted to its new shape.

See:

- "Reference Summary" on page 226 for an overview of member functions
- "Reference Index" on page 226 for a list of member functions

About ooConvertInOutObject Instances

The `ooConvertInOutObject` class enables you to set the values of data members in objects that have been converted as a result of schema evolution. You use instances of this class in any *conversion functions* you create to augment the object conversion process. When a conversion function is called during object conversion, two objects are passed to it:

- A representation of an existing, unconverted object (an instance of `ooConvertInObject`)
- A representation of the object after it has been converted (an instance of `ooConvertInOutObject`)

You use member functions of `ooConvertInObject` to get values from the unconverted object's primitive data members. From these pre-conversion values, you can compute data-member values to be set after the object has been converted. You use member functions of `ooConvertInOutObject` to set data-member values in the converted object.

The `ooConvertInOutObject` class allows you to set primitive data members. You can access data members that are defined:

- Directly in the class whose objects are being converted (see the `set`*`Type`* member functions).
- In a base class of the class whose objects are being converted (see the `getNewBaseClass` member function).
- In an embedded class of the class whose objects are being converted (see the `getNewDataMember` member function).

See Chapter 19, "Object Conversion," in the Objectivity/C++ programmer's guide for an example that uses this class.

Reference Summary

Setting Primitive Data Members	`setFloat32`
	`setFloat64`
	`setInt8`
	`setInt16`
	`setInt32`
	`setInt64`
	`setUInt8`
	`setUInt16`
	`setUInt32`
	`setUInt64`

Accessing a Base or Embedded Class	`getNewBaseClass` `getNewDataMember`

Reference Index

`getNewBaseClass`	Gets the part of a converted object that is inherited from the specified base class.
`getNewDataMember`	Gets the part of a converted object that belongs to the specified embedded object.
`setFloat32`	Sets the value of the specified `float32` data member in a converted object.
`setFloat64`	Sets the value of the specified `float64` data member in a converted object.

setInt8	Sets the value of the specified int8 data member in a converted object.
setInt16	Sets the value of the specified int16 data member in a converted object.
setInt32	Sets the value of the specified int32 data member in a converted object.
setInt64	Sets the value of the specified int64 data member in a converted object.
setUInt8	Sets the value of the specified uint8 data member in a converted object.
setUInt16	Sets the value of the specified uint16 data member in a converted object.
setUInt32	Sets the value of the specified uint32 data member in a converted object.
setUInt64	Sets the value of the specified uint64 data member in a converted object.

Member Functions

getNewBaseClass

Gets the part of a converted object that is inherited from the specified base class.

```
ooStatus getNewBaseClass(
    const char *baseClassName,
    ooConvertInOutObject &basePartOfConvObj);
```

Parameters

baseClassName

Name of a base class of the class whose objects are being converted.

basePartOfConvObj

Name of the ooConvertInOutObject variable in which the inherited part is to be returned.

Returns

oocSuccess if successful, or oocError if the base class path is not valid.

Discussion

You use this member function when you need to set the value of an inherited primitive data member of a converted object. First, you use this member function to get the part of the object that is inherited from the specified base class. Then, you use the appropriate set*Type* member function on the inherited part

(accessed through the *basePartOfConvObj* variable) to set the value of the desired data member.

See also `ooConvertInObject::getOldBaseClass`

getNewDataMember

Gets the part of a converted object that belongs to the specified embedded object.

```
ooStatus getNewDataMember(
     const char *memberName,
     ooConvertInOutObject &embeddedPartOfConvObj);
```

Parameters

memberName

Name of the data member that specifies the desired embedded object.

embeddedPartOfConvObj

Name of the `ooConvertInOutObject` variable in which the embedded part is to be returned.

Returns `oocSuccess` if successful, or `oocError` if the data-member name is not valid.

Discussion You use this member function when you need to set the value of a primitive data member of an object that is embedded as a data member of a converted object. First, you use this member function to get the desired embedded part of the converted object. Then, you use the appropriate `set`*Type* member function on the embedded part (accessed through the *embeddedPartOfConvObj* variable) to set the value of the desired primitive data member.

setFloat32

Sets the value of the specified `float32` data member in a converted object.

```
ooStatus setFloat32(
     const char *memberName,
     const float32 value);
```

Parameters

memberName

Name of the data member whose value is to be set.

value

Variable containing the desired value.

Returns `oocSuccess` if successful, or `oocError` if the data-member name is not valid.

setFloat64

Sets the value of the specified `float64` data member in a converted object.

```
ooStatus setFloat64(
     const char *memberName,
     const float64 value);
```

Parameters

memberName

Name of the data member whose value is to be set.

value

Variable containing the desired value.

Returns

`oocSuccess` if successful, or `oocError` if the data-member name is not valid.

setInt8

Sets the value of the specified `int8` data member in a converted object.

```
ooStatus setInt8(
     const char *memberName,
     const int8 value);
```

Parameters

memberName

Name of the data member whose value is to be set.

value

Variable containing the desired value.

Returns

`oocSuccess` if successful, or `oocError` if the data-member name is not valid.

setInt16

Sets the value of the specified `int16` data member in a converted object.

```
ooStatus setInt16(
     const char *memberName,
     const int16 value);
```

Parameters

memberName

Name of the data member whose value is to be set.

value

Variable containing the desired value.

Returns

`oocSuccess` if successful, or `oocError` if the data-member name is not valid.

setInt32

Sets the value of the specified `int32` data member in a converted object.

```
ooStatus setInt32(
     const char *memberName,
     const int32 value);
```

Parameters

memberName

Name of the data member whose value is to be set.

value

Variable containing the desired value.

Returns

`oocSuccess` if successful, or `oocError` if the data-member name is not valid.

setInt64

Sets the value of the specified `int64` data member in a converted object.

```
ooStatus setInt64(
     const char *memberName,
     const int64 value);
```

Parameters

memberName

Name of the data member whose value is to be set.

value

Variable containing the desired value.

Returns

`oocSuccess` if successful, or `oocError` if the data-member name is not valid.

setUInt8

Sets the value of the specified `uint8` data member in a converted object.

```
ooStatus setUInt8(
     const char *memberName,
     const uint8 value);
```

Parameters

memberName

Name of the data member whose value is to be set.

value

Variable containing the desired value.

Returns

`oocSuccess` if successful, or `oocError` if the data-member name is not valid.

setUInt16

Sets the value of the specified `uint16` data member in a converted object.

```
ooStatus setUInt16(
     const char *memberName,
     const uint16 value);
```

Parameters

memberName

Name of the data member whose value is to be set.

value

Variable containing the desired value.

Returns

`oocSuccess` if successful, or `oocError` if the data-member name is not valid.

setUInt32

Sets the value of the specified `uint32` data member in a converted object.

```
ooStatus setUInt32(
     const char *memberName,
     const uint32 value);
```

Parameters

memberName

Name of the data member whose value is to be set.

value

Variable containing the desired value.

Returns

`oocSuccess` if successful, or `oocError` if the data-member name is not valid.

setUInt64

Sets the value of the specified `uint64` data member in a converted object.

```
ooStatus setUInt64(
     const char *memberName,
     const uint64 value);
```

Parameters

memberName

Name of the data member whose value is to be set.

value

Variable containing the desired value.

Returns

`oocSuccess` if successful, or `oocError` if the data-member name is not valid.

ooDBObj Class

Inheritance: **ooObj->ooDBObj**

Handle Class: `ooHandle(ooDBObj)`

Object-Reference Class: `ooRef(ooDBObj)`

The persistence-capable class `ooDBObj` represents an Objectivity / DB *database*. You use `ooDBObj` only for database creation; you work with an existing database through a handle or object reference.

(*DRO*) The `ooDBObj` class also represents a *database image*, which you can create if you have bought and installed both Objectivity / DB Data Replication Option (Objectivity / DRO) and Objectivity / DB Fault Tolerant Option (Objectivity / FTO). You use `ooDBObj` to create an initial database image; additional images are created through a handle or object reference to an existing database.

See:

- "Reference Index" on page 235 for a list of `ooDBObj` member functions

For operations performed through a database handle or object reference, see:

- "Reference Summary" on page 541

About Databases and Database Images

A database is the second highest level in the Objectivity / DB storage hierarchy; a federated database contains one or more databases and every database contains one or more containers.

Physically, a database is maintained in a *database file*, which contains the database and all the persistent data stored in it. Each database is attached to exactly one federated database and is listed in the federated database's catalog. Database

files may reside on different machines than the system-database file for the federated database to which they are attached.

In addition to having a physical filename, each database has a system name, which is its logical name within the federated database. The system name of each database must be unique among all the system names of databases and autonomous partitions in the federated database. A database also has a unique integer identifier within its federated database. When a database is created, you *must* specify a system name for it; you may additionally specify an identifier or allow it to be assigned by Objectivity / DB.

Every database contains a *default container* that is created automatically; default containers are created as instances of class `ooDefaultContObj`.

(*DRO*) In a partitioned federated database, you can use Objectivity / DRO to create and manage multiple *images* of a database to replicate its data across multiple autonomous partitions. Each image of a database is a separate database file that contains all the data in the 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 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; each image is distinguished by the autonomous partition that controls it. 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).

See:

- Chapter 8, “Storage Objects,” of the Objectivity / C++ programmer’s guide for additional information about databases.
- Chapter 28, “Database Images,” of the Objectivity / C++ programmer’s guide for additional information about database images.

Working With Databases and Database Images

Databases are normally created and deleted from within an application, although these tasks can also be performed with administration tools. An application:

- Creates a database using the `ooDBObj` class constructor and `operator new`. The new database is represented in memory as an instance of `ooDBObj`, which serves as a proxy for the actual database file on disk.

- Deletes a database using the global function `ooDelete`; `operator delete` is not available on class `ooDBObj`.

NOTE (*DRO*) You use `ooDBObj` only to create an initial database; you create and work with additional images through a database handle or object reference.

To work with a new database, an application must assign the result of `operator new` to a database handle or object reference (an instance of `ooHandle(ooDBObj)` or `ooRef(ooDBObj)`). Similarly, to identify and work with an existing database, the application must open it through a database handle or object reference; multiple handles and object references can be set to reference the same database. The application then operates on the referenced database by:

- Calling various member functions on any of the referencing handles or object references.
- Passing one of the referencing handles or object references to various global functions or member functions of other classes.

Member functions for operating on existing databases are defined in the *`ooRefHandle`*`(ooDBObj)` classes, not in the `ooDBObj` class itself. Although `ooDBObj` inherits member functions from `ooObj`, these generally do not apply to databases; in fact, the inherited member functions cannot be called, because there is no indirect member-access operator (`->`) on the *`ooRefHandle`*`(ooDBObj)` classes.

You may not derive classes from `ooDBObj`.

Reference Index

Constructors

ooDBObj

Constructs a new database with the specified system name, database file, image weight, and default container parameters.

```
ooDBObj(
     const char *dbSysName,
     const uint32 defContInitPages = 0,
     const uint32 defContGrowth = 0,
     const char *hostName = 0,
     const char *pathName = 0,
     uint32 weight = 1,
     uint16 userDBID = 0);
```

Parameters

dbSysName

System name of the database to create. The specified name:

- Must follow the same naming rules as files of your operating system.
- Must be unique among all the system names of databases and autonomous partitions in the federated database.

The specified name is also used as the database filename if one is not explicitly specified as part of *pathname*.

defContInitPages

Initial number of logical pages to allocate for the default container in the new database. The maximum value is 65535. If you omit this parameter or specify 0, the system default value (4) is used.

defContGrowth

Amount by which the default container may grow, expressed as a percentage of its current size. If you omit this parameter or specify 0, the system default value (10%) is used.

hostName

Name of the data server host on which to create the database file. If you specify a *hostName*, you must also specify the *pathName* parameter. If you omit the *hostName* parameter or specify 0, the database file is created on the same host as the federated database's system database file.

pathName

Fully qualified pathname of the database file on *hostName*. If you specify a *pathName*, you must also specify the *hostName* parameter. If you omit the *pathName* parameter or specify 0, the database file is created in the same directory as the federated database's system database file.

The *pathName* may optionally include a filename for the database file. If you omit a filename, the system name is used.

weight

(*DRO*) Weight of the first database image. *weight* must be an integer greater than zero. If you omit this parameter, the weight is 1.

userDBID

Application-assigned database identifier, expressed as a single integer. If you omit this parameter or specify 0, the identifier is assigned by Objectivity/DB.

See also `ooReplace` global macro

Operators

operator new

Creates a new database in the currently open federated database.

```
1.   void *operator new(
       size_t,
       const ooRefHandle(ooFDObj) &containingFD);
2.   void *operator new(
       size_t,
       const ooRefHandle(ooFDObj) &containingFD,
       const ooRefHandle(ooAPObj) &containingAP);
```

Parameters *size_t*

Do not specify; this parameter is automatically initialized by the compiler with the size of the class type in bytes.

containingFD

Federated database in which to create the new database. A database can be created only in the currently open federated database. As a consequence, you can omit this parameter in variant 1. In either variant, the specified parameter must be an object reference or handle to the open federated database.

containingAP

(*FTO*) Autonomous partition in which to create the new database. You can specify an object reference or handle to the desired autonomous partition. If you omit this parameter, the database is created in the boot autonomous partition.

Returns

Memory pointer to the new database. This pointer is null if an error occurs during the creation of the database or if a database with the specified system name already exists.

Discussion

This operator must be used in an update transaction. When you commit or checkpoint the transaction, the new database is made permanent on disk. If the transaction is aborted, the database is not created.

Expressions containing this operator are of the following form:

`new(`*`locatingDirectives`*`) ooDBObj(`*`initializers`*`)`

The *`locatingDirectives`* are the *`containingFD`* and optional *`containingAP`* parameters.

The *`initializers`* are a list of values to be passed as parameters to the `ooDBObj` constructor. At a minimum, you must specify the system name of the new database. If you omit the host and pathname, the database file is located in the directory containing the system database file of the specified federated database or autonomous partition.

You normally assign the result of `operator new` directly to a database handle. You can verify the creation of the database by checking whether the handle is null.

ooDefaultContObj Class

Inheritance: **ooObj->ooContObj->ooDefaultContObj**

Handle Class: `ooHandle(ooDefaultContObj)`

Object-Reference Class: `ooRef(ooDefaultContObj)`

The persistence-capable class `ooDefaultContObj` represents the *default container* in a database.

See:

- "Reference Summary" on page 240 for an overview of member functions

About Default Containers

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-defined container.
- Scope names of 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`.

Working With Default Containers

A default container is created automatically for each database when the database is created. You should not create additional default containers in a database, nor shoud you delete a database's default container. You specify a default container's

initial size and growth factor when you create the database. An application should not create derived classes from `ooDefaultContObj`.

Like other storage objects, default containers are manipulated through handles or object references. The handle and object-reference classes `ooHandle(ooDefaultContObj)` and `ooRef(ooDefaultContObj)` are derived from `ooHandle(ooContObj)` and `ooRef(ooContObj)`, respectively. You can find a default container in a database using the default container's system name.

Default containers are found by operations that initialize an object iterator of class `ooItr(ooContObj)` or `ooItr(ooObj)`. When advancing such an iterator, you should test for the default container before performing any operation that deletes or moves the found container. For example, when the iterator references the next found container, you could call the container's inherited `ooIsKindOf` member function.

Reference Summary

In the following table, the member functions indicated as *(inherited)* are defined by `ooObj` and are available for default containers; they are documented with the `ooObj` class (page 431) and are the *only* `ooObj` member functions that apply to default containers.

Working With the Container	`ooGetTypeN` *(inherited)* `ooGetTypeName` *(inherited)* `ooIsKindOf` *(inherited)* `ooUpdate` *(inherited)* `ooValidate` *(inherited)*

ooEqualLookupField Class

Inheritance: **ooLookupFieldBase->ooEqualLookupField**

The non-persistence-capable indexing class `ooEqualLookupField` represents a *lookup field* that tests whether the value of an indexed object's key field is equal to (=) a specified comparison value.

See:

- "Reference Index" on page 242 for a list of member functions

Applications that use indexes must include the `ooIndex.h` header file.

About Equal-To Lookup Fields

A lookup field is part of a *lookup key* (an instance of the `ooLookupKey` class that is used for looking up persistent objects in an index). A lookup key consists of one or more lookup fields, each representing a condition that a found object must satisfy. Lookup fields are instances of the concrete classes derived from the abstract base class `ooLookupFieldBase`.

The concrete class `ooEqualLookupField` represents a condition that tests whether the values of a particular key field are equal to a particular comparison value. You use the class constructor to specify the key field and the value.

For complete information about using lookup fields in lookup keys, see "About Lookup Keys" on page 399

Reference Index

ooEqualLookupField	Constructs a new lookup field for testing whether values of the specified key field are equal to the specified comparison value.

Constructors

ooEqualLookupField

Constructs a new lookup field for testing whether values of the specified key field are equal to the specified comparison value.

```
1.  ooEqualLookupField(
      const ooKeyField &field,
      const void *valuePtr);
2.  ooEqualLookupField(
      const ooTypeNumber typeN,
      const char *memberName,
      const void *valuePtr);
```

Parameters

field

Key-field object representing a key field of the indexed class. The values of the specified key field will be tested by this lookup field.

typeN

Type number of the indexed class.

memberName

Name of a data member that serves as a key field of the indexed class. The values of the specified key field will be tested by this lookup field.

The specified data member must be defined or inherited by the class specified by *typeN*. You can qualify the name of an inherited data member using the following notation (where *baseClassName* is the name of the base class that defines the inherited data member):

baseClassName::*dataMemberName*

You *must* qualify the name of an inherited data member if the member name is ambiguous (for example, the same name is defined in both the base class and the class to be indexed) or if the member name is not visible due to access control.

valuePtr

Data value to which key-field values are to be compared. The type of this value must match the key-field type (the type of the data member you specified to the key-field object). Using data of any other type may have unpredictable results.

Discussion

This lookup field causes the values of the specified key field to be tested when an iterator scans the index; an indexed object is found if its tested value is equal to the value specified by *valuePtr*.

This lookup field will be ignored if *field* does not specify a key field of the index being searched.

ooFDObj Class

Inheritance: **ooObj->ooFDObj**

Handle Class: ooHandle(ooFDObj)

Object-Reference Class: ooRef(ooFDObj)
d_Database

The persistence-capable `ooFDObj` class represents an Objectivity / DB *federated database*. The `ooFDObj` class is used internally by Objectivity / DB; you work with a federated database through a handle or object reference.

See:

- "Reference Summary" on page 575 for operations performed through a federated-database handle or object reference

(*ODMG*) An Objectivity / DB federated database corresponds to an ODMG database (see class `d_Database`).

About Federated Databases

A federated database is the highest level in the Objectivity / DB storage hierarchy; each federated database logically contains one or more databases.

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 any 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 attributes. The simple name of the boot file serves as the federated database's system name.

Working With Federated Databases

A federated database can be created and deleted only with administration tools. An application obtains a single instance of `ooFDObj` by opening the federated database through a federated-database handle or object reference (an instance of `ooHandle(ooFDObj)` or `ooRef(ooFDObj)`). The obtained `ooFDObj` instance serves as a proxy for the actual system-database file on disk.

The application operates on the federated database by:

- Calling various member functions on the federated-database handle or object reference.
- Passing the federated-database handle or object reference to various global functions or member functions of other classes.

Member functions for operating on federated databases are defined in the *`ooRefHandle`*`(ooFDObj)` classes, not in the `ooFDObj` class itself. Although `ooFDObj` inherits member functions from `ooObj`, these generally do not apply to federated databases; in fact, the inherited member functions cannot be called, because there is no indirect member-access operator (`->`) on the *`ooRefHandle`*`(ooFDObj)` classes.

You may not derive classes from `ooFDObj`.

See the Objectivity / DB administration book for information about federated database creation, files, and attributes.

ooGCContObj Class

Inheritance: **ooObj->ooContObj->ooGCContObj**

Handle Class: `ooHandle(ooGCContObj)`

Object-Reference Class: `ooRef(ooGCContObj)`

The persistence-capable class `ooGCContObj` represents a *garbage-collectible container*, and serves as the base class for application-defined garbage-collectible container classes. The `ooGCContObj` class and its corresponding handle and object-reference classes together define the behavior of garbage-collectible containers.

See:

- "Reference Summary" on page 248 for an overview of member functions
- "Reference Index" on page 249 for a list of member functions

About Garbage-Collectible Containers

As the name implies, garbage-collectible containers adhere to a garbage-collection paradigm analogous to the memory of languages such as Java and Smalltalk. These languages consider an object to be "garbage" if it is not referenced by any other object, and periodically clean up memory by deleting garbage objects. In an Objectivity/DB garbage-collectible container, a persistent object becomes "garbage" if it is no longer linked to a named root either directly or indirectly through associations, reference attributes, or membership in persistent collections. However, "garbage" persistent objects are not deleted automatically; you must use the `oogc` administration tool when you want to delete invalid objects from the garbage-collectible containers of a federated database.

Because Objectivity/C++ does not directly support named roots, Objectivity/C++ applications generally use standard containers (instances of `ooContObj`) instead of garbage-collectible containers. In a standard container, each unwanted object must be deleted explicitly (for example, using the `ooDelete` global function). However an Objectivity/C++ application can choose to create garbage-collectible containers if it interoperates with Objectivity for Java or Objectivity/Smalltalk applications.

NOTE The `ooGCContObj` class has a predefined derived class called `ooGCRootsCont`. Do not use this derived class in any way—for example, do not instanatiate it or or derive any classes from it.

Working With Garbage-Collectible Containers

You can work with instances of `ooGCContObj` or you can work with instances of application-defined classes derived from `ooGCContObj`.

As is the case for any container, you specify whether a garbage-collectible container is to be transient or persistent when you create it; garbage-collectible containers should be persistent. You create a garbage-collectible container with a call to `operator new`; the clustering directive in that call specifies where in the federated database to store the new container.

Like other persistent objects, garbage-collectible containers are normally manipulated through handles or object references. You can store and find a garbage-collectible container in the database just as you would any other persistent object.

Reference Summary

In the following table, the member functions indicated as *(inherited)* are defined by `ooObj` and are available for containers; they are documented with the `ooObj` class (page 431) and are the *only* `ooObj` member functions that apply to containers. `operator new` is documented with `ooContObj` (page 207).

Creating and Deleting a Garbage-Collectible Container	ooGCContObj operator new *(inherited from ooContObj)* operator delete *(inherited)*
Working With a Garbage-Collectible Container	ooGetTypeN *(inherited)* ooGetTypeName *(inherited)* ooIsKindOf *(inherited)* ooThis ooUpdate *(inherited)* ooValidate *(inherited)*
ODMG Interface	operator new

Reference Index

ooGCContObj	Default constructor that constructs a new garbage-collectible container.
ooThis	Sets an object reference or handle to reference this garbage-collectible container.

Constructors

ooGCContObj

Default constructor that constructs a new garbage-collectible container.

```
ooGCContObj();
```

Member Functions

ooThis

Sets an object reference or handle to reference this garbage-collectible container.

```
1.   ooHandle(ooGCContObj) ooThis() const;
2.   ooRef(ooGCContObj) &ooThis(
       ooRef(ooGCContObj) &container) const;
```

```
3.   ooHandle(ooGCContObj) &ooThis(
       ooHandle(ooGCContObj) &container) const;
```

Parameters

container

Object reference or handle to be set to this garbage-collectible container.

Returns

Object reference or handle to this garbage-collectible container.

Discussion

You normally use `ooThis` in a member function of an application-defined container class; when such a member function is called on a container of the class, `ooThis` provides the member function with an object reference or handle to the container. The member function can then perform operations on the container that are available only through an object reference or handle.

When called without a *container* parameter, this member function allocates a new handle and returns it. Otherwise, this member function returns the object reference or handle that is passed to it.

ooGeneObj Class

Inheritance: **ooObj->ooGeneObj**

Handle Class: `ooHandle(ooGeneObj)`

Object-Reference Class: `ooRef(ooGeneObj)`

The persistence-capable class `ooGeneObj` represents a *genealogy*. The `ooGeneObj` class and its corresponding handle and object-reference classes together define the behavior of genealogies.

See:

- "Reference Summary" on page 253 for an overview of member functions
- "Reference Index" on page 253 for a list of member functions

About Genealogies

A genealogy manages a set of next and previous versions of a particular basic object, allowing you to appoint one of these versions as the default version. The semantics of the default version are application-specific—some applications may successively appoint each new version as 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 maintains:

- A bidirectional to-many association to each of the versions it manages.
- A bidirectional one-to-one association to its default version.

```
class ooGeneObj : public ooObj {
   // Links a genealogy to the versions in it.
   ooRef(ooObj) allVers[ ] <-> geneObj;
```

```
    // Links a genealogy to the default version.
    ooRef(ooObj) defaultVers <-> defaultToGeneObj;
...
}
```

Because of these associations, you can find the default version from any other version in the genealogy. Furthermore, you can easily find all of the versions in the genealogy by iterating over the destination objects of the `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.

A genealogy may, but need not, track an entire set of next and previous versions. After the first default version is appointed, all subsequent versions created from it are automatically linked to the genealogy. If you appoint an object to be the default version before you create any versions from it, then all of the object's versions will be linked to the genealogy. If, however, you create a few versions and then appoint a default, the pre-existing versions are not automatically added to the genealogy; you can add them explicitly by setting them as destination objects of the genealogy's `allVers` association.

Working With Genealogies

Genealogies can be created either implicitly or explicitly. An instance of `ooGeneObj` is created implicitly the first time you appoint a default version by calling the `setDefaultVers` member function on a handle to a basic object. Alternatively, you can create a genealogy explicitly as you would any other basic object and then appoint its default version by calling the genealogy's `set_defaultVers` member function.

As is the case for any basic object, you specify whether a genealogy is to be transient or persistent when you create it; genealogies *must be persistent*. You create a genealogy with a call to the `new` operator; the clustering directive in that call specifies where in the federated database to store the new genealogy.

You may create and work with instances of the `ooGeneObj` class. To support more complex versioning semantics, an application can define its own custom genealogy class derived from `ooGeneObj`, and then work with instances of the derived class.

Like other persistent objects, genealogies are normally manipulated through handles or object references. You can store and find a genealogy in the database just as you would any other persistent object.

Reference Summary

In the following table, operators indicated as *(inherited)* are overloaded in this class with no change in behavior; they are documented with the ooObj class (page 431), along with the other inherited member functions not listed here.

Creating and Deleting a Genealogy	ooGeneObj operator new *(inherited)* operator delete *(inherited)*
Referencing This Genealogy	ooThis
Managing the Default Version	defaultVers del_defaultVers exist_defaultVers set_defaultVers
Managing All Versions	add_allVers allVers exist_allVers sub_allVers

Reference Index

exist_defaultVers	Tests whether this genealogy has a default version; if a parameter is given, tests whether the specified object is the default version in this genealogy.
ooGeneObj	Default constructor that constructs a new genealogy with no associated versions.
ooThis	Gets an object reference or handle to this genealogy.
set_defaultVers	Links this genealogy to the specified object, setting the specified object as the default version in the genealogy.
sub_allVers	Removes the specified version from this genealogy by deleting the `allVers` association link from this genealogy to the specified version.

Constructors and Destructors

ooGeneObj

Default constructor that constructs a new genealogy with no associated versions.

```
ooGeneObj();
```

Member Functions

allVers

Initializes an object iterator to find, and optionally open, all versions in this genealogy that satisfy any specified selection criteria.

```
1.  ooStatus allVers(
      ooItr(ooObj) &iterator,
      const ooMode openMode = oocNoOpen) const;
2.  ooStatus allVers(
      ooItr(ooObj) &iterator,
      const char *predicate) const;
3.  ooStatus allVers(
      ooItr(ooObj) &iterator,
      const ooMode openMode,
      const ooAccessMode access,
      const char *predicate) const;
```

Parameters *iterator*

Object iterator for finding all versions associated with this genealogy.

openMode

Intended level of access to each version found by the iterator's `next` member function:

- `oocNoOpen` (the default in variant 1) causes `next` to set the iterator to the next found version without opening it.
- `oocRead` causes `next` to open the next found version for read.
- `oocUpdate` causes `next` to open the next found version for update.

  Warning: If versioning is enabled for one or more found objects, specifying `oocUpdate` means that `next` will create a new version of each such object.

predicate

String expression in predicate query language; specifies the condition to be met by the found versions. The iterator is initialized only with versions that match *predicate*.

access

Level of access control of the data members that *predicate* can test:

- Specify `oocPublic` to permit the predicate to test only public data members, preserving encapsulation.
- Specify `oocAll` to permit the predicate to test any data member. To preserve encapsulation, you should use this mode only within member functions of the class you are querying.

Returns `oocSuccess` if successful; otherwise `oocError`.

See also `add_allVers`
`sub_allVers`
`del_allVers`

add_allVers

Links this genealogy with the specified version, adding the version to this genealogy.

```
ooStatus add_allVers(const ooHandle(ooObj) &object);
```

Parameters *object*

Handle to a basic object.

Returns `oocSuccess` if successful; otherwise `oocError`.

Discussion The operation links this genealogy to the specified version by:

- Creating an `allVers` association link in this genealogy.
- Creating the inverse `geneObj` association link in the specified version.

The application must be able to obtain update locks on both objects.

See also `allVers`
`sub_allVers`
`del_allVers`

defaultVers

Finds, and optionally opens, the default version in this genealogy.

```
1.  ooHandle(ooObj) defaultVers(
      const ooMode openmode = oocNoOpen) const;
2.  ooRef(ooObj) &defaultVers(
      ooRef(ooObj) &defaultVersion,
      const ooMode openmode = oocNoOpen) const;
3.  ooHandle(ooObj) &defaultVers(
      ooHandle(ooObj) &defaultVersion,
      const ooMode openmode = oocNoOpen) const;
```

Parameters *defaultVersion*

Object reference or handle to set to the default version.

openmode

Intended level of access to the default version:

- Specify `oocNoOpen` (the default) to set the returned object reference or handle to the object without opening it.
- Specify `oocRead` to open the object for read.
- Specify `oocUpdate` to open the object for update.

Returns Object reference or handle to the default version. A null object reference or handle is returned if this genealogy has no default version.

Discussion When called without an object reference or handle argument, this member function allocates a new handle and returns it. Otherwise, this member function returns the result in the object reference or handle that is passed to it.

See also `set_defaultVers`
`del_defaultVers`

del_allVers

Removes all versions from this genealogy by deleting all of its `allVers` associations.

```
ooStatus del_allVers();
```

Returns `oocSuccess` if successful; otherwise `oocError`.

Discussion This operation also deletes the inverse `geneObj` association link from each of the former versions. The application must be able to obtain update locks on all of the affected objects.

You should call the `exist_allVers` member function to test whether any associations exist before you try to delete them.

See also `allVers`
`add_allVers`
`sub_allVers`

del_defaultVers

Deletes any `defaultVers` association that exists for this genealogy, leaving this genealogy without a default version.

```
ooStatus del_defaultVers();
```

Returns `oocSuccess` if successful; otherwise `oocError`.

Discussion This operation also deletes the inverse `defaultToGeneObj` association link from the former default version. The application must be able to obtain update locks on both objects.

You typically call this function to remove a genealogy's previous default version before setting a new one. You shouldn't leave a genealogy without a default version.

You should call the `exist_defaultVers` member function to test whether an association exists before you try to delete it.

See also `defaultVers`
`set_defaultVers`

exist_allVers

Tests whether this genealogy has any versions; if a parameter is given, tests whether the specified object is a version in this genealogy.

```
1.  ooBoolean exist_allVers() const;
2.  ooBoolean exist_allVers(
      const ooHandle(ooObj) &object) const;
```

Parameters *object*

Handle to the object to be tested as a version.

Returns (Variant 1) `oocTrue` if this genealogy has any versions, otherwise `oocFalse`. (Variant 2) `oocTrue` if the specified object is a version in this genealogy, otherwise `oocFalse`.

exist_defaultVers

Tests whether this genealogy has a default version; if a parameter is given, tests whether the specified object is the default version in this genealogy.

```
1.  ooBoolean exist_defaultVers() const;
2.  ooBoolean exist_defaultVers(
      const ooHandle(ooObj) &object) const;
```

Parameters *object*

Handle to the object to be tested as the default version.

Returns (Variant 1) `oocTrue` if this genealogy has a default version, otherwise `oocFalse`. (Variant 2) `oocTrue` if the specified object is the default version in this genealogy, otherwise `oocFalse`.

ooThis

Gets an object reference or handle to this genealogy.

```
1.  ooHandle(ooGeneObj) ooThis() const;
2.  ooRef(ooGeneObj) &ooThis(
      ooRef(ooGeneObj) &genealogy) const;
3.  ooHandle(ooGeneObj) &ooThis(
      ooHandle(ooGeneObj) &genealogy) const;
```

Parameters *genealogy*

Object reference or handle to be set to this genealogy.

Returns Object reference or handle to this genealogy.

Discussion You normally use `ooThis` in a member function of a custom genealogy class. When such a member function is called on a genealogy of the class, `ooThis` provides the member function with an object reference or handle to the genealogy. The member function can then perform operations on the genealogy that are available only through an object reference or handle. (**Note:** You actually use the type-specific version of `ooThis` that is generated for your custom genealogy class by the DDL processor.)

When called without a *`genealogy`* parameter, `ooThis` allocates a new handle and returns it. Otherwise, `ooThis` returns the object reference or handle that is passed to it.

set_defaultVers

Links this genealogy to the specified object, setting the specified object as the default version in the genealogy.

```
ooStatus set_defaultVers(const ooHandle(ooObj) &object);
```

Parameters *`object`*

Handle to a basic object.

Returns `oocSuccess` if successful; otherwise `oocError`.

Discussion All subsequently created versions of the specified object are automatically added to this genealogy.

The operation links this genealogy with the specified object by:

- Creating a `defaultVers` association link in this genealogy.
- Creating the inverse `defaultToGeneObj` association link in the specified object.

The application must be able to obtain update locks on both objects.

Because `defaultVers` is a to-one association, an error is signaled if this object already has a link for the association. That is, you must remove any existing default version from the genealogy before you set a new default version.

See also `defaultVers`
`del_defaultVers`

sub_allVers

Removes the specified version from this genealogy by deleting the `allVers` association link from this genealogy to the specified version.

```
ooStatus sub_allVers(const ooHandle(ooObj) &object);
```

Parameters

object

Handle to the version to be removed.

Returns

`oocSuccess` if successful; otherwise `oocError`.

See also

`add_allVers`
`allVers`
`del_allVers`

ooGreaterThanEqualLookupField Class

Inheritance: **ooLookupFieldBase->ooGreaterThanEqualLookupField**

The non-persistence-capable indexing class `ooGreaterThanEqualLookupField` represents a *lookup field* that tests whether the value of an indexed object's key field is greater than or equal to (`>=`) a specified comparison value.

See:

- "Reference Index" on page 262 for a list of member functions

Applications that use indexes must include the `ooIndex.h` header file.

About Greater-Than-Or-Equal-To Lookup Fields

A lookup field is part of a *lookup key* (an instance of the `ooLookupKey` class that is used for looking up persistent objects in an index). A lookup key consists of one or more lookup fields, each representing a condition that a found object must satisfy. Lookup fields are instances of the concrete classes derived from the abstract base class `ooLookupFieldBase`.

The concrete class `ooGreaterThanEqualLookupField` represents a condition that tests whether the values of a particular key field are greater than or equal to a particular comparison value. You use the class constructor to specify the key field and the value.

For complete information about using lookup fields in lookup keys, see "About Lookup Keys" on page 399.

Reference Index

ooGreaterThanEqualLookupField	Constructs a new lookup field for testing whether values of the specified key field are greater than or equal to the specified comparison value.

Constructors

ooGreaterThanEqualLookupField

Constructs a new lookup field for testing whether values of the specified key field are greater than or equal to the specified comparison value.

```
1.  ooGreaterThanEqualLookupField(
      const ooKeyField &field,
      const void *valuePtr);
2.  ooGreaterThanEqualLookupField(
      const ooTypeNumber typeN,
      const char *memberName,
      const void *valuePtr);
```

Parameters

field

Key-field object representing a key field of the indexed class. The values of the specified key field will be tested by this lookup field.

typeN

Type number of the indexed class.

memberName

Name of a data member that serves as a key field of the indexed class. The values of the specified key field will be tested by this lookup field.

The specified data member must be defined or inherited by the class specified by *typeN*. You can qualify the name of an inherited data member using the following notation (where *baseClassName* is the name of the base class that defines the inherited data member):

baseClassName::*dataMemberName*

You *must* qualify the name of an inherited data member if the member name is ambiguous (for example, the same name is defined in both the base class and the class to be indexed) or if the member name is not visible due to access control.

valuePtr

Data value to which key-field values are to be compared. The type of this value must match the key-field type (the type of the data member you specified to the key-field object). Using data of any other type may have unpredictable results.

Discussion

This lookup field causes the values of the specified key field to be tested when an iterator scans the index; an indexed object is found if its tested value is greater than or equal to the value specified by *valuePtr*.

This lookup field will be ignored if *field* does not specify a key field of the index being searched.

ooGreaterThanLookupField Class

Inheritance: **ooLookupFieldBase->ooGreaterThanLookupField**

The non-persistence-capable indexing class `ooGreaterThanLookupField` represents a *lookup field* that tests whether the value of an indexed object's key field is greater than (>) the specified comparison value.

See:

- "Reference Index" on page 266 for a list of member functions

Applications that use indexes must include the `ooIndex.h` header file.

About Greater-Than Lookup Fields

A lookup field is part of a *lookup key* (an instance of the `ooLookupKey` class that is used for looking up persistent objects in an index). A lookup key consists of one or more lookup fields, each representing a condition that a found object must satisfy. Lookup fields are instances of the concrete classes derived from the abstract base class `ooLookupFieldBase`.

The concrete class `ooGreaterThanLookupField` represents a condition that tests whether the values of a particular key field are greater than a particular comparison value. You use the class constructor to specify the key field and the value.

For complete information about using lookup fields in lookup keys, see "About Lookup Keys" on page 399.

Reference Index

ooGreaterThanLookupField	Constructs a new lookup field for testing whether values of the specified key field are greater than the specified comparison value.

Constructors

ooGreaterThanLookupField

Constructs a new lookup field for testing whether values of the specified key field are greater than the specified comparison value.

```
1.  ooGreaterThanLookupField(
      const ooKeyField &field,
      const void *valuePtr);
2.  ooGreaterThanLookupField(
      const ooTypeNumber typeN,
      const char *memberName,
      const void *valuePtr);
```

Parameters

field

Key-field object representing a key field of the indexed class. The values of the specified key field will be tested by this lookup field.

typeN

Type number of the indexed class.

memberName

Name of a data member that serves as a key field of the indexed class. The values of the specified key field will be tested by this lookup field.

The specified data member must be defined or inherited by the class specified by *typeN*. You can qualify the name of an inherited data member using the following notation (where *baseClassName* is the name of the base class that defines the inherited data member):

baseClassName::*dataMemberName*

You *must* qualify the name of an inherited data member if the member name is ambiguous (for example, the same name is defined in both the base class and the class to be indexed) or if the member name is not visible due to access control.

valuePtr

Data value to which key-field values are to be compared. The type of this value must match the key-field type (the type of the data member you specified to the key-field object). Using data of any other type may have unpredictable results.

Discussion

This lookup field causes the values of the specified key field to be tested when an iterator scans the index; an indexed object is found if its tested value is greater than the value specified by *valuePtr*.

This lookup field will be ignored if *field* does not specify a key field of the index being searched.

ooHashAdmin Class

Inheritance: **ooObj->ooAdmin->ooHashAdmin**

Handle Class: `ooHandle(ooHashAdmin)`

Object-Reference Class: `ooRef(ooHashAdmin)`

The persistence-capable class `ooHashAdmin` represents *hash administrators*.

See:

- "Reference Summary" on page 270 for an overview of member functions
- "Reference Index" on page 270 for a list of member functions

To use this class, your application must include the `ooCollections.h` header file. For UNIX linking information, see *Installation and Platform Notes for UNIX*.

About Hash Administrators

Each unordered collection derived from `ooHashSet` has a hash administrator that manages the containers used by the collection's internal objects, namely the hash buckets of the collection's extendible hash table. An unordered collection's hash administrator is created when the collection itself is created.

A hash administrator has a property that you can set to control when the current hash-bucket container is considered "full." The *maximum buckets per container* property specifies how many hash buckets can be clustered together in the same container. It is typical for a hash bucket to be updated frequently. The default value for this property is 1, which minimizes lock conflicts. If you know that a particular collection will be used by a single user, locking is not an issue. In that case, a larger value may be appropriate for the collection's hash administrator.

For additional information, see "Hash Administrator" on page 253 in the Objectivity/C++ programmer's guide.

Working With a Hash Administrator

Like other persistent objects, hash administrators are normally manipulated through handles or object references.

You call an unordered collection's `admin` member function to obtain an object reference to the collection's hash administrator; you must then cast the returned object reference to type `ooRef(ooHashAdmin)` before you access the hash administrator's data member.

Reference Summary

Getting Information	`bucketContainer` `maxBucketsPerContainer`
Setting Information	`setMaxBucketsPerContainer`

Reference Index

`bucketContainer`	Gets this hash administrator's current hash-bucket container.
`maxBucketsPerContainer`	Gets the maximum number of hash buckets per container for this hash administrator.
`setMaxBucketsPerContainer`	Sets the maximum number of hash buckets per container for this hash administrator.

Member Functions

bucketContainer

Gets this hash administrator's current hash-bucket container.

```
ooRef(ooContObj) bucketContainer();
```

Returns Object reference to this hash administrator's current hash-bucket container.

maxBucketsPerContainer

Gets the maximum number of hash buckets per container for this hash administrator.

```
ooInt32 maxBucketsPerContainer();
```

Returns The maximum number of hash buckets that can be stored in a single container.

setMaxBucketsPerContainer

Sets the maximum number of hash buckets per container for this hash administrator.

```
void setMaxBucketsPerContainer(ooInt32 max);
```

Parameters *max*

The maximum number of hash buckets that can be stored in a single container.

Discussion Changing the maximum buckets per container affects only the collection's current hash-bucket container and any hash-bucket containers created in the future. If you increase the number of hash buckets per container from the default, any existing hash-bucket containers are left containing only one hash bucket each.

ooHashMap Class

Inheritance: **ooObj->ooCollection->ooHashSet->ooHashMap**

Handle Class: `ooHandle(ooHashMap)`

Object-Reference Class: `ooRef(ooHashMap)`

The persistence-capable class `ooHashMap` represents *unordered object maps*.

See:

- "Reference Summary" on page 274 for an overview of member functions
- "Reference Index" on page 275 for a list of member functions

To use this class, your application must include the `ooCollections.h` header file. For UNIX linking information, see *Installation and Platform Notes for UNIX*.

About Unordered Object Maps

An object map is a collection of key-value pairs; each key and each value is a persistent object. No two elements of the object map may have the same key. As the name implies, each element of an object map is a mapping from its key object to its value object.

Unordered object maps are *scalable* collections, that is, they can increase in size with minimal performance degradation. They are implemented using an extendible hashing mechanism so that elements can be added, deleted, and retrieved efficiently. The hash value for each element is computed from its key.

The hash values for keys of an unordered object map are computed by the object map's corresponding comparator. If an unordered object map has a default comparator, the hash values are computed from the object identifiers (OIDs) of the keys.

For additional information, see Chapter 11, “Persistent Collections,” in the Objectivity/C++ programmer’s guide.

Working With an Unordered Object Map

As is the case for any basic object, you specify whether an unordered object map is to be transient or persistent when you create it; unordered object maps *must be persistent*. You create an unordered object map with a call to the `new` operator; the clustering directive in that call specifies where in the federated database to store the new unordered object map.

Like other persistent objects, unordered object maps are normally manipulated through handles or object references. You can store and find an unordered object map in the database just as you would any other persistent object.

Related Classes

Two additional classes represent persistent collections of key-value pairs:

- `ooTreeMap` represents a *sorted* object map. It is implemented using a B-tree data structure.
- `ooMap` represents an unordered name map, that is, a collection of key-value pairs in which *the key is a string* and the value is an object reference to a persistent object. It uses a traditional (non-extendible) hashing mechanism.

Reference Summary

In the following table:

- Operators indicated as *(inherited)* are overloaded in this class with no change in behavior; they are documented with the `ooObj` class (page 431), along with the other inherited member functions not listed here.
- Member function indicated as *(inherited)* are inherited from the `ooHashSet` class (page 283) or the `ooCollection` class (page 173) and are documented with the defining class.

Creating and Deleting	ooHashMap operator new *(inherited)* operator delete *(inherited)*
Adding, Removing, and Changing Elements	add addAll clear *(inherited)* put remove removeAll *(inherited)* removeAllDeleted retainAll *(inherited)*
Getting Elements	get keyIterator valueIterator
Getting Information	hashOf *(inherited)* size *(inherited)*
Finding Auxiliary Objects	admin *(inherited)* comparator *(inherited)*
Testing	containsKey containsValue containsAll *(inherited)* isEmpty *(inherited)*
Viewing in an MROW Transaction	refresh *(inherited)*

Reference Index

add	Adds the specified object to this unordered object map.
addAll	Adds all elements in the specified collection to this unordered object map.
containsKey	Tests whether this unordered object map contains an element with the specified key.
containsValue	Tests whether this unordered object map contains an element with the specified value.
get	Finds the value paired with the specified key in this unordered object map.

keyIterator	Initializes a scalable-collection iterator to find all the keys of this unordered object map.
ooHashMap	Constructs a new empty unordered object map.
put	Maps the specified key to the specified value in this unordered object map.
remove	Removes the element, if any, with the specified key from this unordered object map.
removeAllDeleted	Removes from this unordered object map all elements in which either the key or the value has been deleted from the federated database.
valueIterator	Initializes a scalable-collection iterator to find all values in this unordered object map.

Constructors

ooHashMap

Constructs a new empty unordered object map.

```
1.   ooHashMap(
       int bucketSize = 30011,
       int initialBuckets = 1,
       ooHandle(ooContObj) contAdminH = 0,
       ooHandle(ooContObj) contBucketH = 0);
2.   ooHashMap(
       ooHandle(ooCompare) &compH,
       int bucketSize = 30011,
       int initialBuckets = 1,
       ooHandle(ooContObj) contAdminH = 0,
       ooHandle(ooContObj) contBucketH = 0);
```

Parameters

bucketSize

The size of a hash bucket in the new unordered object map's hash table. The size of a hash bucket is the number of elements that can be hashed into each bucket. For optimal performance, the bucket size should be a prime number. If you specify a bucket size that is not a prime number, the next higher prime number is computed and used as the actual bucket size.

initialBuckets

The minimum number of initial hash buckets to create for the new unordered object map. The actual number of hash buckets created is the smallest power of 2 that is greater than *initialBuckets*. For example, if *initialBuckets* is 10, the number of hash buckets is 2**4, or 16.

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 object map with a large number of initial hash buckets requires more disk space, more memory for the directory, and more time to create.

By default, each initial hash bucket is stored in its own newly created container. If you specify a container in the *contBucketH* parameter, however, all initial hash buckets are instead created in that container. In that case, you reduce the storage overhead for the object map, but you also reduce concurrent access.

contAdminH

Handle to the container in which to store the hash administrator for the new unordered object map. The default value (0) creates a container for the hash administrator in the same database as the unordered object map itself.

contBucketH

Handle to the container in which to store the initial hash bucket(s) for the new unordered object map. The default value (0) creates a separate container for each initial hash bucket, all in the same database as the unordered object map itself.

compH

Handle to the comparator for the new unordered object map; must be an instance of an application-specific derived class of `ooCompare`.

Discussion

Variant 1 creates an empty unordered object map with a default comparator and the specified bucket size.

Variant 2 creates an empty unordered object map with the specified comparator and the specified bucket size.

Member Functions

add

Adds the specified object to this unordered object map.

```
virtual ooBoolean add(
     const ooHandle(ooObj) &objH);
```

Parameters

objH

Handle to the object to be added.

Returns

`oocTrue` if an element was added; otherwise, `oocFalse`.

Discussion

This member function adds a new key-value pair to this unordered object map with the specified object as its key and a null value. If this unordered object map currently has an element whose key is the specified object, the value of the existing element is replaced by null.

See also

`addAll`
`remove`

addAll

Adds all elements in the specified collection to this unordered object map.

```
virtual ooBoolean addAll(
     const ooHandle(ooCollection) &collectionH);
```

Parameters

collectionH

Handle to the unordered or sorted object map whose elements are to be added to this unordered object map.

Returns

`oocTrue` if any elements were added; otherwise, `oocFalse`.

Discussion

If the specified collection is an object map, its elements are added to this unordered object map. If this unordered object map currently has an element with the same key as an element of the specified collection, the existing element is replaced by the element of the specified collection.

If the specified collection is a collection of persistent objects, each of its elements is added as a key to this unordered object map; a null value is paired with each key. If this unordered object map currently has an element whose key is an

element of the specified collection, the value of the existing element is replaced by null.

See also add

containsKey

Tests whether this unordered object map contains an element with the specified key.

```
1.   ooBoolean containsKey(
       const ooHandle(ooObj) &keyH) const;
2.   ooBoolean containsKey(
       const void *lookupVal) const;
```

Parameters *keyH*

Handle to the key to be tested for containment in this unordered object map.

lookupVal

Pointer to data that identifies the key to be tested for containment in this unordered object map.

Returns `oocTrue` if this unordered object map contains an element with the specified key; otherwise, `oocFalse`.

Discussion You can call this member function to check whether this unordered object map maps the specified key to some value.

Variant 2 tests whether any key is “equal” to the specified lookup data, as determined by the comparator for this unordered object map. It is useful if this unordered object map has an application-defined comparator that can identify a key based on class-specific data.

See also `containsValue`

containsValue

Tests whether this unordered object map contains an element with the specified value.

```
ooBoolean containsValue(
     const ooHandle(ooObj) &valueH) const;
```

Parameters *valueH*

Handle to the value to be tested for containment in this unordered object map.

Returns `oocTrue` if this unordered object map contains an element whose value is the specified object; otherwise, `oocFalse`.

Discussion You can call this member function to check whether this unordered object map maps at least one key to the specified value.

See also `containsKey`

get

Finds the value paired with the specified key in this unordered object map.

```
1.  ooRef(ooObj) get(const ooHandle(ooObj) &keyH) const;
2.  ooRef(ooObj) get(const void *lookupVal) const;
```

Parameters *keyH*

Handle to the key to be looked up.

lookupVal

Pointer to data that identifies the desired key.

Returns Object reference to the value in the element with the specified key, or a null object reference if this unordered object map contains no mapping for that key.

Discussion A return value of null does not necessarily indicate that no element has the specified key. It is possible that this unordered object map explicitly maps the key to null. You can use the `containsKey` member function to distinguish these two cases.

Variant 2 finds the element whose key is “equal” to the specified lookup data, as determined by the comparator for this unordered object map. It is useful if this unordered object map has an application-defined comparator that can identify a key based on class-specific data.

See also `put`
`addAll`

keyIterator

Initializes a scalable-collection iterator to find all the keys of this unordered object map.

```
virtual ooCollectionIterator *keyIterator() const;
```

Returns A pointer to a scalable-collection iterator for finding the keys of this unordered object map; the caller is responsible for deleting the iterator when it is no longer needed.

Discussion The returned iterator finds the keys in an undefined order; however, it iterates through the key-value pairs of this unordered object map in the same order as does an iterator returned by `valueIterator`.

You must delete the iterator when you have finished using it.

put

Maps the specified key to the specified value in this unordered object map.

```
ooStatus put(
     const ooHandle(ooObj) &keyH,
     const ooHandle(ooObj) &valueH);
```

Parameters *keyH*

Handle to the key.

valueH

Handle to the value.

Returns `oocSuccess` if successful; otherwise, `oocError`.

Discussion If this unordered object map already contains an element with the specified key, this member function replaces the value in that element. Otherwise, this member function adds a new element with the specified key and value.

See also `get`
`addAll`

remove

Removes the element, if any, with the specified key from this unordered object map.

```
virtual ooBoolean remove(
     const ooHandle(ooObj) &keyH);
```

Parameters *keyH*

Handle to the key of the element to be removed.

Returns `oocTrue` if an element was removed; otherwise, `oocFalse`.

See also `add`

removeAllDeleted

Removes from this unordered object map all elements in which either the key or the value has been deleted from the federated database.

```
virtual void removeAllDeleted();
```

Discussion You can call this member function to restore this unordered object map's referential integrity.

valueIterator

Initializes a scalable-collection iterator to find all values in this unordered object map.

```
ooCollectionIterator *valueIterator() const;
```

Returns A pointer to a scalable-collection iterator for finding all the persistent objects used as values in elements of this unordered object map.

Discussion The returned iterator finds the keys in an undefined order; however, it iterates through the key-value pairs of this unordered object map in the same order as does an iterator returned by `keyIterator`.

You must delete the iterator when you have finished using it.

ooHashSet Class

Inheritance: **ooObj->ooCollection->ooHashSet**

Handle Class: `ooHandle(ooHashSet)`

Object-Reference Class: `ooRef(ooHashSet)`

The persistence-capable class `ooHashSet` represents *unordered sets* of persistent objects with no duplicate elements.

See:

- "Reference Summary" on page 284 for an overview of member functions
- "Reference Index" on page 285 for a list of member functions

To use this class, your application must include the `ooCollections.h` header file. For UNIX linking information, see *Installation and Platform Notes for UNIX*.

About Unordered Sets

An unordered set is an unordered collection of persistent objects with no duplicate elements. Unordered sets are *scalable* collections, that is, they can increase in size with minimal performance degradation. They are implemented using an extendible hashing mechanism so that elements can be added, deleted, and retrieved efficiently. The hash value for each element is computed from the element itself.

The hash values for elements of an unordered set are computed by the set's corresponding comparator. If an unordered set has a default comparator, the hash values are computed from the OIDs of the elements.

For additional information, see Chapter 11, "Persistent Collections," in the Objectivity/C++ programmer's guide.

Working With an Unordered Set

As is the case for any basic object, you specify whether an unordered set is to be transient or persistent when you create it; unordered sets *must be persistent*. You create an unordered set with a call to the `new` operator; the clustering directive in that call specifies where in the federated database to store the unordered set.

Like other persistent objects, unordered sets are normally manipulated through handles or object references. You can store and find an unordered set in the database just as you would any other persistent object.

Related Classes

Two additional classes represent persistent collections of persistent objects:

- `ooTreeSet` represents a *sorted* collection of persistent objects with no duplicate elements.
- `ooTreeList` represents a collection of persistent objects that are *maintained in the order specified when they are added* to the collection. A list *can contain duplicate elements*.

Both sorted sets and lists are implemented using a B-tree data structure.

Reference Summary

In the following table:

- Operators indicated as *(inherited)* are overloaded in this class with no change in behavior; they are documented with the `ooObj` class (page 431), along with the other inherited member functions not listed here.
- Member function indicated as *(inherited)* are inherited from the `ooCollection` class and are documented with that class (page 173).

Creating and Deleting	ooHashSet operator new *(inherited)* operator delete *(inherited)*
Adding and Removing Elements	add addAll *(inherited)* clear *(inherited)* remove removeAll *(inherited)* removeAllDeleted *(inherited)* retainAll *(inherited)*
Getting Elements	get iterator
Getting Information	hashOf size
Finding Auxiliary Objects	admin comparator
Testing	contains containsAll *(inherited)* isEmpty
Viewing in an MROW Transaction	refresh

Reference Index

ooHashSet	Constructs a new empty unordered set.
refresh	Refreshes each container used internally by this unordered collection, except for the container in which the unordered collection itself is stored.
remove	Removes the specified object from this unordered set.
size	Gets the size of this unordered collection.

Constructors

ooHashSet

Constructs a new empty unordered set.

```
1.   ooHashSet(
       int bucketSize = 30011,
       int initialBuckets = 1,
       ooHandle(ooContObj) contAdminH = 0,
       ooHandle(ooContObj) contBucketH = 0);
2.   ooHashSet(
       ooHandle(ooCompare) &compH,
       int bucketSize = 30011,
       int initialBuckets = 1,
       ooHandle(ooContObj) contAdminH = 0,
       ooHandle(ooContObj) contBucketH = 0);
```

Parameters

bucketSize

The size of a hash bucket in the new unordered set's hash table. The size of a hash bucket is the number of elements that can be hashed into each bucket. For optimal performance, the bucket size should be a prime number. If you specify a bucket size that is not a prime number, the next higher prime number is computed and used as the actual bucket size.

initialBuckets

The minimum number of initial hash buckets to create for the new unordered set. The actual number of hash buckets created is the smallest power of 2 that is greater than *initialBuckets*. For example, if *initialBuckets* is 10, the number of hash buckets is 2**4, or 16.

Preallocating multiple hash buckets increases the speed of adding and finding 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 set with a large number of initial hash

buckets requires more disk space, more memory for the directory, and more time to create.

By default, each initial hash bucket is stored in its own newly created container. If you specify a container in the *contBucketH* parameter, however, all initial hash buckets are instead created in that container. In that case, you reduce the storage overhead for the set, but you also reduce concurrent access.

All initial hash buckets are created in the container indicated by *contBucketH*.

contAdminH

Handle to the container in which to store the hash administrator for the new unordered set. The default value (0) creates a container for the hash administrator in the same database as the unordered set itself.

contBucketH

Handle to the container in which to store the initial hash bucket(s) for the new unordered set. The default value (0) creates a separate container for each initial hash bucket, all in the same database as the unordered set itself.

compH

Handle to the comparator for the new unordered set; must be an instance of an application-specific derived class of `ooCompare`.

Discussion

Variant 1 creates an empty unordered set with a default comparator and the specified bucket size.

Variant 2 creates an empty unordered set with the specified comparator and the specified bucket size.

Member Functions

add

Adds the specified object to this unordered set.

```
virtual ooBoolean add(
     const ooHandle(ooObj) &objH);
```

Parameters

objH

Handle to the object to be added.

Returns

`oocTrue` if an element was added; otherwise, `oocFalse`.

Discussion This member function returns false if the specified object is already an element of this unordered set.

See also `remove`

admin

Gets the hash administrator for this unordered collection.

```
virtual ooRef(ooAdmin) admin() const;
```

Returns Object reference to the hash administrator for this unordered collection.

Discussion You typically call this member function when you want to change the way that this unordered collection's internal objects (hash buckets) are assigned to containers. Before you do so, you must cast the returned object reference to `ooRef(ooHashAdmin)`.

comparator

Finds the comparator for this unordered collection.

```
virtual ooRef(ooCompare) comparator() const;
```

Returns Object reference to the comparator for this unordered collection, or null if this unordered collection has a default comparator.

contains

Tests whether this unordered set contains the specified object.

```
1.  virtual ooBoolean contains(
      const ooHandle(ooObj) &objH) const;
2.  ooBoolean contains(
      const void *lookupVal) const;
```

Parameters *objH*

Handle to the element to be tested for containment in this unordered set.

lookupVal

Pointer to data that identifies the object to be tested for containment in this unordered set.

Returns `oocTrue` if this unordered set contains an element equal to the specified object; otherwise, `oocFalse`.

Discussion Variant 2 tests whether any element is “equal” to the specified lookup data, as determined by the comparator for this unordered set. It is useful if this unordered set has an application-defined comparator that can identify an element based on class-specific data.

get

Finds the specified element of this unordered set.

```
virtual ooRef(ooObj) get(const void *lookupVal) const;
```

Parameters *lookupVal*

Pointer to data that identifies the desired element.

Returns Object reference to the element that is “equal” to the specified lookup data, as determined by the comparator for this unordered set, or a null object reference if this unordered set does not contain such an element.

Discussion This member function is useful if this unordered set has an application-defined comparator that can identify an element based on class-specific data.

hashOf

Gets the hash code for the specified object in this unordered collection.

```
int hashOf(const ooHandle(ooObj) &objH) const;
```

Parameters *objH*

Handle to the object whose hash value is to be computed.

Returns The hash code for the specified object.

Discussion If this unordered collection has a comparator of an application-defined class, that comparator computes the hash code; otherwise, the hash code is computed from the object’s OID.

isEmpty

Tests whether this unordered collection is empty.

```
virtual ooBoolean isEmpty() const;
```

Returns `oocTrue` if this unordered collection has no elements; otherwise, `oocFalse`.

iterator

Initializes a scalable-collection iterator to find the elements of this unordered set.

```
virtual ooCollectionIterator *iterator() const;
```

Returns A pointer to a scalable-collection iterator for finding the elements of this unordered set; the caller is responsible for deleting the iterator when it is no longer needed.

Discussion The returned iterator finds the elements in an undefined order.

You must delete the iterator when you have finished using it.

refresh

Refreshes each container used internally by this unordered collection, except for the container in which the unordered collection itself is stored.

```
virtual ooStatus refresh(ooMode &openMode) const;
```

Parameters *openMode*

Intended level of access to each refreshed container:

- Specify `oocRead` to open the container for read. This implicitly requests a read lock on the container.
- Specify `oocUpdate` to open the container for update (read and write). This implicitly requests an update lock on the container.

Returns `oocSuccess` if every container can be refreshed; otherwise, `oocError`.

Discussion You typically call this member function when you need to refresh your view of an unordered collection that you are reading in an MROW transaction. This member function calls `refreshOpen` on each container that is used internally by the unordered collection—that is, on the administrator and hash-bucket containers maintained by the unordered collection. This member function does not refresh the container in which the unordered collection itself is stored, nor does it necessarily refresh the containers that store the collection's elements.

remove

Removes the specified object from this unordered set.

```
virtual ooBoolean remove(
     const ooHandle(ooObj) &objH);
```

Parameters *objH*
Handle to the object to be removed.

Returns `oocTrue` if an element was removed; otherwise, `oocFalse`.

See also `add`

size

Gets the size of this unordered collection.

```
virtual int size() const;
```

Returns The number of elements in this unordered collection.

ooItr(*appClass*) Class

Inheritance: **ooHandle(*appClass*)->ooItr(*appClass*)**

The non-persistence-capable class ooItr(*appClass*) represents an *object iterator* for finding instances of the application-defined persistence-capable class *appClass*. *appClass* can be either a basic-object class (derived from ooObj) or a container class (derived from ooContObj).

See:

- "Reference Summary" on page 294 for an overview of member functions
- "Reference Index" on page 295 for a list of member functions

To use the ooItr(*appClass*) class, you must include and compile with files generated by the DDL processor, as described in "Obtaining Generated Class Definitions" on page 294.

About Class-Specific Object Iterators

When an application defines a persistence-capable class *appClass* and adds it to the federated-database schema, the DDL processor generates the corresponding iterator class ooItr(*appClass*).

An object iterator of class ooItr(*appClass*) finds objects of class *appClass* and its derived classes.

- You can initialize the object iterator to find objects below a given storage object in the storage hierarchy. To do so, call the object iterator's scan member function.
- If *appClass* is the destination class for the to-many association *linkName*, you can initialize the object iterator to find the destination objects linked to a given source object by this association. To do so, cast the object iterator to type ooItr(ooObj) and pass it as a parameter when you call the *linkName* member function on the source object.

You advance a class-specific object iterator, use it to reference objects in the iteration set, and termination the iteration as you would with any object iterator. For more information, see the ooItr(ooObj) class.

Obtaining Generated Class Definitions

To use the ooItr(*appClass*) class, you must include either the primary header file or the references header file generated by the DDL processor for *appClass*. Thus, if *appClass* is defined in the DDL file *classDefFile*.ddl, you must include one of the following files:

- The primary header file *classDefFile*.h
- The references header file *classDefFile*_ref.h

Furthermore, you must compile the method implementation file *classDefFile*_ddl.cxx with your application code files.

For more information about DDL-generated files and how to use them, see the Objectivity/C++ Data Definition Language book.

When *appClass* is a Template Class

When *appClass* is a persistence-capable template class with multiple parameters, the name of the generated object-iterator class contains the symbol OO_COMMA to separate the template parameters. For example, for a persistence-capable template class Example<Float, Node>, the generated object-iterator class is ooItr(Example<Float OO_COMMA Node>). This is because the macro syntax of the generated class name interprets embedded commas as separators between the as macro parameters instead of as separators between the template parameters.

Reference Summary

Creating an Object Iterator	ooItr(appClass)
Finding Objects	end next scan

Reference Index

Constructors

ooItr(*appClass*)

Default constructor that constructs a new uninitialized object iterator for finding instances of *appClass* and its derived classes.

```
ooItr(appClass)();
```

Member Functions

end

Explicitly terminates iteration by this object iterator.

```
ooStatus end();
```

Discussion

You can call this member function to signal that you are finished using this object iterator even though it has not yet found all the objects in the iteration set.

next

Advances this object iterator to the next object in the iteration set.

```
ooBoolean next();
```

Returns `oocTrue` if another object is found; `oocFalse` if all objects in the iteration set have been found or if an error occurred.

scan

Initializes this object iterator to find all instances of *`appClass`* and its derived classes in the specified storage object, satisfying the specified condition, if any.

1.
```
ooStatus scan(
  const ooRefHandle(ooObj) &storageObject,
  const ooMode openMode = oocNoOpen);
```

2.
```
ooStatus scan(
  const ooRefHandle(ooObj) &storageObject,
  const ooLookupKey &lookupKey,
  const ooMode openMode = oocNoOpen);
```

3.
```
ooStatus scan(
  const ooRefHandle(ooObj) &storageObject,
  const char *predicate);
```

4.
```
ooStatus scan(
  const ooRefHandle(ooObj) &storageObject,
  const ooMode openMode,
  const ooAccessMode access,
  const char *predicate);
```

Parameters

`storageObject`

Object reference or handle to the container, database, or federated database to be scanned. If you specify a basic object, its container is scanned.

If *`appClass`* is a container class, and *`storageObject`* specifies a container or basic object, this object iterator is set to null (initialized with an empty iteration set).

(*FTO*) You may not scan an autonomous partition.

`lookupKey`

Lookup key representing the condition that each found object must satisfy.

`openMode`

Intended level of access to each object found by the `next` member function:

- `oocNoOpen` (the default in variants 1 and 2) causes `next` to obtain a reference to the next object without opening it.
- `oocRead` causes `next` to open the next object for read.
- `oocUpdate` causes `next` to open the next object for update.

`access`

Limits the data members that can be specified in *`predicate`*:

- `oocPublic` permits the predicate to test only public data members, thus preserving encapsulation.
- `oocAll` permits the predicate to test any data member, thus decreasing encapsulation. To preserve maximum encapsulation, you should specify `oocAll` only within member functions of the class you are querying.

`predicate`

Condition that objects must satisfy to be found by this object iterator. This string must be a valid expression in the predicate query language. This object iterator finds only those objects that satisfy the *`predicate`*.

Returns

`oocSuccess` if successful; otherwise `oocError`.

Discussion

This scan operation initializes this object iterator to find objects of class *`appClass`* or any class derived from *`appClass`*. If no such objects exist in the specified storage object, this object iterator is set to null and `scan` returns `oocSuccess`. (This object iterator's `next` member function will return `oocFalse`, however.)

Variant 3 finds persistent objects without opening them; the other variants may open the found objects, as specified by the *`openMode`* parameter.

If you specify the *`predicate`* parameter and the specified storage object has an index over the appropriate objects, `scan` can use the index to optimize its search. (You may need to enable index usage; see the `ooUseIndex` global function.) If no index is available, `scan` finds objects by inspection.

An error is signaled if *`predicate`* tries to test a non-existent data member, or if *`predicate`* tries to test a protected or private data member when the *`access`* parameter is `oocPublic`.

If you specify the *`lookupKey`* parameter, `scan` finds objects *only* if a compatible index exists in specified storage object; otherwise, this object iterator is set to null. For performance reasons, you should specify the *`lookupKey`* parameter only if you know the specified storage object contains such an index. You can test the storage object for compatible indexes by calling the `ooLookupKey::anyIndex` member function. (Indexes need not be enabled to scan with a lookup key.)

See also

`next`

ooItr(ooAPObj) Class

Inheritance: **ooItr(ooAPObj)**

The non-persistence-capable class `ooItr(ooAPObj)` represents an *object iterator* for finding the autonomous partitions in a federated database.

See:

- “Reference Summary” on page 300 for an overview of member functions
- “Reference Index” on page 300 for a list of member functions

About Autonomous-Partition Iterators

An iterator of class `ooItr(ooAPObj)` finds the autonomous partitions of the open federated database. You can initialize an autonomous-partition iterator in either of two ways:

- Call its `scan` member function, passing a handle to the federated database as a parameter.
- Call the `contains` member function on a handle to the federated database, passing the autonomous-partition iterator as a parameter.

You advance an autonomous-partition iterator, use it to reference autonomous partitions in the iteration set, and termination the iteration as you would with any object iterator. For more information, see the `ooItr(ooObj)` class.

Reference Summary

Creating an Object Iterator	ooItr(ooAPObj)
Finding Autonomous Partitions	end next scan

Reference Index

Constructors and Destructors

ooItr(ooAPObj)

Default constructor that constructs a new uninitialized object iterator for finding autonomous partitions.

```
ooItr(ooAPObj)();
```

Member Functions

end

Explicitly terminates iteration by this object iterator.

```
ooStatus end();
```

Discussion

You can call this member function to signal that you are finished using this autonomous-partition iterator even though it has not yet found all the autonomous partitions in the iteration set.

next

Advances this object iterator to the next autonomous partition in the iteration set.

```
ooBoolean next();
```

Returns

`oocTrue` if another autonomous partition is found; `oocFalse` if all autonomous partitions in the iteration set have been found or if an error occurred.

scan

Initializes this object iterator to find all autonomous partitions in the specified federated database.

```
ooStatus scan(
     const ooHandle(ooFDObj) &federation,
     const ooMode openMode = oocNoOpen);
```

Parameters

federation

Handle to the federated database to be scanned for autonomous partitions.

openMode

Intended level of access to each autonomous partition found by the `next` member function:

- `oocNoOpen` (the default) causes `next` to obtain a reference to the next partition without opening it.
- `oocRead` causes `next` to open the next partition for read.
- `oocUpdate` causes `next` to open the next partition for update.

Returns

`oocSuccess` if successful; otherwise `oocError`.

Discussion

If no autonomous partitions exist within the federated database, this object iterator is set to null and `scan` returns `oocSuccess`. (This object iterator's `next` member function will return `oocFalse`, however.)

Calling this member function on an autonomous-partition iterator is equivalent to passing the iterator to the `contains` member function on an object reference or handle to the federated database.

ooItr(ooContObj) Class

Inheritance: **ooHandle(ooContObj)->ooItr(ooContObj)**

The non-persistence-capable class `ooItr(ooContObj)` represents a *container iterator*—that is, an object iterator for finding groups of containers in a database or federated database.

See:

- "Reference Summary" on page 304 for an overview of member functions
- "Reference Index" on page 304 for a list of member functions

About Container Iterators

An iterator of class `ooItr(ooContObj)` finds the containers in the open federated database or a particular database. You can initialize a container iterator in either of two ways:

- Call its `scan` member function, passing a handle to the federated database or a database as a parameter.
- Call the `contains` member function on a handle to a database, passing the container iterator as a parameter.

You advance a container iterator, use it to reference containers in the iteration set, and termination the iteration as you would with any object iterator. For more information, see the `ooItr(ooObj)` class.

NOTE Operations that initialize a container iterator find default containers (instances of `ooDefaultContObj`). When advancing such an iterator, you should test for the default container before performing any operation that deletes or moves the found container. For example, when the iterator references the next found container, you could call the container's inherited `ooIsKindOf` member function.

Reference Summary

Creating an Object Iterator	ooItr(ooContObj)
Finding Containers	end next scan

Reference Index

end	Explicitly terminates iteration by this container iterator.
next	Advances this container iterator to the next container in the iteration set.
ooItr(ooContObj)	Default constructor that constructs a new uninitialized container iterator.
scan	Initializes this container iterator to find all containers in the specified storage object, satisfying the specified condition, if any.

Constructors and Destructors

ooItr(ooContObj)

Default constructor that constructs a new uninitialized container iterator.

```
ooItr(ooContObj)();
```

Member Functions

end

Explicitly terminates iteration by this container iterator.

```
ooStatus end();
```

Discussion

You can call this member function to signal that you are finished using this container iterator even though it has not yet found all the containers in the iteration set.

next

Advances this container iterator to the next container in the iteration set.

```
ooBoolean next();
```

Returns

`oocTrue` if another container is found; `oocFalse` if all containers in the iteration set have been found or if an error occurred.

scan

Initializes this container iterator to find all containers in the specified storage object, satisfying the specified condition, if any.

```
1.  ooStatus scan(
      const ooRefHandle(ooObj) &storageObject,
      const ooMode openmode = oocNoOpen);
2.  ooStatus scan(
      const ooRefHandle(ooObj) &storageObject,
      const ooLookupKey &lookupKey,
      const ooMode openmode = oocNoOpen);
3.  ooStatus scan(
      const ooRefHandle(ooObj) &storageObject,
      const char *predicate);
4.  ooStatus scan(
      const ooRefHandle(ooObj) &storageObject,
      const ooMode openMode,
      const ooAccessMode access,
      const char *predicate);
```

Parameters

`storageObject`

Object reference or handle to the database or federated database to be scanned for containers. If you specify a container or basic object, this container iterator is set to null (initialized with an empty iteration set). (*FTO*) You may not scan an autonomous partition.

`lookupKey`

Condition that each found container must satisfy.

openMode

Intended level of access to each container found by the `next` member function:

- `oocNoOpen` (the default in variants 1 and 2) causes `next` to obtain a reference to the next container without opening it.
- `oocRead` causes `next` to open the next container for read.
- `oocUpdate` causes `next` to open the next container for update.

access

Limits the data members that can be specified in *`predicate`*:

- `oocPublic` permits the predicate to test only public data members, thus preserving encapsulation.
- `oocAll` permits the predicate to test any data member, thus decreasing encapsulation. To preserve maximum encapsulation, you should specify `oocAll` only within member functions of the class you are querying.

predicate

Condition that each retrieved container must satisfy. This string must be a valid expression in the predicate query language. This container iterator finds only those containers that satisfy the *`predicate`*.

Returns

`oocSuccess` if successful; otherwise `oocError`.

Discussion

This scan operation initializes this container iterator to find objects of class `ooContObj` or any class derived from `ooContObj`. If no such objects exist in the specified storage object, this container iterator is set to null and `scan` returns `oocSuccess`. (This container iterator's `next` member function will return `oocFalse`, however.)

Variant 3 finds containers without opening them; the other variants may open the found objects, as specified by the *`openMode`* parameter.

If you specify the *`predicate`* parameter and the specified storage object has an index over the appropriate objects, `scan` can use the index to optimize its search. (You may need to enable index usage; see the `ooUseIndex` global function.) If no index is available, `scan` finds objects by inspection.

An error is signaled if *`predicate`* tries to test a non-existent data member, or if *`predicate`* tries to test a protected or private data member when the *`access`* parameter is `oocPublic`.

If you specify the *`lookupKey`* parameter, the `scan` member function finds objects only if a compatible index exists in the specified scope; otherwise this container iterator is set to null. For performance reasons, you should specify the *`lookupKey`* parameter only if you know the specified scope contains such an index. You can test the scope for compatible indexes by calling the

`ooLookupKey::anyIndex` member function. (Indexes need not be enabled to scan with a lookup key.)

ooItr(ooDBObj) Class

Inheritance: **ooHandle(ooObj)->ooItr(ooDBObj)**

The non-persistence-capable class `ooItr(ooDBObj)` represents a *database iterator*—that is, an object iterator for finding the databases in a federated database.

See:

- “Reference Summary” on page 310 for an overview of member functions
- “Reference Index” on page 310 for a list of member functions

About Database Iterators

An iterator of class `ooItr(ooDBObj)` finds the databases in the open federated database. You can initialize a database iterator in either of two ways:

- Call its `scan` member function, passing a handle to the federated database as a parameter.
- Call the `contains` member function on a handle to the federated database, passing the database iterator as a parameter.

You advance a database iterator, use it to reference databases in the iteration set, and termination the iteration as you would with any object iterator. For more information, see the `ooItr(ooObj)` class.

Reference Summary

Creating an Object Iterator	ooItr(ooDBObj)
Finding Databases	end next scan

Reference Index

Constructors and Destructors

ooItr(ooDBObj)

Default constructor that constructs a new uninitialized database iterator.

```
ooItr(ooDBObj)();
```

Member Functions

end

Explicitly terminates iteration by this database iterator.

```
ooStatus end();
```

Discussion You can call this member function to signal that you are finished using this database iterator even though it has not yet found all the databases in the iteration set.

next

Advances this database iterator to the next database in the iteration set.

```
ooBoolean next();
```

Returns `oocTrue` if another database is found; `oocFalse` if all databases in the iteration set have been found or if an error occurred.

scan

Initializes this database iterator to find all databases in the specified federated database.

```
ooStatus scan(
     const ooHandle(ooFDObj) &federation,
     const ooMode openmode = oocNoOpen);
```

Parameters *`federation`*

Handle to the federated database to be scanned for databases.

`openMode`

Intended level of access to each database found by the `next` member function:

- `oocNoOpen` (the default) causes `next` to obtain a reference to the next database without opening it.
- `oocRead` causes `next` to open the next database for read.
- `oocUpdate` causes `next` to open the next database for update.

Returns `oocSuccess` if successful; otherwise `oocError`.

Discussion If no databases exist within the federated database, this database iterator is set to null and `scan` returns `oocSuccess`. (The database iterator's `next` member function will return `oocFalse`, however.)

Calling this member function on a database iterator is equivalent to passing the database iterator to the `contains` member function on an object reference or handle to the federated database.

ooItr(ooObj) Class

Inheritance: **`ooHandle(ooObj)->ooItr(ooObj)`**

The non-persistence-capable class `ooItr(ooObj)` represents an *object iterator* for finding Objectivity / DB objects—that is, instances of `ooObj` or classes derived from `ooObj`. The `ooItr(ooObj)` class is also the base class for all of object-iterator classes.

See:

- "Reference Summary" on page 315 for an overview of member functions
- "Reference Index" on page 315 for a list of member functions

About Object Iterators

An object iterator steps through a group of objects found in the federated database. During a transaction, you can create and initialize an object iterator to find a specified group of Objectivity / DB objects. For example, you can use an object iterator to find all the basic objects in a particular container. The group of objects to be found is called an *iteration set*.

An object iterator consists of:

- A description of an iteration set and the object iterator's current location in this set.
- Other information, such as the intended level of access to each found object.
- A handle for referencing the object at the current location in the iteration set.

When you create a null object iterator, its state is undefined and its handle is null. Initializing the object iterator identifies an iteration set and locates the object iterator just before the first object in the set. If the set is nonnull, the first call to the `next` member function advances the object iterator to the first object in the set and initializes the handle to reference it. Successive calls to `next` advance through the set, so that the handle references each object in turn. An object

iterator makes a single pass through the iteration set, returning the objects in the set in an undefined order. When the end of the iteration set is reached, the `next` member function returns `oocFalse`.

An application can initialize an object iterator of type `ooItr(ooObj)` in a number of ways. Most Objectivity / C++ operations that initialize iterators of this type do so to find persistent objects (containers and basic objects), although occasionally, such iterators are initialized to find Objectivity / DB objects of any class. The most common ways of initializing an object iterator include:

- Calling the object iterator's `scan` member function to find the persistent objects that reside in a particular storage object.
- Passing the object iterator to the *`linkName`* member function on a persistent object to find the destination objects of a to-many association.
- Passing the object iterator to the `contains` member function on a container to find the basic objects in the container.

Note that initializing an object iterator does not result in an intermediate collection in memory. Objects can be added, moved, or deleted from the database while the object iterator is active, and such changes can affect the set of objects returned by the object iterator.

Iteration is terminated automatically after the object iterator has found all objects in an iteration set. 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 its `end` member function. Doing so signals that you will not use the object iterator again and that its 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.

Related Classes

In addition to object iterators of class `ooItr(`*`className`*`)`, Objectivity / C++ provides three other kinds of iterators:

- A name-map iterator is an instance of `ooMapItr`; it steps through the key-value pairs in a name map.

- A scalable-collection iterator is an instance of a class derived from `ooCollectionIterator`; it steps through the objects in a scalable persistent collection.
- A VArray iterator is an instance of a class created from the class template `d_Iterator<`*`element_type`*`>`; it steps through the elements of a VArray. The *`element_type`* parameter specifies the type of elements in the VArray.

Reference Summary

Creating an Object Iterator	`ooItr(ooObj)`

Finding Objects	`end`
	`next`
	`scan`

Reference Index

`end`	Explicitly terminates iteration by this object iterator.
`next`	Advances this object iterator to the next Objectivity/DB object in the iteration set.
`ooItr(ooObj)`	Default constructor that constructs a new uninitialized object iterator for finding Objectivity/DB objects.
`scan`	Initializes this object iterator to find all persistent objects in the specified storage object, satisfying the specified condition, if any.

Constructors

ooItr(ooObj)

Default constructor that constructs a new uninitialized object iterator for finding Objectivity / DB objects.

```
ooItr(ooObj)();
```

Member Functions

end

Explicitly terminates iteration by this object iterator.

```
ooStatus end();
```

Discussion

You can call this member function to signal that you are finished using this object iterator even though it has not yet found all the objects in the iteration set.

next

Advances this object iterator to the next Objectivity / DB object in the iteration set.

```
ooBoolean next();
```

Returns

`oocTrue` if another object is found; `oocFalse` if the iteration set is null, if all the objects in the iteration set have been found, or if an error occurred.

scan

Initializes this object iterator to find all persistent objects in the specified storage object, satisfying the specified condition, if any.

```
1.  ooStatus scan(
      const ooRefHandle(ooObj) &storageObject,
      const ooMode openmode = oocNoOpen);
2.  ooStatus scan(
      const ooRefHandle(ooObj) &storageObject,
      const ooLookupKey &lookupKey,
      const ooMode openmode = oocNoOpen);
3.  ooStatus scan(
      const ooRefHandle(ooObj) &storageObject,
      const char *predicate);
4.  ooStatus scan(
      const ooRefHandle(ooObj) &storageObject,
      const ooMode openMode,
      const ooAccessMode access,
      const char *predicate);
```

Parameters

storageObject

Object reference or handle to the container, database, or federated database to be scanned for persistent objects:

- If you specify a container, the iteration set includes all basic objects in the container.
- If you specify a database or federated database, the iteration set includes all persistent objects (that is, basic objects and containers) in the database or federated database.

If you specify a basic object, its container is scanned for basic objects.

(*FTO*) You may not scan an autonomous partition.

lookupKey

Lookup key that represents the condition that each found object must satisfy.

openMode

Intended level of access to each object found by the `next` member function:

- `oocNoOpen` (the default in variants 1 and 2) causes `next` to obtain a reference to the next object without opening it.
- `oocRead` causes `next` to open the next object for read.
- `oocUpdate` causes `next` to open the next object for update.

access

Limits the data members that can be specified in *`predicate`*:

- `oocPublic` permits the predicate to test only public data members, thus preserving encapsulation.
- `oocAll` permits the predicate to test any data member, thus decreasing encapsulation. To preserve maximum encapsulation, you should specify `oocAll` only within member functions of the class you are querying.

predicate

Condition that each found object must satisfy. This string must be a valid expression in the predicate query language. This object iterator finds only those objects that satisfy the *`predicate`*.

Returns

`oocSuccess` if successful; otherwise `oocError`.

Discussion

This scan operation initializes this object iterator to find objects of class `ooObj`, class `ooContObj`, any basic-object class derived from `ooObj`, or any container class derived from `ooContObj`. If no such objects exist in the specified storage object, this object iterator is set to null and `scan` returns `oocSuccess`. (The object iterator's `next` member function will return `oocFalse`, however.)

Variant 3 finds persistent objects without opening them; the other variants may open the found objects, as specified by the *`openMode`* parameter.

If you specify the *`predicate`* parameter and the specified storage object has an index over the appropriate objects, `scan` can use the index to optimize its search.

(You may need to enable index usage; see the ooUseIndex global function.) If no index is available, scan finds objects by inspection.

An error is signaled if *predicate* tries to test a non-existent data member, or if *predicate* tries to test a protected or private data member when the *access* parameter is oocPublic.

If you specify the *lookupKey* parameter, scan finds objects *only* if a compatible index exists in specified storage object; otherwise, this object iterator is set to null. For performance reasons, you should specify the *lookupKey* parameter only if you know the specified storage object contains such an index. You can test the storage object for compatible indexes by calling the ooLookupKey::anyIndex member function. (Indexes need not be enabled to scan with a lookup key.)

See also next

oojArray Class

Inheritance: **ooObj->oojArray**

Handle Class: `ooHandle(oojArray)`

Object-Reference Class: `ooRef(oojArray)`

The persistence-capable class `oojArray` is the abstract base class for all Java-compatibility classes that represent variable-size arrays. Each concrete derived class represents variable-size arrays with elements of a particular type.

See:

- "Reference Index" on page 321 for a list of member functions

To use the Java-compatibility classes, your application source must include the `javaBuiltins.h` header file.

About Java Arrays

Unlike a C++ object, a Java object cannot contain another embedded object. As a consequence, a Java object cannot have an embedded variable-size array (VArray) the way a C++ object can. A variable-sized array in a Java persistent object is stored in an Objectivity / DB federated database as an object reference to a persistent array object of some Java-compatibility class. The persistent array object is a wrapper for a VArray with the same number of elements of the appropriate type. For example, a Java array of 16-bit integers (of the Java type `short[]`) is stored as an object reference of the type `ooRef(oojArrayOfInt16)`.

NOTE Java does not support fixed-size arrays.

Because the `oojArray` class is abstract, you never create instances of it; instead, you work with instances of its concrete derived classes.

You should not create your own subclasses of this class.

Multidimensional Arrays

Unlike a VArray, a Java array can be multidimensional.

Element Order

When the a multidimensional Java array is "flattened" into a one-dimensional VArray, elements are written to the VArray in row-major order, that is, with the last array index varying first.

EXAMPLE Consider the following two-dimensional 2-by-3 array of 16-bit integers:

```
11    12    13
21    22    23
```

This array could be represented in Java by a `short[]` array (called `ary` for illustration) whose individual elements have the values:

```
ary[0][0] = 11     ary[0][1] = 12     ary[0][2] = 13
ary[1][0] = 21     ary[1][1] = 22     ary[1][2] = 23
```

If that array is stored in a database and accessed by a C++ application as an `oojArrayOfInt16`, the array object's corresponding VArray (called `vary` for illustration) would have six elements with the values:

```
vary[0] = 11
vary[1] = 12
vary[2] = 13
vary[3] = 21
vary[4] = 22
vary[5] = 23
```

Array Dimensions

The persistent array's `getDimensionsArray` member function returns a VArray containing the dimensions of the Java array.

- The number of elements in the VArray is the number of dimensions in the Java array.
- The value of each element is the size of the corresponding array.

For example, consider a 3-dimensional Java array with dimensions 10 by 10 by 3. The corresponding C++ persistent array object wraps a VArray with 300 elements. The array object's dimensions array is a VArray of 3 elements: 10, 10, and 3.

WARNING You must not change the size of the VArray for a multidimensional array object. If you do so, the stored dimensions for the array object will be incorrect and applications will not be able to index the elements correctly.

Reference Index

Member Functions

getDimensionsArray

Gets the dimensions of the Java array corresponding to this array object.

```
ooVArrayT<int32> &getDimensionsArray();
```

Returns A VArray containing the dimensions of the Java array. If the Java array is unidimensional, the returned VArray will contain a single element, which is the size (number of elements) in the Java array. Otherwise, the returned VArray will contain one element from each dimension of the Java array.

oojArrayOfBoolean Class

Inheritance: **ooObj->oojArray->oojArrayOfBoolean**

Handle Class: `ooHandle(oojArrayOfBoolean)`

Object-Reference Class: `ooRef(oojArrayOfBoolean)`

The persistence-capable class `oojArrayOfBoolean` is a Java-compatibility class that represents a variable-size array of Boolean elements.

See:

- "Reference Index" on page 324 for a list of member functions

To use the Java-compatibility classes, your application source must include the `javaBuiltins.h` header file.

About Boolean Arrays

A Java array of Boolean values (of the Java type `boolean[]`) is stored in an Objectivity / DB federated database as an object reference to a persistent array object of the class `oojArrayOfBoolean`. If your application interoperates with a Java application to access objects with a field that contains an array of Boolean values, you can define the corresponding data member of your C++ class to be of type `ooRef(oojArrayOfBoolean)`.

An instance of `oojArrayOfBoolean` is a wrapper for a variable-size array (VArray) with elements of the type `uint8`. You can obtain the VArray by calling the array object's `getBooleanArray` member function.

As is the case for all persistence-capable classes, you specify whether an instance of `oojArrayOfBoolean` is to be transient or persistent when you create it. You create the Boolean array object with a call to the `new` operator; the clustering

directive in that call specifies whether to make the new array object persistent and, if so, where to locate it.

Like other persistent objects, persistent Boolean arrays are normally manipulated through handles or object references.

Reference Index

This class overloads `operator new` and `operator delete`, which behave as described for the `ooObj` class (page 431).

`oojArrayOfBoolean`	Constructs a new Boolean array.
`getBooleanArray`	Gets this Boolean array's VArray.

Constructors

oojArrayOfBoolean

Constructs a new Boolean array.

```
1.  oojArrayOfBoolean();
2.  oojArrayOfBoolean(int initialSize);
```

Parameters *initialSize*

Initial number of elements for which space should be allocated.

Discussion Variant 1 is the default constructor. It creates a Boolean array whose VArray has no element vector.

Variant 2 creates a Boolean array whose VArray's element vector contains the specified number of elements. If *initialSize* is 0, no element vector is allocated for the VArray.

Member Functions

getBooleanArray

Gets this Boolean array's VArray.

```
ooVArrayT<uint8> &getBooleanArray();
```

Returns This Boolean array's VArray.

oojArrayOfCharacter Class

Inheritance: **ooObj->oojArray->oojArrayOfCharacter**

Handle Class: `ooHandle(oojArrayOfCharacter)`

Object-Reference Class: `ooRef(oojArrayOfCharacter)`

The persistence-capable class `oojArrayOfCharacter` is a Java-compatibility class that represents a variable-size array of characters.

See:

- "Reference Index" on page 328 for a list of member functions

To use the Java-compatibility classes, your application source must include the `javaBuiltins.h` header file.

About Character Arrays

A Java array of characters (of the Java type `char[]`) is stored in an Objectivity / DB federated database as an object reference to a persistent array object of the class `oojArrayOfCharacter`. If your application interoperates with a Java application to access objects with a field that contains an array of characters, you can define the corresponding data member of your C++ class to be of type `ooRef(oojArrayOfCharacter)`.

An instance of `oojArrayOfCharacter` is a wrapper for a variable-size array (VArray) with elements of the type `uint16`. You can obtain the VArray by calling the array object's `getCharacterArray` member function.

As is the case for all persistence-capable classes, you specify whether an instance of `oojArrayOfCharacter` is to be transient or persistent when you create it. You create the character array object with a call to the `new` operator; the clustering

directive in that call specifies whether to make the new array object persistent and, if so, where to locate it.

Like other persistent objects, persistent character arrays are normally manipulated through handles or object references.

Reference Index

This class overloads `operator new` and `operator delete`, which behave as described for the `ooObj` class (page 431).

`getCharacterArray`	Gets this character array's VArray.
`oojArrayOfCharacter`	Constructs a new character array.

Constructors

oojArrayOfCharacter

Constructs a new character array.

```
1.  oojArrayOfCharacter();
2.  oojArrayOfCharacter(int initialSize);
```

Parameters

initialSize

Initial number of elements for which space should be allocated.

Discussion

Variant 1 is the default constructor. It creates a character array whose VArray has no element vector.

Variant 2 creates a character array whose VArray's element vector contains the specified number of elements. If *initialSize* is 0, no element vector is allocated for the VArray.

Member Functions

getCharacterArray

Gets this character array's VArray.

```
ooVArrayT<uint16> &getCharacterArray();
```

Returns This character array's VArray.

oojArrayOfDouble Class

Inheritance: **ooObj->oojArray->oojArrayOfDouble**

Handle Class: `ooHandle(oojArrayOfDouble)`

Object-Reference Class: `ooRef(oojArrayOfDouble)`

The persistence-capable class `oojArrayOfDouble` is a Java-compatibility class that represents a variable-size array of double-precision floating-point numbers.

See:

- "Reference Index" on page 332 for a list of member functions

To use the Java-compatibility classes, your application source must include the `javaBuiltins.h` header file.

About Arrays of Double

A Java array of double-precision floating-point numbers (of the Java type `double[]`) is stored in an Objectivity / DB federated database as an object reference to a persistent array object of the class `oojArrayOfDouble`. If your application interoperates with a Java application to access objects with a field that contains an array of double-precision floating-point numbers, you can define the corresponding data member of your C++ class to be of type `ooRef(oojArrayOfDouble)`.

An instance of `oojArrayOfDouble` is a wrapper for a variable-size array (VArray) with elements of the type `float64`. You can obtain the VArray by calling the array object's `getDoubleArray` member function.

As is the case for all persistence-capable classes, you specify whether an instance of `oojArrayOfDouble` is to be transient or persistent when you create it. You create the array of double with a call to the `new` operator; the clustering directive

in that call specifies whether to make the new array object persistent and, if so, where to locate it.

Like other persistent objects, persistent arrays of double are normally manipulated through handles or object references.

Reference Index

This class overloads `operator new` and `operator delete`, which behave as described for the `ooObj` class (page 431).

`getDoubleArray`	Gets this array of double's VArray.
`oojArrayOfDouble`	Constructs a new array of double.

Constructors

oojArrayOfDouble

Constructs a new array of double.

```
1.  oojArrayOfDouble();
2.  oojArrayOfDouble(int initialSize);
```

Parameters *initialSize*

Initial number of elements for which space should be allocated.

Discussion Variant 1 is the default constructor. It creates an array of double whose VArray has no element vector.

Variant 2 creates an array of double whose VArray's element vector contains the specified number of elements. If *initialSize* is 0, no element vector is allocated for the VArray.

Member Functions

getDoubleArray

Gets this array of double's VArray.

```
ooVArrayT<float64> &getDoubleArray();
```

Returns This array of double's VArray.

oojArrayOfFloat Class

Inheritance: **ooObj->oojArray->oojArrayOfFloat**

Handle Class: `ooHandle(oojArrayOfFloat)`

Object-Reference Class: `ooRef(oojArrayOfFloat)`

The persistence-capable class `oojArrayOfFloat` is a Java-compatibility class that represents a variable-size array of single-precision floating-point numbers.

See:

- "Reference Index" on page 336 for a list of member functions

To use the Java-compatibility classes, your application source must include the `javaBuiltins.h` header file.

About Arrays of Float

A Java array of floating-point numbers (of the Java type `float[]`) is stored in an Objectivity / DB federated database as an object reference to a persistent array object of the class `oojArrayOfFloat`. If your application interoperates with a Java application to access objects with a field that contains an array of floating-point numbers, you can define the corresponding data member of your C++ class to be of type `ooRef(oojArrayOfFloat)`.

An instance of `oojArrayOfFloat` is a wrapper for a variable-size array (VArray) with elements of the type `float32`. You can obtain the VArray by calling the array object's `getFloatArray` member function.

As is the case for all persistence-capable classes, you specify whether an instance of `oojArrayOfFloat` is to be transient or persistent when you create it. You create the array of float with a call to the `new` operator; the clustering directive in

that call specifies whether to make the new array object persistent and, if so, where to locate it.

Like other persistent objects, persistent arrays of float are normally manipulated through handles or object references.

Reference Index

This class overloads `operator new` and `operator delete`, which behave as described for the `ooObj` class (page 431).

`getFloatArray`	Gets this array of float's VArray.
`oojArrayOfFloat`	Constructs a new array of float.

Constructors

oojArrayOfFloat

Constructs a new array of float.

```
1.  oojArrayOfFloat();
2.  oojArrayOfFloat(int initialSize);
```

Parameters *initialSize*

Initial number of elements for which space should be allocated.

Discussion Variant 1 is the default constructor. It creates an array of float whose VArray has no element vector.

Variant 2 creates an array of float whose VArray's element vector contains the specified number of elements. If *initialSize* is 0, no element vector is allocated for the VArray.

Member Functions

getFloatArray

Gets this array of float's VArray.

```
ooVArrayT<float32> &getFloatArray();
```

Returns This array of float's VArray.

oojArrayOfInt8 Class

Inheritance: **ooObj->oojArray->oojArrayOfInt8**

Handle Class: `ooHandle(oojArrayOfInt8)`

Object-Reference Class: `ooRef(oojArrayOfInt8)`

The persistence-capable class `oojArrayOfInt8` is a Java-compatibility class that represents a variable-size array of 8-bit integers.

See:

- "Reference Index" on page 340 for a list of member functions

To use the Java-compatibility classes, your application source must include the `javaBuiltins.h` header file.

About 8-Bit Integer Arrays

A Java array of 8-bit integers (of the Java type `byte[]`) is stored in an Objectivity / DB federated database as an object reference to a persistent array object of the class `oojArrayOfInt8`. If your application interoperates with a Java application to access objects with a field that contains an array of 8-bit integers, you can define the corresponding data member of your C++ class to be of type `ooRef(oojArrayOfInt8)`.

An instance of `oojArrayOfInt8` is a wrapper for a variable-size array (VArray) with elements of the type `int8`. You can obtain the VArray by calling the array object's `getInt8Array` member function.

As is the case for all persistence-capable classes, you specify whether an instance of `oojArrayOfInt8` is to be transient or persistent when you create it. You create the 8-bit integer array with a call to the `new` operator; the clustering directive in

that call specifies whether to make the new array object persistent and, if so, where to locate it.

Like other persistent objects, persistent 8-bit integer arrays are normally manipulated through handles or object references.

Reference Index

This class overloads `operator new` and `operator delete`, which behave as described for the `ooObj` class (page 431).

`getInt8Array`	Gets this 8-bit integer array's VArray.
`oojArrayOfInt8`	Constructs a new 8-bit integer array.

Constructors

oojArrayOfInt8

Constructs a new 8-bit integer array.

```
1.  oojArrayOfInt8();
2.  oojArrayOfInt8(int initialSize);
```

Parameters *initialSize*

Initial number of elements for which space should be allocated.

Discussion Variant 1 is the default constructor. It creates an 8-bit integer array whose VArray has no element vector.

Variant 2 creates an 8-bit integer array whose VArray's element vector contains the specified number of elements. If *initialSize* is 0, no element vector is allocated for the VArray.

Member Functions

getInt8Array

Gets this 8-bit integer array's VArray.

```
ooVArrayT<int8> &getInt8Array();
```

Returns This 8-bit integer array's VArray.

oojArrayOfInt16 Class

Inheritance: **ooObj->oojArray->oojArrayOfInt16**

Handle Class: `ooHandle(oojArrayOfInt16)`

Object-Reference Class: `ooRef(oojArrayOfInt16)`

The persistence-capable class `oojArrayOfInt16` is a Java-compatibility class that represents a variable-size array of 16-bit integers.

See:

- "Reference Index" on page 344 for a list of member functions

To use the Java-compatibility classes, your application source must include the `javaBuiltins.h` header file.

About 16-Bit Integer Arrays

A Java array of 16-bit integers (of the Java type `byte[]`) is stored in an Objectivity / DB federated database as an object reference to a persistent array object of the class `oojArrayOfInt16`. If your application interoperates with a Java application to access objects with a field that contains an array of 16-bit integers, you can define the corresponding data member of your C++ class to be of type `ooRef(oojArrayOfInt16)`.

An instance of `oojArrayOfInt16` is a wrapper for a variable-size array (VArray) with elements of the type `int16`. You can obtain the VArray by calling the array object's `getInt16Array` member function.

As is the case for all persistence-capable classes, you specify whether an instance of `oojArrayOfInt16` is to be transient or persistent when you create it. You create the 16-bit integer array with a call to the `new` operator; the clustering

directive in that call specifies whether to make the new array object persistent and, if so, where to locate it.

Like other persistent objects, persistent 16-bit integer arrays are normally manipulated through handles or object references.

Reference Index

This class overloads `operator new` and `operator delete`, which behave as described for the `ooObj` class (page 431).

`getInt16Array`	Gets this 16-bit integer array's VArray.
`oojArrayOfInt16`	Constructs a new 16-bit integer array.

Constructors

oojArrayOfInt16

Constructs a new 16-bit integer array.

```
1.  oojArrayOfInt16();
2.  oojArrayOfInt16(int initialSize);
```

Parameters *initialSize*

Initial number of elements for which space should be allocated.

Discussion Variant 1 is the default constructor. It creates a 16-bit integer array whose VArray has no element vector.

Variant 2 creates a 16-bit integer array whose VArray's element vector contains the specified number of elements. If *initialSize* is 0, no element vector is allocated for the VArray.

Member Functions

getInt16Array

Gets this 16-bit integer array's VArray.

```
ooVArrayT<int16> &getInt16Array();
```

Returns This 16-bit integer array's VArray.

oojArrayOfInt32 Class

Inheritance: **ooObj->oojArray->oojArrayOfInt32**

Handle Class: `ooHandle(oojArrayOfInt32)`

Object-Reference Class: `ooRef(oojArrayOfInt32)`

The persistence-capable class `oojArrayOfInt32` is a Java-compatibility class that represents a variable-size array of 32-bit integers.

See:

- "Reference Index" on page 348 for a list of member functions

To use the Java-compatibility classes, your application source must include the `javaBuiltins.h` header file.

About 32-Bit Integer Arrays

A Java array of 32-bit integers (of the Java type `byte[]`) is stored in an Objectivity / DB federated database as an object reference to a persistent array object of the class `oojArrayOfInt32`. If your application interoperates with a Java application to access objects with a field that contains an array of 32-bit integers, you can define the corresponding data member of your C++ class to be of type `ooRef(oojArrayOfInt32)`.

An instance of `oojArrayOfInt32` is a wrapper for a variable-size array (VArray) with elements of the type `int32`. You can obtain the VArray by calling the array object's `getInt32Array` member function.

As is the case for all persistence-capable classes, you specify whether an instance of `oojArrayOfInt32` is to be transient or persistent when you create it. You create the 32-bit integer array with a call to the `new` operator; the clustering

directive in that call specifies whether to make the new array object persistent and, if so, where to locate it.

Like other persistent objects, persistent 32-bit integer arrays are normally manipulated through handles or object references.

Reference Index

This class overloads `operator new` and `operator delete`, which behave as described for the `ooObj` class (page 431).

`getInt32Array`	Gets this 32-bit integer array's VArray.
`oojArrayOfInt32`	Constructs a new 32-bit integer array.

Constructors

oojArrayOfInt32

Constructs a new 32-bit integer array.

```
1.  oojArrayOfInt32();
2.  oojArrayOfInt32(int initialSize);
```

Parameters *initialSize*

Initial number of elements for which space should be allocated.

Discussion Variant 1 is the default constructor. It creates a 32-bit integer array whose VArray has no element vector.

Variant 2 creates a 32-bit integer array whose VArray's element vector contains the specified number of elements. If *initialSize* is 0, no element vector is allocated for the VArray.

Member Functions

getInt32Array

Gets this 32-bit integer array's VArray.

```
ooVArrayT<int32> &getInt32Array();
```

Returns This 32-bit integer array's VArray.

oojArrayOfInt64 Class

Inheritance: **ooObj->oojArray->oojArrayOfInt64**

Handle Class: `ooHandle(oojArrayOfInt64)`

Object-Reference Class: `ooRef(oojArrayOfInt64)`

The persistence-capable class `oojArrayOfInt64` is a Java-compatibility class that represents a variable-size array of 64-bit integers.

See:

- "Reference Index" on page 352 for a list of member functions

To use the Java-compatibility classes, your application source must include the `javaBuiltins.h` header file.

About 64-Bit Integer Arrays

A Java array of 64-bit integers (of the Java type `byte[]`) is stored in an Objectivity / DB federated database as an object reference to a persistent array object of the class `oojArrayOfInt64`. If your application interoperates with a Java application to access objects with a field that contains an array of 64-bit integers, you can define the corresponding data member of your C++ class to be of type `ooRef(oojArrayOfInt64)`.

An instance of `oojArrayOfInt64` is a wrapper for a variable-size array (VArray) with elements of the type `int64`. You can obtain the VArray by calling the array object's `getInt64Array` member function.

As is the case for all persistence-capable classes, you specify whether an instance of `oojArrayOfInt64` is to be transient or persistent when you create it. You create the 64-bit integer array with a call to the `new` operator; the clustering

directive in that call specifies whether to make the new array object persistent and, if so, where to locate it.

Like other persistent objects, persistent 64-bit integer arrays are normally manipulated through handles or object references.

Reference Index

This class overloads `operator new` and `operator delete`, which behave as described for the `ooObj` class (page 431).

`getInt64Array`	Gets this 64-bit integer array's VArray.
`oojArrayOfInt64`	Constructs a new 64-bit integer array.

Constructors

oojArrayOfInt64

Constructs a new 64-bit integer array.

```
1.  oojArrayOfInt64();
2.  oojArrayOfInt64(int initialSize);
```

Parameters *initialSize*

Initial number of elements for which space should be allocated.

Discussion Variant 1 is the default constructor. It creates a 64-bit integer array whose VArray has no element vector.

Variant 2 creates a 64-bit integer array whose VArray's element vector contains the specified number of elements. If *initialSize* is 0, no element vector is allocated for the VArray.

Member Functions

getInt64Array

Gets this 64-bit integer array's VArray.

```
ooVArrayT<int64> &getInt64Array();
```

Returns This 64-bit integer array's VArray.

oojArrayOfObject Class

Inheritance: **ooObj->oojArray->oojArrayOfObject**

Handle Class: `ooHandle(oojArrayOfObject)`

Object-Reference Class: `ooRef(oojArrayOfObject)`

The persistence-capable class `oojArrayOfObject` is a Java-compatibility class that represents a persistent variable-size array of object references.

See:

- "Reference Index" on page 356 for a list of member functions

To use the Java-compatibility classes, your application source must include the `javaBuiltins.h` header file.

About Object-Reference Arrays

A Java array of objects is stored in an Objectivity / DB federated database as an object reference to a persistent array object of the class `oojArrayOfObject`. The Java array may be of types shown in the following table.

Java Array Type	Schema Class of Objects Referenced by Array Elements
`java.lang.String[]`	`oojString`
`java.util.Date[]`	`oojDate`
`java.sql.Date[]`	`oojDate`
`java.sql.Time[]`	`oojTime`
`java.sql.Timestamp[]`	`oojTimestamp`

Java Array Type	Schema Class of Objects Referenced by Array Elements
AppClass[], where *AppClass* is an application-defined persistence-capable class whose schema class name is *PCclass*.	*PCclass*
APIclass[], where *APIclass* is a persistence-capable class in the public Objectivity for Java programmer interface (for example, ooContObj or ooMap).	*APIclass*
PCinterface[], where *PCinterface* is an interface (implemented by one or more persistence-capable classes).	The schema class corresponding to a Java class that implements *PCinterface*. (Different elements of the array may reference objects of different classes.)

If your application interoperates with a Java application to access objects with a field that contains an array of object references, you can define the corresponding data member of your C++ class to be of type ooRef(oojArrayOfObject).

An instance of oojArrayOfObject is a wrapper for a variable-size array (VArray) with elements of the type ooRef(ooObj). You can obtain the VArray by calling the array object's getObjectArray member function.

As is the case for all persistence-capable classes, you specify whether an instance of oojArrayOfObject is to be transient or persistent when you create it. You create the object-reference array object with a call to the new operator; the clustering directive in that call specifies whether to make the new array object persistent and, if so, where to locate it.

Like other persistent objects, persistent object-reference arrays are normally manipulated through handles or object references.

Reference Index

This class overloads operator new and operator delete, which behave as described for the ooObj class (page 431).

getObjectArray	Gets this object-reference array's VArray.
oojArrayOfObject	Constructs a new object-reference array.

Constructors

oojArrayOfObject

Constructs a new object-reference array.

```
oojArrayOfObject(int initialSize);
```

Parameters

initialSize

Initial number of elements for which space should be allocated.

Discussion

Variant 1 is the default constructor. It creates an object-reference array whose VArray has no element vector.

Variant 2 creates an object-reference array whose VArray's element vector contains the specified number of elements. If *initialSize* is 0, no element vector is allocated for the VArray.

Member Functions

getObjectArray

Gets this object-reference array's VArray.

```
ooVArrayT<ooRef(ooObj)> &getObjectArray();
```

Returns

This object-reference array's VArray.

oojDate Class

Inheritance: **ooObj->oojDate**

Handle Class: `ooHandle(oojDate)`

Object-Reference Class: `ooRef(oojDate)`

The persistence-capable class `oojDate` is a Java-compatibility class that represents an instant in time with millisecond precision.

See:

- "Reference Index" on page 360 for a list of member functions

To use the Java-compatibility classes, your application source must include the `javaBuiltins.h` header file.

About Dates

A Java date (of the Java class `java.util.Date` or `java.sql.Date`) is stored in an Objectivity / DB federated database as an object reference to a persistent object of the class `oojDate`. If your application interoperates with a Java application to access objects with a field that contains a date, you can define the corresponding data member of your C++ class to be of type `ooRef(oojDate)`.

The class `oojDate` represents an instant in time as the number of milliseconds since January 1, 1970 00:00:00.000 GMT.

As is the case for all persistence-capable classes, you specify whether an instance of `oojDate` is to be transient or persistent when you create it. You create the date object with a call to the `new` operator; the clustering directive in that call specifies whether to make the new date object persistent and, if so, where to locate it.

Like other persistent objects, persistent dates are normally manipulated through handles or object references.

Reference Index

This class overloads operator new and operator delete, which behave as described for the ooObj class (page 431).

getMillis	Gets the millisecond representation of this date.
oojDate	Constructs a new date object.
setMillis	Sets the millisecond representation of this date.

Constructors

oojDate

Constructs a new date object.

```
1.  oojDate();
2.  oojDate(int64 millisecs);
```

Parameters *millisecs*

The millisecond representation of the new date.

Discussion Variant 1 is the default constructor, which creates a date representing January 1, 1970 00:00:00.000 GMT.

Variant 2 creates a date with the specified millisecond representation.

Member Functions

getMillis

Gets the millisecond representation of this date.

```
int64 getMillis();
```

Returns The millisecond representation of this date.

setMillis

Sets the millisecond representation of this date.

```
void setMillis(int64 millisecs);
```

Parameters *millisecs*

The new millisecond representation of this date.

oojString Class

Inheritance: **ooObj->oojString**

Handle Class: `ooHandle(oojString)`

Object-Reference Class: `ooRef(oojString)`

The persistence-capable class `oojString` is a Java-compatibility class that represents a string element of a persistent object-reference array.

See:

- "Reference Index" on page 364 for a list of member functions

To use the Java-compatibility classes, your application source must include the `javaBuiltins.h` header file.

About String Elements

A Java string array (of the Java type `String[]`) is stored in an Objectivity / DB federated database as an object reference to a persistent array object of the class `oojArrayOfObject`. Elements of the array are object references to instances of `oojString`.

This class is a wrapper for a Unicode string of the `ooUtf8String` class; you can obtain the string from a string element by calling the `getStringValue` member function.

As is the case for all persistence-capable classes, you specify whether an instance of `oojString` is to be transient or persistent when you create it. You create the string element with a call to the `new` operator; the clustering directive in that call specifies whether to make the new string element persistent and, if so, where to locate it.

Like other persistent objects, persistent string elements are normally manipulated through handles or object references.

Reference Index

This class overloads `operator new` and `operator delete`, which behave as described for the `ooObj` class (page 431).

`getStringValue`	Gets the Unicode string in this string element.
`oojString`	Default constructor that creates a new string element with an empty string.

Constructors

oojString

Default constructor that creates a new string element with an empty string.

```
oojString();
```

Member Functions

getStringValue

Gets the Unicode string in this string element.

```
ooUtf8String &getStringValue();
```

Returns The Unicode string in this string element.

Discussion Modifying the return string modifies this string element.

oojTime Class

Inheritance: **ooObj->oojTime**

Handle Class: `ooHandle(oojTime)`

Object-Reference Class: `ooRef(oojTime)`

The class `oojTime` is a Java-compatibility class that represents a time with millisecond precision.

See:

- "Reference Index" on page 366 for a list of member functions

To use the Java-compatibility classes, your application source must include the `javaBuiltins.h` header file.

About Times

A Java time object (of the Java class `java.sql.Time`) is stored in an Objectivity / DB federated database as an object reference to a persistent object of the class `oojTime`. If your application interoperates with a Java application to access objects with a field that contains a time, you can define the corresponding data member of your C++ class to be of type `ooRef(oojTime)`.

The class `oojTime` represents an instant in time as the number of milliseconds since midnight. Thus 1000 milliseconds corresponds to the time one second after midnight, namely, 12:00:01 AM.

As is the case for all persistence-capable classes, you specify whether an instance of `oojTime` is to be transient or persistent when you create it. You create the time object with a call to the `new` operator; the clustering directive in that call specifies whether to make the new time object persistent and, if so, where to locate it.

Like other persistent objects, persistent times are normally manipulated through handles or object references.

Reference Index

This class overloads `operator new` and `operator delete`, which behave as described for the `ooObj` class (page 431).

`getMillis`	Gets the millisecond representation of this time.
`oojTime`	Constructs a new time object.
`setMillis`	Sets the millisecond representation of this time.

Constructors

oojTime

Constructs a new time object.

```
1.   oojTime();
2.   oojTime(int64 millisecs);
```

Parameters *millisecs*

The millisecond representation of the new time.

Discussion Variant 1 is the default constructor, which creates a time representing midnight.

Variant 2 creates a time with the specified millisecond representation.

Member Functions

getMillis

Gets the millisecond representation of this time.

```
int64 getMillis();
```

Returns The millisecond representation of this time.

setMillis

Sets the millisecond representation of this time.

```
void setMillis(int64 millisecs);
```

Parameters *millisecs*

The new millisecond representation of this time.

oojTimestamp Class

Inheritance: **`ooObj->oojTimestamp`**

Handle Class: `ooHandle(oojTimestamp)`

Object-Reference Class: `ooRef(oojTimestamp)`

The class `oojTimestamp` is a Java-compatibility class that represents an instant in time with nanosecond precision.

See:

- "Reference Summary" on page 370 for an overview of member functions
- "Reference Index" on page 370 for a list of member functions

To use the Java-compatibility classes, your application source must include the `javaBuiltins.h` header file.

About Timestamps

A Java timestamp (of the Java class `java.sql.Timestamp`) is stored in an Objectivity / DB federated database as an object reference to a persistent object of the class `oojTimestamp`. If your application interoperates with a Java application to access objects with a field that contains a timestamp, you can define the corresponding data member of your C++ class to be of type `ooRef(oojTimestamp)`.

The class `oojTimestamp` represents an instant in time as the number of nanoseconds since January 1, 1970 00:00:00.000000000 GMT, given as an integral part and a fractional part:

- The integral part specifies the number of seconds since January 1, 1970 00:00:00 GMT; however, it is expressed in *milliseconds* rather than seconds.

- The fractional part specifies the number of nanoseconds.

For example, a timestamp representing January 1, 1970 00:00:01.111222333 GMT consists of the integral part 1000 and the fractional part 111222333.

As is the case for all persistence-capable classes, you specify whether an instance of `oojTimestamp` is to be transient or persistent when you create it. You create the timestamp object with a call to the `new` operator; the clustering directive in that call specifies whether to make the new timestamp object persistent and, if so, where to locate it.

Like other persistent objects, persistent timestamps are normally manipulated through handles or object references.

Reference Summary

In the following table, operators indicated as *(inherited)* are overloaded in this class with no change in behavior; they are documented with the `ooObj` class (page 431).

Creating and Deleting a Timestamp	`oojTimestamp` `operator new` *(inherited)* `operator delete` *(inherited)*
Getting Information About the Timestamp	`getMillis` `getNanos`
Setting Information About the Timestamp	`setMillis` `setNanos`

Reference Index

Constructors

oojTimestamp

Constructs a new timestamp object.

```
1.  oojTimestamp();
2.  oojTimestamp(int64 millisecs, int nanosecs);
```

Parameters

millisecs

The integral part of the new timestamp.

nanosecs

The fractional part of the new timestamp.

Discussion

Variant 1 is the default constructor, which creates a timestamp representing January 1, 1970 00:00:00.000000000 GMT.

Variant 2 creates a timestamp with the specified integral and fractional parts.

Member Functions

getMillis

Gets the integral part of this timestamp.

```
int64 getMillis();
```

Returns

The integral part of this timestamp.

getNanos

Gets the fractional part of this timestamp.

```
int32 getNanos();
```

Returns

The fractional part of this timestamp

setMillis

Sets the integral part of this timestamp.

```
void setMillis(int64 millisecs);
```

Parameters *millisecs*

The new integral part of this timestamp.

setNanos

Sets the fractional part of this timestamp.

```
void setNanos(int32 nanosecs);
```

Parameters *nanosecs*

The new fractional part of this timestamp.

ooKeyDesc Class

Inheritance: **ooObj->...->ooKeyDesc**

Handle Class: `ooHandle(ooKeyDesc)`

Object-Reference Class: `ooRef(ooKeyDesc)`

The persistence-capable class `ooKeyDesc` represents a *key description* from which one or more indexes can be created.

See:

- "Reference Summary" on page 374 for an overview of member functions
- "Reference Index" on page 374 for a list of member functions

Applications that use indexes must include the `ooIndex.h` header file.

About Key Descriptions

A key description is a persistent object from which an index is created. Each key description identifies the class of objects to be indexed, the key fields on which to sort, and whether the index is to be unique. The key description is then used to create an index for a particular storage object, sometimes called the *scope* of the index. 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.

You can use a particular key description to create multiple indexes, provided that each index is in a different storage object. Multiple indexes can exist in the same storage object only if each was created from a different key description.

Working With Key Descriptions

As is the case for any basic object, you specify whether a key description is to be transient or persistent when you create it; key descriptions *must be persistent*. You create a key description with a call to the `new` operator; the clustering directive in that call specifies where in the federated database to store the new key description.

Like other persistent objects, key descriptions are normally manipulated through handles or object references. You can store and find a key description in the database just as you would any other persistent object.

Reference Summary

In the following table, operators indicated as *(inherited)* are overloaded in this class with no change in behavior; they are documented with the `ooObj` class (page 431), along with the other inherited member functions not listed here.

Creating and Deleting a Key Description	`ooKeyDesc` `operator new` *(inherited)* `operator delete` *(inherited)*
Working With Key-Field Objects	`addField` `nField`

Creating and Dropping Indexes	`createIndex`
	`dropIndex`
	`removeIndexes`

Getting Information	`getTypeN`
	`getTypeName`
	`isConsistent`
	`isUnique`

Reference Index

dropIndex	Drops the index associated with this key description from the referenced storage object.
getTypeN	Gets the type number of the class that is indexed by this key description.
getTypeName	Gets the name of the class indexed by this key description.
isConsistent	Checks whether the key-field objects in this key description are still consistent with the corresponding data members of the indexed class.
isUnique	Determines whether a particular index was created to be unique. When an index is unique, each of its indexed objects must have unique key field values.
nField	Gets the number of fields in the index.
ooKeyDesc	Constructs a new key description with the specified characteristics.
removeIndexes	Removes from the federated database all indexes that were created from this key description.

Constructors

ooKeyDesc

Constructs a new key description with the specified characteristics.

```
ooKeyDesc(
     const ooTypeNumber typeN,
     const ooBoolean unique = oocFalse);
```

Parameters

typeN

Type number of the persistence-capable class whose objects are to be indexed. This class, and all of its subclasses, will be included in the index.

unique

Specifies whether the indexes created from this key description are to be unique. If you omit this parameter, this key description will create nonunique indexes, in which multiple indexed objects may have the same combination of key field values.

Member Functions

addField

Adds the referenced key-field object to this key description.

```
ooStatus addField(const ooHandle(ooKeyField) &fieldH);
```

Parameters *fieldH*

Handle to the key-field object to be added.

Returns `oocSuccess` if successful; otherwise `oocError`.

Discussion You may add a key-field object to a key description only if both are defined on the same class (the class to be indexed). The key-field object being added must refer to a data member of the indexed class or one of its base classes.

You can add multiple key-field objects to the same description. The order in which key-field objects are added to a key description determines the sorting order of the indexed objects.

The same key-field object can be added to multiple key descriptions.

createIndex

Creates an index for the specified storage object from this key description.

```
1.  ooStatus createIndex(
      const ooHandle(ooContObj) &storageObject);
2.  ooStatus createIndex(
      const ooHandle(ooDBObj) &storageObject);
3.  ooStatus createIndex(
      const ooHandle(ooFDObj) &storageObject);
```

Parameters *storageObject*

Handle to the container, database, or federated database for which the index is to be created. The specified storage object limits the objects that are referenced by the new index—for example, an index created for a container references only the objects of the indexed class that reside in that container.

storageObject may not specify a container if this key description specifies a container class.

Returns `oocSuccess` if successful; otherwise `oocError`.

Discussion You can create indexes for different storage objects from the same key description.

If an index cannot be created, a message is displayed and the transaction is aborted.

dropIndex

Drops the index associated with this key description from the referenced storage object.

```
1.  ooStatus dropIndex(
      const ooHandle(ooContObj) &storageObject);
2.  ooStatus dropIndex(
      const ooHandle(ooDBObj) &storageObject);
3.  ooStatus dropIndex(
      const ooHandle(ooFDObj) &storageObject);
```

Parameters *storageObject*

Handle to the container, database, or federated database from which the index is to be dropped.

Returns `oocSuccess` if successful; otherwise `oocError`.

getTypeN

Gets the type number of the class that is indexed by this key description.

```
ooTypeNumber getTypeN() const;
```

Returns Type number of the indexed class.

getTypeName

Gets the name of the class indexed by this key description.

```
const char *getTypeName() const;
```

Returns String name of the indexed class.

isConsistent

Checks whether the key-field objects in this key description are still consistent with the corresponding data members of the indexed class.

```
ooBoolean isConsistent();
```

Returns Returns `oocTrue` if all the key-field objects in this key description are consistent; returns `oocFalse` if any key-field object in this key description is inconsistent.

Discussion If this member function returns `oocFalse`, you can call the `isConsistent` member function on each key-field object to determine which one is inconsistent.

You should test for consistency after performing a schema-evolution operation that affects an indexed class or a data member of an indexed class.

isUnique

Determines whether a particular index was created to be unique. When an index is unique, each of its indexed objects must have unique key field values.

```
ooBoolean isUnique();
```

Returns `oocTrue` if the target index is unique and `oocFalse` if it is not.

nField

Gets the number of fields in the index.

```
unit32 nField() const;
```

Returns Integer giving the number of fields in the index.

removeIndexes

Removes from the federated database all indexes that were created from this key description.

```
ooStatus removeIndexes();
```

Returns `oocSuccess` if successful; otherwise `oocError`.

Discussion This member function deletes indexes while preserving the key description itself. Alternatively, if you no longer need the key description, you can delete it with the `ooDelete` function, which automatically deletes all its indexes.

ooKeyField Class

Inheritance: **ooObj->ooKeyField**

Handle Class: `ooHandle(ooKeyField)`

Object-Reference Class: `ooRef(ooKeyField)`

The persistence-capable class `ooKeyField` represents a *key field* of an index.

See:

- "Reference Summary" on page 382 for an overview of member functions
- "Reference Index" on page 382 for a list of member functions

Applications that use indexes must include the `ooIndex.h` header file.

About Key-Field Objects

Key-field objects are added to a key description (an instance of `ooKeyDesc`) to define an index's key fields. Each key-field object in a key description corresponds to a particular data member of the indexed class, and designates that data member as a key field of the index. The order in which key field objects are added to a key description determines the order in which the resulting index considers the corresponding key fields when sorting the indexed objects.

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`, and then adding them, in that order, 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. You can create different sorts over the same objects by defining different key descriptions

on the same indexed class and adding the same key-field objects in different orders.

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.

The data member may be C++ `private`, `protected`, or `public`. Its data type must be a numeric, character, Boolean, or string type. In particular:

- 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)

Indexes can optimize tests that compare a key field with a literal numeric or string value. Indexes can optimize tests that compare a key field with a literal numeric or string value. If you define a key field of type `ooBoolean`, an optimized predicate must use an *integer* literal (1 for `oocTrue`, 0 for `oocFalse`).

Working With a Key-Field Object

As is the case for any basic object, you specify whether a key-field object is to be transient or persistent when you create it; key-field objects *must be persistent*. You create a key-field object with a call to the `new` operator; the clustering directive in that call specifies where in the federated database to store the new key-field object.

Like other persistent objects, key-field objects are normally manipulated through handles or object references. You can store and find a key-field object in the database just as you would any other persistent object.

Optimizing String Storage and Lookup

When you create a key-field object for a string-typed data member (for example, `char[]`, `ooString(`*`N`*`)`, `ooVString`, or `ooUtf8String`), you can optimize the space required to store each string key in the index, as well as the processing time

required to access it during lookup. You adjust these characteristics by setting the *ooKeyField* constructor's *fixed* and *maxstrlen* parameters.

Choosing the default values for these parameters (*fixed*=oocFalse and *maxstrlen* =24) causes the index to allocate 24 bytes per indexed object to store a string key for that object (a copy of the string value of the relevant key field). Shorter string keys are padded with null characters. If a string key is longer, only the first 24 bytes are stored in 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 use the *maxstrlen* parameter 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 occupied by null characters.
- If most of the string keys are larger than 24 bytes, you can set *maxstrlen* to a larger value to reduce number of times 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.

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 the length of most strings in the class 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
class Person : public ooObj {
   ooString(8) name;     // Most strings are < 8 bytes long
}
```

```
// Application code
ooHandle(ooKeyField) keyFieldH;
ooHandle(ooKeyDesc) keyDescH;
ooHandle(ooContObj) contH;

keyDescH = new(contH) ooKeyDesc(ooTypeN(Person),oocTrue);
keyFieldH = new(keyDescH) ooKeyField (ooTypeN(Person),"name",
   oocFalse, 8);         // Allows a variable length
```

Reference Summary

In the following table, operators indicated as *(inherited)* are overloaded in this class with no change in behavior; they are documented with the ooObj class (page 431), along with the other inherited member functions not listed here.

Creating and Deleting	ooKeyField operator new *(inherited)* operator delete *(inherited)*
Getting Information	getName getTypeN isConsistent isNamed

Reference Index

isConsistent	Checks whether this key-field object is still consistent with the corresponding data member of the indexed class.
isNamed	Tests whether the data member corresponding to this key-field object has the specified name.
ooKeyField	Constructs a new key-field object with the specified characteristics.

Constructors

ooKeyField

Constructs a new key-field object with the specified characteristics.

```
ooKeyField(
     const ooTypeNumber typeN,
     const char *memberName,
     ooBoolean fixed = oocFalse,
     uint32 maxstrlen = 24);
```

Parameters

typeN

Type number of the indexed class—that is, persistence-capable class whose objects are to be indexed.

memberName

Name of the data member to be represented by this key-field object. The values of this data member will be used to sort the indexed objects.

- 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*"

fixed

Specifies whether the values of this key field are to be treated as fixed or variable in length; applies only if *memberName* specifies a data member whose type is `char[]`, `ooString(`*N*`)`, or `ooVString`.

If you specify `oocFalse` (the default), the index allows string keys of any length:

- If a string key is *maxstrlen* bytes or shorter, it is stored in the index (padded with null characters if necessary).
- If a string key is longer than *maxstrlen* bytes, the first *maxstrlen* bytes are stored in the index for preliminary comparisons during lookup; when the index needs the complete string value, it accesses the indexed object.

If you specify `oocTrue`, the index allows only string keys of length *maxstrlen* or shorter. If the string key for an object is longer than *maxstrlen* bytes, that object is omitted from the index and an error is signalled. If you know the maximum length of the string keys for all objects, specifying `oocTrue` can optimize the index's performance during lookup, because it never has to consult an indexed object to obtain a complete string value.

maxstrlen

Specifies the number of bytes to be allocated by the index for storing a copy of each value of this key field; applies only if *memberName* specifies a data member whose type is `char[]`, `ooString(`*N*`)`, or `ooVString`.

The index allocates *maxstrlen* bytes for each string key. Shorter string keys are padded with null characters. Longer string keys are handled only if *fixed* is set to `oocFalse` (see *fixed* above).

Member Functions

getName

Gets the name of the data member to which this key-field object corresponds.

```
virtual const char *getName() const;
```

Returns String name of a data member of the indexed class.

getTypeN

Gets the type number of the indexed class for which the key-field object was created.

```
ooTypeNumber getTypeN() const;
```

isConsistent

Checks whether this key-field object is still consistent with the corresponding data member of the indexed class.

```
ooBoolean isConsistent();
```

Returns

Returns `oocTrue` if this key-field object is still consistent with the data member with which it was initialized. Returns `oocFalse` if any of the following is true:

- The type of the data member has been changed.
- The data member has been deleted
- The indexed class has been deleted.

Discussion

You should test for consistency after performing a schema-evolution operation that affects an indexed class or a data member of an indexed class.

isNamed

Tests whether the data member corresponding to this key-field object has the specified name.

```
virtual ooBoolean isNamed(const char *name) const;
```

Parameters

name

String name to be matched against the data-member name.

Returns

`oocSuccess` if successful; otherwise `oocError`.

ooLessThanEqualLookupField Class

Inheritance: **ooLookupFieldBase->ooLessThanEqualLookupField**

The non-persistence-capable indexing class `ooLessThanEqualLookupField` represents a *lookup field* that tests whether the value of an indexed object's key field is less than or equal to (<=) the specified comparison value.

See:

- "Reference Index" on page 388 for a list of member functions

Applications that use indexes must include the `ooIndex.h` header file.

About Less-Than-Or-Equal-To Lookup Fields

A lookup field is part of a *lookup key* (an instance of the `ooLookupKey` class that is used for looking up persistent objects in an index). A lookup key consists of one or more lookup fields, each representing a condition that a found object must satisfy. Lookup fields are instances of the concrete classes derived from the abstract base class `ooLookupFieldBase`.

The concrete class `ooLessThanEqualLookupField` represents a condition that tests whether the values of a particular key field are less than or equal to a particular comparison value. You use the class constructor to specify the key field and the value.

For complete information about using lookup fields in lookup keys, see "About Lookup Keys" on page 399

Reference Index

ooLessThanEqualLookupField	Constructs a new lookup field for testing whether values of the specified key field are less than or equal to the specified comparison value.

Constructors

ooLessThanEqualLookupField

Constructs a new lookup field for testing whether values of the specified key field are less than or equal to the specified comparison value.

```
1.  ooLessThanEqualLookupField(
      const ooKeyField &field,
      const void *valuePtr);
2.  ooLessThanEqualLookupField(
      const ooTypeNumber typeN,
      const char *memberName,
      const void *valuePtr);
```

Parameters

field

Key-field object representing a key field of the indexed class. The values of the specified key field will be tested by this lookup field.

typeN

Type number of the indexed class.

memberName

Name of a data member that serves as a key field of the indexed class. The values of the specified key field will be tested by this lookup field.

The specified data member must be defined or inherited by the class specified by *typeN*. You can qualify the name of an inherited data member using the following notation (where *baseClassName* is the name of the base class that defines the inherited data member):

baseClassName::*dataMemberName*

You *must* qualify the name of an inherited data member if the member name is ambiguous (for example, the same name is defined in both the base class and the class to be indexed) or if the member name is not visible due to access control.

valuePtr

Data value to which key-field values are to be compared. The type of this value must match the key-field type (the type of the data member you specified to the key-field object). Using data of any other type may have unpredictable results.

Discussion

This lookup field causes the values of the specified key field to be tested when an iterator scans the index; an indexed object is found if its tested value is less than or equal to the value specified by *valuePtr*.

This lookup field will be ignored if *field* does not specify a key field of the index being searched.

ooLessThanLookupField Class

Inheritance: **ooLookupFieldBase->ooLessThanLookupField**

The non-persistence-capable indexing class `ooLessThanLookupField` represents a *lookup field* that tests whether the value of an indexed object's key field is less than (<) the specified comparison value.

See:

- "Reference Index" on page 392 for a list of member functions

Applications that use indexes must include the `ooIndex.h` header file.

About Less-Than Lookup Fields

A lookup field is part of a *lookup key* (an instance of the `ooLookupKey` class that is used for looking up persistent objects in an index). A lookup key consists of one or more lookup fields, each representing a condition that a found object must satisfy. Lookup fields are instances of the concrete classes derived from the abstract base class `ooLookupFieldBase`.

The concrete class `ooLessThanLookupField` represents a condition that tests whether the values of a particular key field are less than a particular comparison value. You use the class constructor specify the key field and the value.

For complete information about using lookup fields in lookup keys, see "About Lookup Keys" on page 399

Reference Index

Constructors

ooLessThanLookupField

Constructs a new lookup field for testing whether values of the specified key field are less than the specified comparison value.

```
1.  ooLessThanLookupField(
      const ooKeyField &field,
      const void *valuePtr);
2.  ooLessThanLookupField(
      const ooTypeNumber typeN,
      const char *memberName,
      const void *valuePtr);
```

Parameters

field

Key-field object representing a key field of the indexed class. The values of the specified key field will be tested by this lookup field.

typeN

Type number of the indexed class.

memberName

Name of a data member that serves as a key field of the indexed class. The values of the specified key field will be tested by this lookup field.

The specified data member must be defined or inherited by the class specified by *typeN*. You can qualify the name of an inherited data member using the following notation (where *baseClassName* is the name of the base class that defines the inherited data member):

baseClassName::*dataMemberName*

You *must* qualify the name of an inherited data member if the member name is ambiguous (for example, the same name is defined in both the base class and the class to be indexed) or if the member name is not visible due to access control.

valuePtr

Data value to which key-field values are to be compared. The type of this value must match the key-field type (the type of the data member you specified to the key-field object). Using data of any other type may have unpredictable results.

Discussion

This lookup field causes the values of the specified key field to be tested when an iterator scans the index; an indexed object is found if its tested value is less than the value specified by *valuePtr*.

This lookup field will be ignored if *field* does not specify a key field of the index being searched.

ooLookupFieldBase Class

Inheritance: **ooLookupFieldBase**

The non-persistence-capable class ooLookupFieldBase is the abstract base class for classes that represent *lookup fields*.

See:

- "Reference Index" on page 396 for a list of member functions

Applications that use indexes must include the ooIndex.h header file.

About Lookup Fields

A lookup field is part of a *lookup key* (an instance of the ooLookupKey class that is used for looking up persistent objects in an index). A lookup key consists of one or more lookup fields, each representing a condition that a found object must satisfy. When a lookup key contains multiple lookup fields, every found object must satisfy all of the conditions.

The concrete classes derived from the ooLookupFieldBase class each represent a specific condition to be satisfied. These are: ooEqualLookupField, ooGreaterThanEqualLookupField, ooGreaterThanLookupField, ooLessThanEqualLookupField, and ooLessThanLookupField.

Because the ooLookupFieldBase class is abstract, you never instantiate it; instead, you work with instances of its concrete derived classes. You should not create your own subclasses of this class.

Reference Index

Constructors

ooLookupFieldBase

Constructs a new lookup field.

```
ooLookupFieldBase(
     const ooKeyField &field,
     const void *valuePtr,
     ooRelatOp relatOp = oocEQ);
```

Parameters

field

Key-field object representing a key field of the indexed class. The values of the specified key field will be tested by this lookup field.

valuePtr

Data value to which key-field values are to be compared. The type of this value must match the key-field type (the type of the data member you specified to the key-field object). Using data of any other type may have unpredictable results.

relatOp

Relational operator used by this lookup field. This parameter value is supplied by the predefined concrete derived classes.

Member Functions

isNamed

Compares the specified name to the name of the key field whose values will be tested by this lookup field.

```
virtual ooBoolean isNamed(const char *name) const;
```

Returns `oocTrue` if the specified name matches the key-field name; otherwise `oocFalse`.

Parameters *name*

Name to be matched by the key-field name.

ooLookupKey Class

Inheritance: **ooLookupKey**

The non-persistence-capable class `ooLookupKey` represents a *lookup key* used for looking up persistent objects in an index.

See:

- "Reference Summary" on page 405 for an overview of member functions
- "Reference Index" on page 406 for a list of member functions

Applications that use indexes must include the `ooIndex.h` header file.

About Lookup Keys

A lookup key is a transient object for looking up persistent objects in an index. A lookup key consists of one or more *lookup fields*, each representing a condition that the found objects must satisfy. When a lookup key contains multiple lookup fields, every found object must satisfy all of the conditions. Lookup fields are instances of the concrete classes derived from `ooLookupFieldBase`.

You create a lookup key and add lookup fields to it in a transaction. You then use the lookup key to initialize an object iterator to scan a particular storage object. Because lookup keys are transient, they become invalid after you commit the transaction in which they were created.

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 storage object. If no such index exists, no objects are found. In contrast, an object iterator initialized by a predicate scan can find objects in the scanned storage object even if no compatible index exists. An index scan can search an index even if indexes are disabled; disabling indexes affect only predicate scans, not index scans.

Lookup Fields

Every lookup field in a lookup key specifies a relational operator (such as equal or greater than) and a comparison value for testing a particular key field of an index. For example, to search an index of `Person` objects whose key fields are `age` and `weight`, you could create lookup fields that represent the conditions `age = 40` and `weight > 150`.

A lookup key can contain two conditions that apply to the same key field—for example, `age > 20` and `age < 40`. Because the found objects must satisfy all of the conditions (a conjunction), you may not specify constraints that could only be satisfied by a disjunction—for example, `age < 20` and `age > 40`.

Lookup fields are instances of classes derived from `ooLookupFieldBase`. Each such derived class corresponds to a permitted relational operator, as shown in the following table:

Relational Operator	Lookup-Field Class
=	ooEqualLookupField
>	ooGreaterThanLookupField
<	ooLessThanLookupField
>=	ooGreaterThanEqualLookupField
<=	ooLessThanEqualLookupField

As the table illustrates, a lookup field cannot perform string matching or test an application-defined relational operator.

To create a lookup field, instantiate the lookup-field class that corresponds to the desired relational operator. For example, instantiate `ooEqualLookupField` to represent the relational operator `=`.

When you create a lookup field, parameters to the constructor specify the indexed class, the attribute to be tested, and the value to be compared with the attribute value. You can specify both the indexed class and attribute with a key-field object; alternatively, you can specify the class by its type number and the attribute by its name.

Compatible Indexes

A lookup key can search only *compatible indexes*—that is, indexes created from a compatible key description. A lookup key and a key description are compatible only if both of the following are true:

- The lookup key was created for the same indexed class as the key description—that is, the type number specified to the lookup key constructor must match the type number specified to the key description constructor.
- The lookup key contains at least one lookup field that corresponds to the primary key field of the index—that is, at least one lookup field specifies the first key-field object that was added to the key description.

An object iterator initialized with a lookup key finds objects only if they are referenced by a compatible index within the storage object being scanned. If the storage object has no compatible index, the object iterator is initialized to null, so it will find no objects. This is true even if the container, database, or federated database being scanned actually contains objects that would satisfy the lookup key's conditions. You can use the `anyIndex` member function on a lookup key to test whether a given storage object contains a compatible index.

Lookup Fields Used in an Index Scan

When a compatible index is searched by a lookup key with multiple lookup fields, the lookup key uses at least one lookup field during the search (the lookup field corresponding to the index's primary key field). However, the other lookup fields may or may not be used, depending on how they correspond to the index's key fields. In general, a lookup key uses only the lookup fields that correspond to the first *n* consecutive key fields in the index's key description, where *n* is an integer from 1 to the number of key fields in the index. Any other lookup fields are ignored.

For example, assume an index of `Person` objects has key fields `age`, `weight`, and `height`, and you create five lookup keys whose lookup fields correspond to the combinations of key fields listed in the following table. If you then use each lookup key to search the index, certain lookup fields would be used or ignored, as indicated in the table:

Key Fields Referenced by the Lookup Key	Key Fields Tested by the Lookup Key
`age`	`age`; the sole lookup field is used because it corresponds to the primary key field.
`age`, `weight`	`age`, `weight`; both lookup fields are used because they correspond to the first 2 key fields of the index.

Key Fields Referenced by the Lookup Key	Key Fields Tested by the Lookup Key
`age`, `weight`, `height`	`age`, `weight`, `height`; all lookup fields are used because they correspond to the first 3 key fields of the index.
`age`, `height`	`age`; the lookup field corresponding to `height` is ignored, because `height` is the index's third key field, and the second key field (`weight`) is not represented in the lookup key.
`weight`, `height`	None; both lookup fields are ignored because the primary key field (`age`) is not represented in the lookup key.

A lookup key ignores a lookup field that does not correspond to any key field of the index being searched. For example, assume you want to search an index of `Person` objects with key fields `age`, `weight`, and `height`. If you search the index using a lookup key that has fields corresponding to `age`, `weight`, and `eyecolor`, the lookup field for testing `eyecolor` is ignored.

WARNING If lookup fields are ignored by a particular scan operation, the found objects are not guaranteed to match the entire condition represented by the lookup key. Instead, they match a conjunction of the conditions represented by the lookup fields that were tested.

Whenever a lookup field is ignored, the application is responsible for explicitly testing the appropriate data member of each found object. To avoid confusion, you should use a lookup key only if you know that a compatible index exists in which *all* lookup fields can be tested.

Finding Indexed Objects With a Lookup Key

To use a lookup key to find an object in a particular container, database, or federated database:

1. Create a lookup key by calling the `ooLookupKey` class constructor within a transaction.
2. Create a lookup field for each condition to be satisfied by the found objects. You must create at least one lookup field that tests the primary key field of the indexed objects. To create a lookup field:
   a. Choose the lookup-field class that corresponds to the desired relational operator. For example, choose `ooLessThanLookupField` to represent the relational operator <=.

b. Invoke the constructor of the chosen class, specifying the desired comparison value and the key field to be tested.

Note: You specify a range of values for testing a single key field by creating two lookup fields. For example, to find indexed objects whose `age` key field is between 15 and 30, you create lookup fields that express the conditions `age >= 15` and `age <= 30`.

3. Add each lookup field to the lookup key by calling the `addField` member function on the lookup key. **Note:** The order in which lookup fields are added determines the order in which key fields are tested. For optimal performance:
   - Keep the number of lookup fields small.
   - Add lookup fields to the lookup key in the same order in which the corresponding key fields were added to the key description, starting with the most major key field.
   - Specify as many equality (=) comparisons as possible. If you are adding multiple lookup fields, only the last one should represent a relational operator other than =.
4. Test whether the lookup key is compatible with any indexes in the storage object you plan to search. To do this, call the `anyIndex` member function on the lookup key.
5. Use the lookup key to initialize an object iterator of type `ooItr(`*`className`*`)`, where *`className`* is the name of the indexed class. To initialize the object iterator, call the `scan` member function on it, specifying the lookup key and the container, database, or federated database to be searched.

If no compatible index exists in the specified storage object, the object iterator is initialized with a null iteration set. If a compatible index exists, the object iterator is initialized to find objects satisfying the applicable lookup fields, in ascending order.

The lookup key and its lookup fields must exist (remain in scope) while you are initializing and advancing the object iterator; if they have been destructed, the iteration will fail.

EXAMPLE Assume an index is created from a key description whose key-field objects were added in the following order: `plantNumber`, `deptNumber`, `groupNumber`, `projLeaderId`.

A lookup key whose lookup fields represent the following conditions would produce a fast search, because they match the key-field order and because only the last condition is not an equality comparison.

```
plantNumber = 1
deptNumber = 6
groupNumber > 15
```

The following search key looks for groups with group numbers ranging from 15 to 30:

```
plantNumber = 1
deptNumber = 6
groupNumber >= 15
groupNumber <= 30
```

EXAMPLE This example defines the persistence-capable class `Point` in a DDL file. The application code creates an index of `Point` objects with `xCoord` and `yCoord` as the primary and secondary key fields.

```
// DDL file
class Point : public ooObj {
public:
   char name[32];
   int32 xCoord;
   int32 yCoord;
};
```

```
// Application code
ooHandle(ooKeyField) keyFieldH;
ooHandle(ooKeyDesc) keyDescH;
ooHandle(ooContObj) contH;
...                 // Set contH to the desired container.

// Create the key description in the container.
keyDescH = new(contH) ooKeyDesc(ooTypeN(Point), oocTrue);

// Define the primary key field; add it to the key description.
keyFieldH = new(keyDescH) ooKeyField(ooTypeN(Point), "xCoord");
keyDescH->addField(keyFieldH);

// Define the secondary key field; add it to the key description.
keyFieldH = new(keyDescH) ooKeyField(ooTypeN(Point), "yCoord");
keyDescH->addField(keyFieldH);

// Create an index in the container.
ooStatus status = keyDescH->createIndex(contH);
```

Another transaction (or a different program) creates a lookup key to search for indexed `Point` objects that have an x-coordinate equal to 100 and a y-coordinate

greater than 50. The lookup key is used to initialize an object iterator to find such points.

```
// Application code
ooHandle(ooContObj) contH;
ooItr(Point) pointI;
const int xPoint = 100;      // x == 100
const int yMinPoint = 50;    // y > 50
...           // Set contH to the container containing the index.

// Create a lookup key.
ooLookupKey lookupKey(ooTypeN(Point), 2);

// Create the lookup fields and add them to the lookup key.
ooEqualLookupField xLookupField(
   ooTypeN(Point), "xCoord", &xPoint);
lookupKey.addField(xLookupField);

ooGreaterThanLookupField yLookupField(
   ooTypeN(Point), "yCoord", &yMinPoint);
lookupKey.addField(yLookupField);

// Initialize the object iterator by scanning the container
// with the lookup key.
if (pointI.scan(contH, lookupKey)) {
   // Advance the object iterator
   while (pointI.next()) {
      cout  <<    "Found point at (" << pointI->xCoord
            <<    ", " << pointI->yCoord << ")." << endl;
   }
}
```

Reference Summary

Creating	ooLookupKey
Modifying	addField
Getting Information	anyIndex nField

Reference Index

Constructors

ooLookupKey

Constructs a new lookup key with the specified characteristics.

```
ooLookupKey(
     const ooTypeNumber typeN,
     const uint32 number);
```

Parameters

typeN

Type number of the persistence-capable class of indexed objects. For the lookup key to be valid and usable, *typeN* must match exactly the *typeN* that was used in creating the key description.

number

Number of lookup fields you intend to add to the lookup key. You cannot add more fields to the lookup key than specified by *number*, but you can add fewer fields. This number cannot be changed after the lookup key is created. Every lookup field counts separately, even if multiple lookup fields test the values of the same key field.

Member Functions

addField

Adds the specified lookup field to this lookup key.

```
ooStatus addField(const ooLookupFieldBase &field);
```

Parameters *field*

Lookup field to be added.

Returns `oocSuccess`, if the lookup field is added successfully; otherwise, `oocError` if the lookup key is full.

Discussion You may add a lookup field to a lookup key only if both are defined on the same class.

You can add multiple lookup fields to the same lookup key.

The order in which lookup fields are added to a lookup key affects the performance of the lookup. For optimal performance, you should add lookup fields to the lookup key in the same order in which the corresponding key fields were added to the key description, starting with the primary key field.

The same lookup field can be added to multiple lookup keys.

anyIndex

Tests whether this lookup key is compatible with one or more indexes in the specified storage object.

```
1.  ooBoolean anyIndex(
      const ooHandle(ooContObj) &storageObject) const;
2.  ooBoolean anyIndex(
      const ooHandle(ooDBObj) &storageObject) const;
3.  ooBoolean anyIndex(
      const ooHandle(ooFDObj) &storageObject) const;
```

Parameters *storageObject*

Handle to the container, database, or federated database you want to test for a compatible index. An error is reported if any other type of handle is specified.

Returns `oocTrue` if there is a compatible index, `oocFalse` if there is no compatible index.

Discussion If this member function returns `oocTrue`, then you can use `scan` on an object iterator to look up objects using an index.

The following table shows the return values for various combinations of indexed key fields (x, y, and z) and corresponding lookup fields.

Key Fields	Lookup Fields	anyIndex Return Value
x	x	oocTrue
x	y	oocFalse
x, y	x	oocTrue
x, y	x, y	oocTrue
x, y	y	oocFalse
x, y, z	x	oocTrue
x, y, z	x, y	oocTrue
x, y, z	x, y, z	oocTrue
x, y, z	x, z	oocTrue (but only the lookup field for x will be used; the application must test z values explicitly)

nField

Gets the number of lookup fields that have been added to this lookup key.

```
uint32 nField() const;
```

Returns An integer number of lookup fields.

ooMap Class

Inheritance: **ooObj->ooMap**

Handle Class: `ooHandle(ooMap)`

Object-Reference Class: `ooRef(ooMap)`

The persistence-capable class `ooMap` represents unordered *name maps*.

See:

- "Reference Summary" on page 412 for an overview of member functions
- "Reference Index" on page 413 for a list of member functions

To use this class, your application must include the `ooMap.h` header file. No extra linking is required.

About Name Maps

A name map is a collection whose elements (instances of `ooMapElem`) are key-value pairs, where each key is a string and each value is an object reference to a persistent object. No two elements of the name map may have the same key.

Each element of a name map defines a mapping from a name to an object with that name. The word *name* is used as a synonym for *key* because an object's key is often a name; however, a key can be any valid C++ string that identifies the object. Keys can be strings of any length.

A name map provides an efficient way to assign identifying keys to objects and to look up objects by their keys. Objectivity / DB uses name maps to implement its dictionaries of scope-named objects. In addition, you can instantiate this class to create your own application-specific dictionary for objects.

Growth Characteristics

A name map is implemented as a hash table using a traditional hashing mechanism. The hash table can grow dynamically; however, increasing its size requires rehashing the entire hash table.

When you create a name map, you specify:

- The *initial number of bins* (hash buckets). For optimal performance, the number of bins should always be a prime number.
- The *maximum average density*, that is, the average number of elements per bin 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 bins is increased by the growth factor, then rounded up to the nearest prime number.

Performance Considerations

Name maps are good for looking up one object at a time. They give better performance than indexes on data involving frequent updates. Indexes perform better on read-only data.

To avoid rehashing the name map, you need a good estimate of the initial number of hash bins. Also, you should put the name map in a container by itself (its `ooMapElem` elements will automatically be clustered with it). Clustering other objects with a name map and its elements will result in poor performance of the name map.

Runtime Statistics

Name maps accumulate statistical information about runtime usage. You can print a summary of this information or clear the statistical parameters, resetting them to zero. The statistical parameters record:

- The number of elements added.
- The number of elements deleted.
- The number of times the hash table was rehashed.

Referential Integrity

By default, when you add an object to a name map, Objectivity/C++ automatically creates a unidirectional association from the added object to the name-map element that references it. Because this association has delete propagation enabled, deleting the object automatically causes the name-map

element to be deleted as well, preserving the referential integrity of 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.

Hash Function

For efficient lookup, elements of name maps are stored in a hash table; hash values are computed from the key of each element. You can define your own hashing function if you desire.

A hash 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.

All name maps that the application accesses use the same hash function. To install an application-defined hash function for name maps, call the static member function `ooMap::set_nameHashFunction`.

EXAMPLE This example shows how to install the function `myNameHash` as the hash function for all name maps in the current application.

```
uint32 myNameHash(const char *name, const uint32 modulus)
{
// This function should return a value between 0 and modulus-1
   …
}
// Set the name hash function used in name map
// to be myNameHash

ooMap::set_nameHashFunction(myNameHash);
```

Working With a Name Map

As is the case for any basic object, you specify whether a name map is to be transient or persistent when you create it; name maps *must be persistent*. You create a name map with a call to the `new` operator; the clustering directive in that call specifies where in the federated database to store the new name map.

Like other persistent objects, name maps are normally manipulated through handles or object references. You can store and find an unordered name map in the database just as you would any other persistent object.

Related Classes

The class `ooMapElem` represents an element in a name map. Each element is a key-value pair that associates a name with a persistent object. The key is a C++ string and the value is an object reference to a persistent object.

The class `ooMapItr` defines an iterator for name maps; you can use an iterator to obtain each element of the name map.

Two additional classes represent persistent collections of key-value pairs.

- `ooHashMap` represents an unordered *object map*, that is, a collection of key-value pairs in which *both the key and the value are persistent objects*. It uses an extendible hashing mechanism that allows the hash table to grow without being completely rehashed.
- `ooTreeMap` represents a *sorted object map*.

Reference Summary

In the following table, operators indicated as *(inherited)* are overloaded in this class with no change in behavior; they are documented with the `ooObj` class (page 431), along with the other inherited member functions not listed here.

Creating and Deleting	`ooMap` `operator new` *(inherited)* `operator delete` *(inherited)*

Adding and Removing Elements	`add`
	`forceAdd`
	`remove`
	`replace`

Finding Objects	`lookup`

Getting Information	maxAvgDensity nameHashFunction nBin nElement percentGrow printStat
Testing	isMember refEnable
Working With Runtime Statistics	clearParam printStat
Maintaining the Hash Table	refEnable rehash set_nameHashFunction set_refEnable
Static Utilities	set_nameHashFunction nameHashFunction

Reference Index

nElement	Returns the total number of elements in this name map's hash table.
ooMap	Constructs a new name map with the specified growth characteristics.
percentGrow	Returns the percentage by which this name map's hash table grows when it is resized.
printStat	Prints runtime statistical information about this name map to the specified file.
refEnable	Tests whether Objectivity/DB automatically maintains the referential integrity of this name map.
rehash	Resizes this name map's hash table.
remove	Removes the specified element from this name map.
replace	Replaces the object associated with the specified name in this name map.
set_nameHashFunction	Sets the hash function for all instances of ooMap in the federated database.
set_refEnable	Enables or disables maintenance of referential integrity for this name map.

Constructors

ooMap

Constructs a new name map with the specified growth characteristics.

```
ooMap(
     const uint32 nBin = oocMapInitHashBinSize,
     const uint32 maxAvgDensity = oocMapMaxAvgDensity,
     const uint32 percentGrowth = oocMapPercentGrow);
```

Parameters

nBin

Initial number of bins in the new name map's hash table. The default value is 11.

maxAvgDensity

Average number of elements per bin allowed before hash table is resized. For example, if the total number of elements in the table is greater than or equal

to `nBins *maxAvgDensity`, the table is resized. The default value of this parameter is 5.

`percentGrowth`

Parameter (expressed as percent) used to resize the hash table. The default value is 100. This growth factor is used to multiply the original bin size to get the new bin size. The system rounds the number of bins after rehashing to a prime number.

Member Functions

add

Adds a new element to this name map.

```
ooStatus add(
     const char *name,
     const ooRef(ooObj) &objR,
     ooRefHandle(ooMapElem) &elem = oocDefaultMapElemHandle);
```

Parameters

`name`

String name; specifies the key in the added element.

`objR`

Object reference to a persistent object; specifies the value in the added element.

`elem`

Object reference or handle to set to the new element. By default, *`elem`* is a null map-element object reference or handle.

Returns

`oocSuccess` if successful; otherwise `oocError`.

Discussion

If you specify *`elem`*, you can use the initialized object reference or handle to perform operations such as setting a unidirectional association to the new element.

See also

`forceAdd`
`remove`

clearParam

Resets this name map's statistical parameters to zero.

```
ooStatus clearParam();
```

Returns `oocSuccess` if successful; otherwise `oocError`.

forceAdd

Forces the addition of an element to the name map without checking for the prior existence of an element with the same name.

```
ooStatus forceAdd(
     const char *name,
     const ooRef(ooObj) &objR,
     ooRefHandle(ooMapElem) &elem = oocDefaultMapElemHandle);
```

Parameters

name

String name; specifies the key in the added element.

objR

Object reference to a persistent object; specifies the value in the added element.

elem

Object reference or handle to set to the new element. The default is a null reference to a map-element handle.

Returns `oocSuccess` if successful; otherwise `oocError`.

Discussion

If there is already an element with this name in the table, two elements will contain the same name after a successful `forceAdd` operation. The element that is found by `lookup` is undefined. You should only use this member function if you know there is no element with the same name.

If you specify *elem*, you can use the initialized object reference or handle to perform operations such as setting a unidirectional association to the new element.

See also add

isMember

Tests whether any element in the name map contains the key *name*.

```
ooBoolean isMember(const char *name);
```

Parameters *name*

Name to be checked.

Returns `oocTrue` if any element contains the name, otherwise `oocFalse`.

lookup

Looks up the specified name in this name map and returns either the corresponding object or the name-map element.

```
1.  ooRef(ooObj) lookup(
      const char *name);
2.  ooStatus lookup(
      const char *name,
      ooRefHandle(ooObj) &object,
      const ooMode openMode = oocRead);
3.  ooStatus lookup(
      const char *name,
      ooRefHandle(ooMapElem) &elem);
```

Parameters *name*

Name (key) of the object to be looked up.

object

Object reference or handle to set to the found object. *object* is set to null if the specified name is not found or if an error occurs.

openMode

Mode in which to open the found object. Specify `oocNoOpen` to prevent the object from being opened.

elem

Object reference or handle to set to the found name-map element.

Returns (Variant 1) A type-independent object reference to the found object; you must cast the returned object reference to the appropriate type.

(Variants 2 and 3) `oocSuccess` if successful; otherwise `oocError`.

maxAvgDensity

Returns the allowable maximum average density of this name map's hash table.

```
uint32 maxAvgDensity() const;
```

nameHashFunction

Gets the hash function used by all name maps in the federated database.

```
static ooNameHashFuncPtr nameHashFunction();
```

Returns A pointer to the hash function currently in use.

See also set_nameHashFunction

nBin

Returns the number of bins in this name map's hash table.

```
uint32 nBin() const;
```

nElement

Returns the total number of elements in this name map's hash table.

```
uint32 nElement() const;
```

percentGrow

Returns the percentage by which this name map's hash table grows when it is resized.

```
uint32 percentGrow() const;
```

Discussion 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.

printStat

Prints runtime statistical information about this name map to the specified file.

```
void printStat(FILE *outFile = stdout) const;
```

Parameters *outfile*

Name of the output file.

Discussion The output from this member function is similar to:

```
Run statistics of ooMap #2-3-3-3 (Fri Aug 25 15:22:45 PDT 1992)

** Number of elements added              => 0
** Number of elements removed            => 0
** Number of rehashes                    => 0
```

```
** Current state:
      Number of bins                    => 23
      Number of elements                => 100
      Maximum average density           => 5
      Percent Growth                    => 100
      Average length per bin            => 4.34783
      Maximum length                    => 7
      Standard deviation of length      => 1.08783
      Maintain referential integrity    => Yes
```

refEnable

Tests whether Objectivity / DB automatically maintains the referential integrity of this name map.

```
ooBoolean refEnable() const;
```

Returns `oocTrue` if referential integrity is being maintained automatically; otherwise `oocFalse`.

Discussion By default, Objectivity / DB maintains the referential integrity of name maps. You can disable the automatic maintenance of referential integrity for a particular name map by calling its `set_refEnable` member function.

See also `set_refEnable`

rehash

Resizes this name map's hash table.

```
ooStatus rehash(
     const uint32 binSize = oocInitMapHashBinSize);
```

Parameters *`binSize`*

New number of bins in the hash table. If this parameter is 0, the default number of bins (11) is used. For optimal performance, the number of bins should be a prime number.

Returns `oocSuccess` if successful; otherwise `oocError`.

Discussion Objectivity / DB automatically increases the size of the hash table as necessary. The only time you should need to call this member function is when the number of bins is very large and the total number of elements is relatively small. This situation can occur when many elements are deleted after the hash table has grown to its peak size.

remove

Removes the specified element from this name map.

```
1.  ooStatus remove(const char *name);
2.  ooStatus remove(const ooHandle(ooMapElem) &objH);
```

Parameters

name

Name of the element to be removed.

objH

Handle to the element to be removed.

Returns

`oocSuccess` if successful; otherwise `oocError`. This function returns `oocSuccess` if the specified element does not exist.

Discussion

If `forceAdd` was used to add more than one element with the same name, `remove` only deletes the first such element it finds and gives no indication that there are other elements with the same name.

See also

add

replace

Replaces the object associated with the specified name in this name map.

```
ooStatus replace(
     const char *name,
     const ooRef(ooObj) &objR,
     ooRefHandle(ooMapElem) &elem = oocDefaultMapElemHandle);
```

Parameters

name

Name of the element to be replaced.

objR

Object reference of the referenced element.

elem

Object reference or handle to set to the replaced element. The default is a null reference to a map-element handle.

Returns

`oocSuccess` if successful; otherwise `oocError`.

Discussion

If this name map contains an element with the key *name*, this member function replaces that element's value with the specified object reference. Otherwise, this

member function adds a new element whose key is *name* and whose value is the specified object reference.

If *elem* is given, it is initialized to reference the element whose key is *name*. You can use this initialized object reference or handle to perform operations such as setting a unidirectional association to the new element.

set_nameHashFunction

Sets the hash function for all instances of ooMap in the federated database.

```
static void set_nameHashFunction(
     ooNameHashFuncPtr hashFunction);
```

Parameters

hashFunction

Function pointer to the new hash function.

Discussion

After a new hash function is set, existing name maps must be rehashed before they can be accessed.

The hash function is set as an attribute of the ooMap class and so is not stored as part of any persistent instance. Consequently, the same hash function must be set by every application that is to access a name map in the federated database.

See also

nameHashFunction
ooNameHashFuncPtr global type

set_refEnable

Enables or disables maintenance of referential integrity for this name map.

```
ooStatus set_refEnable(ooBoolean refEnable = oocTrue);
```

Parameters

refEnable

Specifies whether to enable (oocTrue) or disable (oocFalse) referential integrity.

Returns

oocSuccess if successful; otherwise oocError.

Discussion

By default, Objectivity / DB maintains the referential integrity of name maps. You may disable referential integrity to reduce the overhead of adding and deleting elements. However, if you disable referential integrity, you are responsible for deleting elements from the name map when the corresponding objects are deleted from the database.

If you call this member function, you must call it before any elements are added to this name map.

See also `refEnable`

ooMapElem Class

Inheritance: **ooObj->ooMapElem**

Handle Class: `ooHandle(ooMapElem)`

Object-Reference Class: `ooRef(ooMapElem)`

The persistence-capable class `ooMapElem` represents individual elements in a name map (an instance of `ooMap`).

See:

- "Reference Summary" on page 424 for an overview of member functions
- "Reference Index" on page 424 for a list of member functions

To use this class, your application must include the `ooMap.h` header file. No extra linking is required.

About Name-Map Elements

Each element of a name map is a key-value pair that associates a name with a persistent object. The key is a C++ string and the value is an object reference to a persistent object. The `ooMapElem` class has member functions to get the key and value from a name-map element and to replace the value in a name-map element.

Working With a Name-Map Element

The `ooMapElem` class is transparent to applications in most cases; the class `ooMap` provides the interface to managing a name map's elements. Thus, instances of the `ooMapElem` class are not created or deleted by an application directly, but

through calls to a name map's add and remove member functions. You can obtain an existing name-map element from the name map's lookup member function.

Like other persistent objects, name-map elements are manipulated through handles and object references. The iterator class ooMapItr iterates through the name-map elements in a name map.

Reference Summary

Getting a Key or Value	name oid
Setting a Value	set_oid

Reference Index

Member Functions

name

Gets the key from this name-map element.

```
const char *name() const;
```

Returns The string that is the key in this name-map element.

oid

Gets the value from this name-map element.

```
ooRef(ooObj) oid() const;
```

Returns The object reference that is the value in this name-map element.

Discussion This member function returns a type-independent object reference, which you must cast to the appropriate type.

set_oid

Sets the value in this name-map element.

```
ooStatus set_oid(const ooRef(ooObj) &objR);
```

Parameters *objR*

The object reference that is to be the new value in this name-map element.

Returns `oocSuccess` if successful; otherwise `oocError`.

ooMapItr Class

Inheritance: **ooHandle(ooMapElem)->ooItr(ooMapElem)->ooMapItr**

The non-persistence-capable class `ooMapItr` defines a *name-map iterator*—that is, an iterator for finding the elements of a name map.

See:

- "Reference Summary" on page 427 for an overview of member functions
- "Reference Index" on page 428 for a list of member functions

To use this class, your application must include the `ooMap.h` header file. No extra linking is required.

About Name-Map Iterators

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 iterators with the `ooMapItr` constructor, passing a handle to the name map as the parameter.
- Initialize a name-map iterator with the assignment operator (=), specifying a handle to the name map as right-hand operand.

Reference Summary

Initializing	`ooMapItr` `operator=`
Advancing	`next`

Reference Index

ooMapItr	Default constructor that constructs a new uninitialized iterator for finding elements in a name map.
ooMapItr	Constructs a new name-map iterator and initializes it with the specified name map.
operator=	Assignment operator; initializes this name-map iterator with the referenced map.
next	Advances this name-map iterator to the next name-map element.

Constructors

ooMapItr

Default constructor that constructs a new uninitialized iterator for finding elements in a name map.

```
ooMapItr();
```

ooMapItr

Constructs a new name-map iterator and initializes it with the specified name map.

```
ooMapItr(const ooRefHandle(ooMap) &map);
```

Parameters

map

Object reference or handle to the map with which to initialize the name-map iterator.

Discussion

The name-map iterator is initialized so that the `next` member function will open each name-map element for read-only access.

Operators

operator=

Assignment operator; initializes this name-map iterator with the referenced map.

```
ooMapItr &operator=(ooRefHandle(ooMap) &map);
```

Parameters

map

Object reference or handle to the map with which to initialize the name-map iterator.

Discussion

The name-map iterator is initialized so that the `next` member function will open each name-map element for read-only access.

Member Functions

next

Advances this name-map iterator to the next name-map element.

```
ooBoolean next();
```

Returns

`oocTrue` if another element is found in the map; `oocFalse` if all of the elements in the map have been found or if an error occurred.

Discussion

This member function finds each successive name-map element in a map. To access the object to which a name-map element refers, you must use the `oid` member function on the name-map element.

This member function opens each found name-map element for read only (`oocRead`), so you cannot modify a found name-map element during an iteration. You can, however, modify an object obtained from a name-map element if you first explicitly open the object for update.

You must not modify the map during an iteration—for example, by adding or deleting name-map elements. This means that, if referential integrity is enabled, you must not delete any object that is referenced by a name-map element, because the deletion will propagate to the name-map element.

ooObj Class

Inheritance: **ooObj**

Handle Class: ooHandle(ooObj)

Object-Reference Class: ooRef(ooObj)

The persistence-capable class ooObj is the base class for all classes of Objectivity / DB *objects*. The ooObj class and its corresponding handle and object-reference classes together define persistence behavior for various kinds of Objectivity / DB objects.

See:

- "Reference Summary" on page 435 for an overview of ooObj member functions
- "Reference Index" on page 436 for a list of ooObj member functions

For operations performed through a handle or object reference, see:

- "Reference Summary" on page 601

(*ODMG*) The ooObj class is equivalent to the ODMG standard class d_Object. You can use the name d_Object interchangeably with ooObj.

About Objectivity/DB Objects

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

- *Basic objects*, which are the fundamental units stored by Objectivity / DB. An application defines persistence-capable classes for basic objects by deriving them from ooObj.

- *Persistent objects*, which include basic objects and containers. An application defines its own persistence-capable classes for containers by deriving them from `ooContObj`.
- *Storage objects*, which include standard containers, databases, and federated databases. These are instances of `ooContObj`, `ooDBObj`, and `ooFDObj`, respectively.
- Autonomous partitions, which partition storage objects into independent units that continue to function even if separated by network or system failures. Autonomous partitions are instances of `ooAPObj`.

Because `ooObj` is the abstract base class for all Objectivity / DB objects, you can use handles and object references of type *`ooRefHandle`*`(ooObj)` to reference any kind of Objectivity / DB object.

Because of the member functions it defines, however, `ooObj` should also be thought of in two more specific roles:

- As the base class for basic objects. Most of the member functions defined by `ooObj` apply *only* to basic objects—for example, member functions that support copying, moving, and versioning.
- As the base class for persistent objects. Member functions that apply to *both* basic objects and containers include `operator delete` and member functions that enable you to operate on "this" object within a member function you are defining on a derived persistence-capable class.

None of the member functions defined by `ooObj` apply to databases, federated databases, or autonomous partitions.

Working With Basic Objects

NOTE This section focuses on the role of `ooObj` as the base class for basic objects. See the `ooContObj`, `ooDBObj`, `ooFDObj`, and `ooAPObj` classes for information about working with containers, databases, federated databases, and autonomous partitions, respectively.

A basic object is an instance of any application-defined class that is derived from `ooObj` (but not through `ooContObj`). Like any persistence-capable class, an application-defined basic-object class must be defined in a DDL file and its definition must be processed by the DDL processor.

An application can create instances of `ooObj`, although such instances are of limited use because they can contain no application-specific data. However,

instances of ooObj can be useful for populating test databases—for example, to estimate file size, establish performance limits, and so on.

Basic objects are created and deleted from within an application. An application:

- Creates a basic object using the constructor and operator new of the appropriate basic-object class—namely, ooObj or an application-defined class *appClass* derived from ooObj.
- Deletes a basic object using either the ooDelete global function or the operator delete defined by ooObj.

As is the case for any persistent object, you specify whether a basic object is to be transient or persistent when you create it. A clustering directive on operator new specifies where to locate a new persistent basic object in the federated database.

To work with a new basic object, an application must assign the result of operator new to a handle—for example, an instance of ooHandle(ooObj). A new basic object of an application-defined class *appClass* is normally assigned to an instance of the type-specific handle class ooHandle(*appClass*).

Similarly, to identify and work with an existing basic object, the application must open it through an appropriate handle or object reference; multiple handles and object references can be set to reference the same basic object. The application then operates on an open basic object by:

- Calling various member functions on any of the referencing handles or object references.
- Passing any of the referencing handles or object references to various global functions or member functions of other classes.

The general handle and object-reference classes *ooRefHandle*(ooObj) provide the primary interface for operating on existing basic objects, with member functions for opening, locking, getting information, moving, copying, versioning, and finding referenced objects. The type-specific handle and object-reference classes *ooRefHandle*(*appClass*) inherit these member functions from *ooRefHandle*(ooObj), redefining them wherever type-specific behavior or parameters are required.

In addition, every application-defined basic-object class *appClass* has:

- DDL-generated member functions for creating, deleting, and accessing any associations defined by the class.
- Member functions defined by ooObj (see “Reference Summary” on page 435), which are inherited or redefined by *appClass*.
- Data members and member functions specific to *appClass*.

You can call these members directly from within a member function of *appClass* or indirectly through the indirect member-access operator (->) on a handle or object reference of type *ooRefHandle*(*appClass*).

Support for Versioning Basic Objects

The ooObj class defines the following bidirectional associations that support the versioning of basic objects:

```
class ooObj {
   ...
   // Links a previous version with one or more next versions.
   ooRef(ooObj) nextVers[] <-> prevVers ;
   ooRef(ooObj) prevVers <-> nextVers[] ;

   // Links the default version with its genealogy.
   ooRef(ooGeneObj) defaultToGeneObj <-> defaultVers ;

   // Links a nondefault version with its genealogy.
   ooRef(ooGeneObj) geneObj <-> allVers[]  : version(copy);

   // Links a derived version with the ancestor from which it
   // is derived; represents merged version branches.
   ooRef(ooObj) derivatives[] <-> derivedFrom[] ;
   ooRef(ooObj) derivedFrom[] <-> derivatives[] ;
};
```

You enable versioning on an object by calling setVersStatus on a handle to the object. When versioning is enabled, opening the object for update creates a new *version* of it. A version is a bit-wise copy of the object that has its own object identifier; Objectivity/DB automatically links each new version with its previous version by setting their prevVers and nextVers associations. When linear versioning is enabled, a previous version can have at most one next version; when branch versioning is enabled, a previous version can have multiple next versions (consequently, the nextVers association is to-many).

A basic interface for versioning is provided by the object-reference and handle classes *ooRefHandle*(ooObj). For example, if you have a handle to an object for which versioning is enabled, you can use the handle to find other versions of the referenced object; you can set up a genealogy of versions with the referenced object as the default version; or you can find the default version (if one exists) for a particular referenced object. This basic interface automatically manages the prevVers, nextVers, defaultToGeneObj, and geneObj associations.

The ooObj class supports application-defined versioning semantics by making public the member functions that create, delete, and access the versioning

associations. These functions correspond to the standard set of functions generated for each association *linkName* defined in a persistence-capable class *appClass* (see page 81).

While Objectivity / DB continues to manage the prevVers and nextVers associations whenever new versions are created, you can use the provided functions to:

- Maintain a sequence of next and previous versions in a linear genealogy—for example, you can reset the nextVers and prevVers association to connect adjacent versions after a version is deleted.
- Manage the branches of a branched genealogy—for example, you can set the derivatives and derivedFrom associations to connect the most recent versions of two different branches, merging those branches.
- Set up a custom genealogy—for example, you can derive a genealogy class from the standard ooGeneObj class, and then use the defaultToGeneObj and geneObj associations to maintain a default version for the custom genealogy.

Reference Summary

Creating and Deleting Persistent Objects	ooObj operator new operator delete
Working With This Persistent Object	ooGetTypeN ooGetTypeName ooIsKindOf ooThis ooUpdate ooValidate
Customizing Behavior for Basic Objects	ooCopyInit ooPreMoveInit ooPostMoveInit ooNewVersInit

Customized Versioning of Basic Objects—Previous and Next Versions	nextVers exist_nextVers add_nextVers sub_nextVers del_nextVers prevVers exist_prevVers set_prevVers del_prevVers
Customized Versioning of Basic Objects—Derived Versions	derivedFrom exist_derivedFrom add_derivedFrom sub_derivedFrom del_derivedFrom derivatives exist_derivatives add_derivatives sub_derivatives del_derivatives
Customized Versioning of Basic Objects—Default Versions	geneObj exist_geneObj set_geneObj del_geneObj defaultToGeneObj exist_defaultToGeneObj set_defaultToGeneObj del_defaultToGeneObj
ODMG Interface	operator new mark_modified

Reference Index

add_derivatives	Links this object with the specified object, making the specified object a derivative (successor version) of this object.
add_derivedFrom	Links this object with the specified object, making the specified object an ancestor version of this object.

add_nextVers	Links this object with the specified object, making the specified object a next version of this object.
defaultToGeneObj	Finds, and optionally opens, the genealogy associated with this object, if this object is the default version in a genealogy.
del_defaultToGeneObj	Deletes any defaultToGeneObj association from this object, indicating this object is no longer the default version.
del_derivatives	Deletes all derivatives associations from this object, indicating this object has no derivatives (successor versions).
del_derivedFrom	Deletes all derivedFrom associations from this object, indicating this object is not derived from any ancestor versions.
del_geneObj	Deletes any geneObj association that exists for this object, indicating this object is no longer in a genealogy.
del_nextVers	Deletes all nextVers associations that exist for this object, indicating that this object has no next versions.
del_prevVers	Deletes any prevVers association that exists for this object, indicating this object has no previous version.
derivatives	Initializes an object iterator to find, and optionally open, all versions that are derived from this object and that satisfy any specified selection criteria.
derivedFrom	Initializes an object iterator to find, and optionally open, all ancestor versions from which this object is derived, and that satisfy any specified selection criteria.
exist_defaultToGeneObj	Tests whether this object is the default version in any genealogy or, if a parameter is given, in the specified genealogy.
exist_derivatives	Tests whether this object has any derivatives (successor versions); if a parameter is given, tests whether the specified object is a derivative of this object.
exist_derivedFrom	Tests whether this object is derived from any other objects (ancestor versions), or, if a parameter is given, from the specified object.

exist_geneObj	Tests whether this object is a version within any genealogy, or, if a parameter is given, within the specified genealogy.
exist_nextVers	Tests whether this object has a next version; if a parameter is given, tests whether the specified object is the next version.
exist_prevVers	Tests whether this object has a previous version; if a parameter is given, tests whether the specified object is the previous version.
geneObj	Finds, and optionally opens, the genealogy associated with this object.
mark_modified	(ODMG) Opens this persistent object for update.
nextVers	Initializes an object iterator to find, and optionally open, all next versions that are created from this object and that satisfy any specified selection criteria.
ooCopyInit	Default implementation for performing custom postprocessing when this object is copied.
ooGetTypeN	Gets the type number of this persistent object's class.
ooGetTypeName	Gets the name of this persistent object's class.
ooIsKindOf	Tests whether this persistent object belongs to the class with the specified type number or to a class derived from that class.
ooNewVersInit	Default implementation for performing custom postprocessing when a new version of this object is created.
ooObj	Default constructor that constructs a new basic object.
ooPreMoveInit	Default implementation for performing custom preprocessing when this object is moved.
ooPostMoveInit	Default implementation for performing custom postprocessing when this object is moved.
ooThis	Sets an object reference or handle to reference this persistent object.
ooUpdate	Opens this persistent object for update.
ooValidate	Default implementation for testing whether this object is valid.

operator delete	Deletes this persistent object.
operator new	Creates a new basic object using the specified clustering directive, if any.
prevVers	Finds, and optionally opens, the previous version of this object.
set_defaultToGeneObj	Links this object with the specified genealogy, making this object the default version in the genealogy.
set_geneObj	Links this object with the specified genealogy, making this object a version in the genealogy.
set_prevVers	Links this object to the specified object, making the specified object the previous version of this object.
sub_derivatives	Deletes one or more `derivatives` associations from this object to the specified destination object, indicating that the specified object is no longer a derivative of this object.
sub_derivedFrom	Deletes one or more `derivedFrom` associations that link this object to the specified object, indicating that the specified object is no longer an ancestor version of this object.
sub_nextVers	Deletes the `nextVers` association that links this object to the specified object, so the specified object is no longer a next version of this object.

Constructors

ooObj

Default constructor that constructs a new basic object.

```
ooObj();
```

Operators

operator delete

Deletes this persistent object.

```
void operator delete(void *objP);
```

Parameters

objP

Pointer to the persistent object to be deleted. This pointer must be extracted from a handle to the object—for example, through operator* on that handle.

Discussion

This operator is an alternative to the ooDelete global function. Note that ooDelete is the recommended choice, however, because it doesn't require you to extract a pointer to the object to be deleted (you specify a handle or object reference instead).

Deleting a basic object or container:

- Calls the object's destructor, if any, before deallocating storage.
- Deletes all associations from the object to destination objects. If an association has delete propagation enabled, the destination objects are deleted as well.

Deleting a container:

- Deletes all of the basic objects in it.
- Deletes all associations from each contained object to destination objects; if delete propagation is enabled for any of these associations, the destination objects are deleted as well.
- Does *not* call the destructors of the contained objects for performance reasons. To ensure that destructors are called, you must iterate through the contained objects and explicitly delete them before deleting the container.

Each deleted object is automatically removed from any bidirectional associations to maintain referential integrity. The delete operation must therefore be able to obtain an update lock on every object that is bidirectionally associated with a deleted object. Note that if another perisistent object references the deleted object through a unidirectional association or directly in one of its data members, you are responsible for removing that reference.

Example

Assume dbh is a valid handle to a database, and that Net is a persistence-capable class, which therefore inherits operator delete from ooObj.

```
ooHandle(Net) netH = new(dbh) Net("vdd");// Create net object
...
Net *pVdd = netH;  // Extract pointer to net object from handle
delete pVdd;       // Delete net object
```

operator new

Creates a new basic object using the specified clustering directive, if any.

```
1.   void *operator new(
       size_t);

2.   void *operator new(
       size_t,
       const ooRefHandle(ooObj) &near);

3.   void *operator new(
       size_t,
       const ooObj *near);

(ODMG) 4.   void *operator new(
       size_t,
       ooRefHandle(ooObj) near,
       const char *type);

(ODMG) 5.   void *operator new(
       size_t,
       d_Database *near,
       const char *type = 0);
```

Parameters

size_t

Do not specify; this parameter is automatically initialized by the compiler with the size of the class type in bytes.

near

Specifies whether the new basic object is to be persistent or transient. If the new basic object is persistent, *near* is the clustering directive that specifies the object's location in the federated database.

To create a transient basic object, specify 0 or a pointer to a transient object.

To create a persistent basic object, choose one of the following alternatives as the clustering directive:

- Omit *near* (variant 1).
- Specify an object reference, handle, or pointer to a database, container, or persistent basic object (variants 2 through 4). *near* may not reference a federated database or an autonomous partition.
- (*ODMG*) Specify a pointer to a `d_Database` object that references the federated database (variant 5).

When you create a persistent basic object:

- If you omit *near*, the new basic object is created in the default container of the most recently opened or created database. An error is signalled if no such database exists.

- If *near* references a database, the new basic object is created in the default container of that database.
- If *near* references a container, the new basic object is created in that container.
- If *near* references a basic object, the new basic object is put into the same container as the specified basic object. If possible, the new object will be put on the same page as the specified object or on a nearby page.
- (*ODMG*) If *near* specifies a valid `d_Database` object, the new basic object is created in the default container of the most recently opened or created database in the federation. If no such database exists, the object is created in the default container of a database called `default_odmg_db`; if necessary, this database is created.

type

(*ODMG*) This parameter is ignored. By convention it is the name of the class you are instantiating.

Returns

Memory pointer to the new basic object. This pointer is null if an error occurs during the creation of the object.

Discussion

When you create a new persistent basic object, you must:

- Use `operator new` in an update transaction. The new basic object is made permanent on disk when the transaction commits or is checkpointed. If the transaction is aborted, the object is not created.
- Assign the result of `operator new` directly to a handle. You can verify the creation of the basic object by checking whether the handle is null. (If an object reference is desired, you can then assign the handle to an object reference.)

WARNING 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.

Member Functions

add_derivatives

Links this object with the specified object, making the specified object a derivative (successor version) of this object.

```
ooStatus add_derivatives(const ooHandle(ooObj) &object);
```

Parameters *object*

Handle to a basic object.

Returns `oocSuccess` if successful; otherwise `oocError`.

Discussion This operation is normally used when merging two version branches into a single branch.

The operation links the specified object to this object by:

- Creating a `derivatives` association link in this object.
- Creating the inverse `derivedFrom` association link in the specified object.

The application must be able to obtain update locks on both objects.

See also `derivatives`
`sub_derivatives`
`del_derivatives`

add_derivedFrom

Links this object with the specified object, making the specified object an ancestor version of this object.

```
ooStatus add_derivedFrom(const ooHandle(ooObj) &object);
```

Parameters *object*

Handle to a basic object.

Returns `oocSuccess` if successful; otherwise `oocError`.

Discussion This operation is normally used when merging two version branches into a single branch.

The operation links the specified object to this object by:

- Creating a `derivedFrom` association link in this object.
- Creating an inverse `derivatives` association link in the specified object.

The application must be able to obtain update locks on both objects.

See also `derivedFrom`
`sub_derivedFrom`
`del_derivedFrom`

add_nextVers

Links this object with the specified object, making the specified object a next version of this object.

```
ooStatus add_nextVers(const ooHandle(ooObj) &object);
```

Parameters

object

Handle to a basic object.

Returns

`oocSuccess` if successful; otherwise `oocError`.

Discussion

The operation links the specified object to this object by:

- Creating a `nextVers` association link in this object.
- Creating an inverse `prevVers` association link in the specified object.

The application must be able to obtain update locks on both objects.

See also

`nextVers`
`sub_nextVers`
`del_nextVers`

defaultToGeneObj

Finds, and optionally opens, the genealogy associated with this object, if this object is the default version in a genealogy.

```
1.  ooHandle(ooGeneObj) defaultToGeneObj
      (const ooMode openMode = oocNoOpen) const;
2.  ooRef(ooGeneObj) &defaultToGeneObj(
      ooRef(ooGeneObj) &genealogy,
      const ooMode openMode = oocNoOpen) const;
3.  ooHandle(ooGeneObj) &defaultToGeneObj(
      ooHandle(ooGeneObj) &genealogy,
      const ooMode openMode = oocNoOpen) const;
```

Parameters

genealogy

Object reference or handle to set to the found genealogy.

openMode

Intended level of access to the found genealogy:

- Specify `oocNoOpen` (the default) to set the returned object reference or handle to the object without opening it.
- Specify `oocRead` to open the object for read.
- Specify `oocUpdate` to open the object for update.

Returns Object reference or handle to the found genealogy. A null object reference or handle is returned if no genealogy is found.

Discussion When called without a *genealogy* parameter, this member function allocates a new genealogy handle and returns it. Otherwise, this member function returns the result in the object reference or handle that is passed to it.

You can call this member function on an object to test whether it is the default version in a genealogy. The object is a default version if an associated genealogy is found; otherwise, if null is returned, the object being tested is not a default version.

See also `set_defaultToGeneObj`
`del_defaultToGeneObj`

del_defaultToGeneObj

Deletes any `defaultToGeneObj` association from this object, indicating this object is no longer the default version.

```
ooStatus del_defaultToGeneObj();
```

Returns `oocSuccess` if successful; otherwise `oocError`.

Discussion This operation also deletes the inverse `defaultVers` association link from the formerly associated genealogy. The application must be able to obtain update locks on both objects.

You typically call this function to remove a genealogy's previous default version before setting a new one. You shouldn't leave a genealogy without a default version.

You should call the `exist_defaultToGeneObj` member function to test whether an association exists before you try to delete it.

See also `defaultToGeneObj`
`set_defaultToGeneObj`

del_derivatives

Deletes all `derivatives` associations from this object, indicating this object has no derivatives (successor versions).

```
ooStatus del_derivatives();
```

Returns `oocSuccess` if successful; otherwise `oocError`.

Discussion This operation also deletes the inverse `derivedFrom` association link from each of the former derivatives. The application must be able to obtain update locks on all of the affected objects.

You should call the `exist_derivatives` member function to test whether any associations exist before you try to delete them.

See also `derivatives`
`add_derivatives`
`sub_derivatives`

del_derivedFrom

Deletes all `derivedFrom` associations from this object, indicating this object is not derived from any ancestor versions.

```
ooStatus del_derivedFrom();
```

Returns `oocSuccess` if successful; otherwise `oocError`.

Discussion This operation also deletes the inverse `derivatives` association link from each of the former ancestor versions. The application must be able to obtain update locks on all of the affected objects.

You should call the `exist_derivedFrom` member function to test whether any associations exist before you try to delete them.

See also `derivedFrom`
`add_derivedFrom`
`sub_derivedFrom`

del_geneObj

Deletes any `geneObj` association that exists for this object, indicating this object is no longer in a genealogy.

```
ooStatus del_geneObj();
```

Returns `oocSuccess` if successful; otherwise `oocError`.

Discussion This operation also deletes the inverse `allVers` association link from the formerly associated genealogy. The application must be able to obtain update locks on both objects.

You should call the `exist_geneObj` member function to test whether an association exists before you try to delete it.

See also geneObj
set_geneObj

del_nextVers

Deletes all nextVers associations that exist for this object, indicating that this object has no next versions.

```
ooStatus del_nextVers();
```

Returns oocSuccess if successful; otherwise oocError.

Discussion This operation also deletes the inverse prevVers association link from each of the former next versions. The application must be able to obtain update locks on all of the affected objects.

You should call the exist_nextVers member function to test whether any associations exist before you try to delete them.

See also nextVers
add_nextVers
sub_nextVers

del_prevVers

Deletes any prevVers association that exists for this object, indicating this object has no previous version.

```
ooStatus del_prevVers();
```

Returns oocSuccess if successful; otherwise oocError.

Discussion This operation also deletes the inverse nextVers association from the former previous version. The application must be able to obtain update locks on both objects.

You should call the exist_prevVers member function to test whether an association exists before you try to delete it.

See also prevVers
set_prevVers

derivatives

Initializes an object iterator to find, and optionally open, all versions that are derived from this object and that satisfy any specified selection criteria.

```
1.  ooStatus derivatives(
      ooItr(ooObj) &iterator,
      const ooMode openMode = oocNoOpen) const;
2.  oStatus derivatives(
      ooItr(ooObj) &iterator,
      const char *predicate) const;
3.  ooStatus derivatives(
      ooItr(ooObj) &iterator,
      const ooMode openMode,
      const ooAccessMode access,
      const char *predicate) const;
```

Parameters

iterator

Object iterator for finding all derivative versions.

openMode

Intended level of access to each derivative object found by the iterator's `next` member function:

- `oocNoOpen` (the default in variant 1) causes `next` to set the iterator to the next derivative object without opening it.
- `oocRead` causes `next` to open the next derivative object for read.
- `oocUpdate` causes `next` to open the next derivative object for update.

  Warning: If versioning is enabled for one or more found objects, specifying `oocUpdate` means that `next` will create a new version of each such object.

predicate

String expression in predicate query language; specifies the condition to be met by the found objects. The iterator is initialized only with derivative objects that match *`predicate`*.

access

Level of access control of the data members that *`predicate`* can test:

- Specify `oocPublic` to permit the predicate to test only public data members, preserving encapsulation.
- Specify `oocAll` to permit the predicate to test any data member. To preserve encapsulation, you should use this mode only within member functions of the class you are querying.

Returns `oocSuccess` if successful; otherwise `oocError`.

See also
```
del_derivatives
sub_derivatives
add_derivatives
```

derivedFrom

Initializes an object iterator to find, and optionally open, all ancestor versions from which this object is derived, and that satisfy any specified selection criteria.

```
1.  ooStatus derivedFrom(
      ooItr(ooObj) &iterator,
      const ooMode openMode = oocNoOpen) const;
2.  ooStatus derivedFrom(
      ooItr(ooObj) &iterator,
      const char *predicate) const;
3.  ooStatus derivedFrom(
      ooItr(ooObj) &iterator,
      const ooMode openMode,
      const ooAccessMode access,
      const char *predicate) const;
```

Parameters

iterator

Object iterator for finding all ancestor versions.

openMode

Intended level of access to each object found by the iterator's `next` member function:

- `oocNoOpen` (the default in variant 1) causes `next` to set the iterator to the next found object without opening it.
- `oocRead` causes `next` to open the next found object for read.
- `oocUpdate` causes `next` to open the next found object for update.

  Warning: If versioning is enabled for one or more found objects, specifying `oocUpdate` means that `next` will create a new version of each such object.

predicate

String expression in predicate query language; specifies the condition to be met by the found objects. The iterator is initialized only with found objects that match *predicate*.

access

Level of access control of the data members that *predicate* can test:

- Specify `oocPublic` to permit the predicate to test only public data members, preserving encapsulation.
- Specify `oocAll` to permit the predicate to test any data member. To preserve encapsulation, you should use this mode only within member functions of the class you are querying.

Returns `oocSuccess` if successful; otherwise `oocError`.

See also
```
add_derivedFrom
sub_derivedFrom
del_derivedFrom
```

exist_defaultToGeneObj

Tests whether this object is the default version in any genealogy or, if a parameter is given, in the specified genealogy.

```
1.  ooBoolean exist_defaultToGeneObj() const;
2.  ooBoolean exist_defaultToGeneObj(
      const ooHandle(ooGeneObj) &genealogy) const;
```

Parameters *genealogy*

Handle to the genealogy to be tested.

Returns (Variant 1) `oocTrue` if this object is the default version in any genealogy, otherwise `oocFalse`.
(Variant 2) `oocTrue` if this object is the default version in the specified genealogy, otherwise `oocFalse`.

Discussion This member function tests whether this object has a `defaultToGeneObj` association link to a genealogy. If so, this object is by convention the default version in the corresponding genealogy.

exist_derivatives

Tests whether this object has any derivatives (successor versions); if a parameter is given, tests whether the specified object is a derivative of this object.

```
1.  ooBoolean exist_derivatives() const;
2.  ooBoolean exist_derivatives(
      const ooHandle(ooObj) &object) const;
```

Parameters *object*

Handle to the object to be tested as a derivative.

Returns (Variant 1) `oocTrue` if this object has any derivatives, otherwise `oocFalse`.
(Variant 2) `oocTrue` if the specified object is a derivative of this object, otherwise `oocFalse`.

exist_derivedFrom

Tests whether this object is derived from any other objects (ancestor versions), or, if a parameter is given, from the specified object.

```
ooBoolean exist_derivedFrom(
     const ooHandle(ooObj) &object) const;
```

Parameters *object*

Handle to the object to be tested.

Returns (Variant 1) `oocTrue` if this object is derived from any other object, otherwise `oocFalse`.
(Variant 2) `oocTrue` if this object is derived from the specified object, otherwise `oocFalse`.

exist_geneObj

Tests whether this object is a version within any genealogy, or, if a parameter is given, within the specified genealogy.

```
1.   ooBoolean exist_geneObj() const;
2.   ooBoolean exist_geneObj(
       const ooHandle(ooGeneObj) &genealogy) const;
```

Parameters *genealogy*

Handle to the genealogy to be tested.

Returns (Variant 1) `oocTrue` if this object belongs to any genealogy, otherwise `oocFalse`.
(Variant 2) `oocTrue` if this object belongs to the specified genealogy, otherwise `oocFalse`.

Discussion This member function tests whether this object has a `geneObj` association to a genealogy. If so, this object is by convention a version within the corresponding genealogy.

exist_nextVers

Tests whether this object has a next version; if a parameter is given, tests whether the specified object is the next version.

```
1.  ooBoolean exist_nextVers() const;
2.  ooBoolean exist_nextVers(
      const ooHandle(ooObj) &object) const;
```

Parameters *object*

Handle to the object to be tested as a next version.

Returns (Variant 1) `oocTrue` if this object has any next version, otherwise `oocFalse`. (Variant 2) `oocTrue` if the specified object is a next version of this object, otherwise `oocFalse`.

exist_prevVers

Tests whether this object has a previous version; if a parameter is given, tests whether the specified object is the previous version.

```
1.  ooBoolean exist_prevVers() const;
2.  ooBoolean exist_prevVers(
      const ooHandle(ooObj) &object) const;
```

Parameters *object*

Handle to the object to be tested as a previous version.

Returns (Variant 1) `oocTrue` if this object has any previous version, otherwise `oocFalse`. (Variant 2) `oocTrue` if the specified object is a previous version of this object, otherwise `oocFalse`.

geneObj

Finds, and optionally opens, the genealogy associated with this object.

```
1.  ooHandle(ooGeneObj) geneObj(
      const ooMode openMode = oocNoOpen) const;
2.  ooRef(ooGeneObj) &geneObj(
      ooRef(ooGeneObj) &genealogy,
      const ooMode openMode = oocNoOpen) const;
3.  ooHandle(ooGeneObj) &geneObj(
      ooHandle(ooGeneObj) &genealogy,
      const ooMode openMode = oocNoOpen) const;
```

Parameters

genealogy

Object reference or handle to set to the found genealogy.

openMode

Intended level of access to the found genealogy:

- Specify `oocNoOpen` (the default) to set the returned object reference or handle to the object without opening it.
- Specify `oocRead` to open the object for read.
- Specify `oocUpdate` to open the object for update.

Returns

Object reference or handle to the found genealogy. A null object reference or handle is returned if no genealogy is found.

Discussion

When called without a *genealogy* parameter, this member function allocates a new genealogy-object handle and returns it. Otherwise, this member function returns the result in the object reference or handle that is passed to it.

See also

`set_geneObj`
`del_geneObj`

mark_modified

(*ODMG*) Opens this persistent object for update.

```
void mark_modified();
```

Discussion

This member function is an ODMG-equivalent name for the member function `ooUpdate`.

nextVers

Initializes an object iterator to find, and optionally open, all next versions that are created from this object and that satisfy any specified selection criteria.

```
1.  ooStatus nextVers(
      ooItr(ooObj) &iterator,
      const ooMode openMode = oocNoOpen) const;
2.  ooStatus nextVers(
      ooItr(ooObj) &iterator,
      const char *predicate) const;
```

```
3.  ooStatus nextVers(
      ooItr(ooObj) &iterator,
      const ooMode openMode,
      const ooAccessMode access,
      const char *predicate) const;
```

Parameters

iterator

Object iterator for finding the next version(s) created from this object:

- Under linear versioning, this iterator finds at most one version.
- Under branch versioning, this iterator may find multiple versions.

openMode

Intended level of access to the versions found by the iterator's `next` member function:

- `oocNoOpen` (the default in variant 1) causes `next` to set the iterator to the next version without opening it.
- `oocRead` causes `next` to open the next version for read.
- `oocUpdate` causes `next` to open the next version for update.

  Warning: If versioning is enabled for one or more found objects, specifying `oocUpdate` means that `next` will create a new version of each such object.

predicate

String expression in predicate query language; specifies the condition to be met by the found objects. The iterator is initialized only with versions that match *predicate*.

access

Level of access control of the data members that *predicate* can test:

- Specify `oocPublic` to permit the predicate to test only public data members, preserving encapsulation.
- Specify `oocAll` to permit the predicate to test any data member. To preserve encapsulation, you should use this mode only within member functions of the class you are querying.

Returns

`oocSuccess` if successful; otherwise `oocError`.

See also

`add_nextVers`
`sub_nextVers`
`del_nextVers`

ooCopyInit

Default implementation for performing custom postprocessing when this object is copied.

```
virtual ooStatus ooCopyInit();
```

Returns `oocSuccess` if postprocessing is successful; otherwise `oocError`.

Discussion This member function is invoked automatically after a basic object is copied—for example, by the `copy` member function on a handle to the object. The default implementation simply returns `oocSuccess`.

An application can override `ooCopyInit` in a basic-object class to perform custom postprocessing after an instance of the class is copied—typically:

- Performing any necessary operations on attribute data members for which bit-wise copying is inadequate.
- Propagating the copy operation to associated or referenced objects (creating a deep copy).

ooGetTypeN

Gets the type number of this persistent object's class.

```
virtual ooTypeNumber ooGetTypeN() const;
```

Returns Type number of this object's class.

Discussion Every persistence-capable class has a schema-defined type number that uniquely identifies the class to the federated database. You can use this number to determine whether an object is an instance of a particular class.

You normally use `ooGetTypeN` in a member function of an application-defined persistence-capable class; when such a member function is called on an object of the class, `ooGetTypeN` provides the member function with the type number of the object's class.

If you already have a handle or object reference to a persistent object, you can equivalently call the `typeN` member function on the handle or object reference.

See also `ooGetTypeName`
`ooIsKindOf`
`ooTypeN` global macro
`ooRefHandle``(ooObj)::typeN` member function

ooGetTypeName

Gets the name of this persistent object's class.

```
virtual char *ooGetTypeName() const;
```

Returns String containing the name of this object's class.

Discussion You normally use `ooGetTypeName` in a member function of an application-defined persistence-capable class; when such a member function is called on an object of the class, `ooGetTypeName` provides the member function with the name of the object's class.

If you already have a handle or object reference to a persistent object, you can equivalently call the `typeName` member function on the handle or object reference.

WARNING Do not modify the returned string in any manner. Doing so may result in unexpected program errors. This string is used internally by Objectivity / DB.

See also `ooGetTypeN`
`ooIsKindOf`
`ooTypeN` global macro
`ooRefHandle``(ooObj)::typeName` member function

ooIsKindOf

Tests whether this persistent object belongs to the class with the specified type number or to a class derived from that class.

```
virtual ooBoolean ooIsKindOf(
     const ooTypeNumber typeN) const;
```

Parameters *`typeN`*

Type number of a persistence-capable class. If you know the name of the desired class, you can call the `ooTypeN` global macro to obtain its type number. If you have a handle to an object of the desired class, you can call the `typeN` member function on the handle to obtain the class's type number.

Returns `oocTrue` if this object is an instance of the class with type number *`typeN`* or to a class derived from that class; otherwise `oocFalse`.

Discussion You normally call `ooIsKindOf` to perform runtime type identification—for example, to determine whether it is safe to downcast a handle that references this object.

See also
ooGetTypeN
ooGetTypeName
ooTypeN global macro
ooRefHandle(ooObj)::typeN member function
ooRefHandle(ooObj)::typeName member function

ooNewVersInit

Default implementation for performing custom postprocessing when a new version of this object is created.

```
virtual ooStatus ooNewVersInit();
```

Returns
oocSuccess if postprocessing is successful; otherwise oocError.

Discussion
This member function is invoked automatically when a new version of a basic object is created. A new version is created when you open an object for update after versioning has been enabled for it. The default implementation simply returns oocSuccess.

A new version is a bit-wise copy of the existing object. An application can override ooNewVersInit in a basic-object class to perform postprocessing after an instance of the class is versioned (copied)—typically:

- Performing any necessary operations on attribute data members for which bit-wise copying is inadequate.
- Propagating the versioning operation to associated or referenced objects (creating a deep copy).

ooPostMoveInit

Default implementation for performing custom postprocessing when this object is moved.

```
virtual ooStatus ooPostMoveInit();
```

Returns
oocSuccess if postprocessing is successful; otherwise oocError.

Discussion
This member function is invoked automatically after a basic object is moved—for example, by the move member function on a handle to the object. The default implementation simply returns oocSuccess.

An application can override ooPostMoveInit in a basic-object class to perform postprocessing after an instance of the class is moved. Such postprocessing typically reestablishes references to an object after it acquires a new object identifier—for example, by updating any referencing attributes, unidirectional

associations, persistent collections, or scope names. (Objectivity / DB automatically maintains referential integrity for bidirectional associations.)

ooPreMoveInit

Default implementation for performing custom preprocessing when this object is moved.

```
virtual ooStatus ooPreMoveInit();
```

Returns `oocSuccess` if preprocessing is successful; otherwise `oocError`.

Discussion This member function is invoked automatically before a basic object is moved—for example, by the `move` member function on a handle to the object. The default implementation simply returns `oocSuccess`.

An application can override `ooPreMoveInit` in a basic-object class to perform preprocessing before an instance of the class is moved. Such preprocessing typically removes existing references to an object before its object identifier becomes invalid—for example, to maintain referential integrity for any referencing attributes, unidirectional associations, persistent collections, or scope names. (Objectivity / DB automatically maintains referential integrity for bidirectional associations.)

ooThis

Sets an object reference or handle to reference this persistent object.

```
1.  ooHandle(ooObj) ooThis() const;
2.  ooRef(ooObj) &ooThis(
      ooRef(ooObj) &object) const;
3.  ooHandle(ooObj) &ooThis(
      ooHandle(ooObj) &object) const;
```

Parameters *`object`*

Object reference or handle to be set to this object.

Returns Object reference or handle to this object.

Discussion You normally use `ooThis` in a member function of an application-defined persistence-capable class. When such a member function is called on an object of the class, `ooThis` provides the member function with an object reference or handle to the object. The member function can then perform operations on the object that are available only through an object reference or handle.

When called without an *object* parameter, ooThis allocates a new handle and returns it. Otherwise, ooThis returns the object reference or handle that is passed to it.

Calling ooThis in a member function of a persistence-capable class serves the same purpose as using the C++ this keyword in a member function of a non-persistence-capable class—both ooThis and this enable you to access the object on which the member function is called. However, ooThis and this differ in the following ways:

- The this keyword is the pointer to the object. Syntactically, the this keyword is a name, and can be used in expressions such as this->get();
- The ooThis member function returns an object reference or handle to the object. Syntactically, ooThis is a function and can be used in expressions such as ooThis()->get();

Member functions of persistence-capable classes should use ooThis (and not this) because the only safe way to operate on a persistent object is through an object reference or handle.

WARNING ooThis is the only safe way to obtain an object reference or handle within a member function; do *not* attempt to initialize an object reference or handle by assigning the this pointer to it.

See also *appClass*::ooThis

ooUpdate

Opens this persistent object for update.

```
ooStatus ooUpdate();
```

Returns oocSuccess if successful; otherwise oocError.

Discussion You normally use ooUpdate in a member function of an application-defined persistence-capable class; doing so informs Objectivity / DB that the member function intends to modify one or more attributes of the object on which it is called. For example, each accessor member function that sets a data-member value should call ooUpdate, to ensure that Objectivity / DB will save the modification persistently.

If you already have a handle or object reference to the persistent object to be modified, you can equivalently call the update member function on the handle or object reference.

ooValidate

Default implementation for testing whether this object is valid.

```
virtual ooBoolean ooValidate();
```

Returns `oocTrue` if this object is valid; otherwise, `oocFalse`.

Discussion The `ooValidate` member function provides a framework for application-specific validation of persistent objects. The default implementation simply returns `oocTrue`.

An application can override this function in a persistence-capable class to perform whatever checks are necessary to test whether an object of the class is valid.

prevVers

Finds, and optionally opens, the previous version of this object.

```
1.  ooHandle(ooObj) prevVers(
      const ooMode openmode = oocNoOpen) const;
2.  ooHandle(ooObj) &prevVers(
      ooHandle(ooObj) &object,
      const ooMode openmode = oocNoOpen) const;
3.  ooRef(ooObj) &prevVers(
      ooRef(ooObj) &object,
      const ooMode openmode = oocNoOpen) const;
```

Parameters *object*

Object reference or handle to be set to the previous version.

openMode

Intended level of access to the previous version:

- Specify `oocNoOpen` (the default) to set the returned object reference or handle to the object without opening it.
- Specify `oocRead` to open the object for read.
- Specify `oocUpdate` to open the object for update.

Returns Object reference or handle to the previous version. A null object reference or handle is returned if no previous version exists.

Discussion When called without an *object* parameter, this member function allocates a new handle and returns it. Otherwise, this member function returns the result in the object reference or handle that is passed to it.

See also
set_prevVers
del_prevVers

set_defaultToGeneObj

Links this object with the specified genealogy, making this object the default version in the genealogy.

```
ooStatus set_defaultToGeneObj(
     const ooHandle(ooGeneObj) &genealogy);
```

Parameters

genealogy

Handle to a genealogy.

Returns

`oocSuccess` if successful; otherwise `oocError`.

Discussion

The operation links the specified genealogy to this object by:

- Creating a `defaultToGeneObj` association link in this object.
- Creating the inverse `defaultVers` association link in the specified genealogy.

The application must be able to obtain update locks on both objects.

Because `defaultToGeneObj` is a to-one association, an error is signaled if this object already has a link for the association. That is, you must remove any existing default version from the genealogy before you set a new default version.

See also
defaultToGeneObj
del_defaultToGeneObj

set_geneObj

Links this object with the specified genealogy, making this object a version in the genealogy.

```
ooStatus set_geneObj(const ooHandle(ooGeneObj) &genealogy);
```

Parameters

genealogy

Handle to a genealogy.

Returns

`oocSuccess` if successful; otherwise `oocError`.

Discussion

The operation links the specified genealogy to this object by:

- Creating a `geneObj` association link in this object.
- Creating the inverse `allVers` association link in the specified genealogy.

The application must be able to obtain update locks on both objects.

Because `geneObj` is a to-one association, an error is signaled if this object already has a link for the association. That is, you must remove this object from its current genealogy before you make it a version of the specified genealogy.

See also `geneObj`
`del_geneObj`

set_prevVers

Links this object to the specified object, making the specified object the previous version of this object.

```
ooStatus set_prevVers(const ooHandle(ooObj) &object);
```

Parameters *object*

Handle to a basic object.

Returns `oocSuccess` if successful; otherwise `oocError`.

Discussion The operation links the specified object to this object by:

- Creating a `prevVers` association link in this object.
- Creating the inverse `nextVers` association link in the specified object.

The application must be able to obtain update locks on both objects.

Because `prevVers` is a to-one association, an error is signaled if this object already has a link for the association. That is, you must remove the current previous version from this object before you set a new previous version.

See also `prevVers`
`del_prevVers`

sub_derivatives

Deletes one or more `derivatives` associations from this object to the specified destination object, indicating that the specified object is no longer a derivative of this object.

```
ooStatus sub_derivatives(
     const ooHandle(ooObj) &object)
     const uint32 number = 1);
```

Parameters *object*

Handle to the unwanted derivative version.

number

Number of derivatives association links to delete between this object and the specified object:

- If you specify 0, all such links are deleted.
- If you specify 1 (the default), the first or only such link is deleted.
- If you specify a number greater than 1, this member function deletes the first *number* links encountered.

Returns

oocSuccess if successful; otherwise oocError. oocSuccess is returned even if no link exists to be deleted or if *number* exceeds the number of existing links. oocError results from internal errors or locking errors.

Discussion

Because derivatives is a many-to-many bidirectional association, it is possible for multiple links to exist between this object and the specified destination object. The *number* parameter allows you specify how many such links to delete.

See also

derivatives
del_derivatives
add_derivatives

sub_derivedFrom

Deletes one or more derivedFrom associations that link this object to the specified object, indicating that the specified object is no longer an ancestor version of this object.

```
ooStatus sub_derivedFrom(
     const ooHandle(ooObj) &object)
     const uint32 number = 1);
```

Parameters

object

Handle to the unwanted ancestor version.

number

Number of derivedFrom association links to delete between this object and the specified object.

- If you specify 0, all such links are deleted.
- If you specify 1 (the default), the first or only such link is deleted.
- If you specify a number greater than 1, this member function deletes the first *number* links encountered.

Returns

oocSuccess if successful; otherwise oocError. oocSuccess is returned even if no link exists to be deleted or if *number* exceeds the number of existing links. oocError results from internal errors or locking errors.

Discussion

Because `derivedFrom` is a many-to-many bidirectional association, it is possible for multiple links to exist between this object and the specified ancestor version. The *`number`* parameter allows you specify how many such links to delete.

See also

`derivedFrom`
`del_derivedFrom`
`add_derivedFrom`

sub_nextVers

Deletes the `nextVers` association that links this object to the specified object, so the specified object is no longer a next version of this object.

```
ooStatus sub_nextVers(
     const ooHandle(ooObj) &object);
```

Parameters

`object`

Handle to the unwanted next version.

Returns

`oocSuccess` if successful; otherwise `oocError`.

See also

`add_nextVers`
`nextVers`
`del_nextVers`

ooOperatorSet Class

Inheritance: **ooOperatorSet**

The non-persistence-capable class `ooOperatorSet` represents an *operator set*—that is, a set of application-defined relational operators.

See:

- "Reference Summary" on page 466 for an overview of member functions
- "Reference Index" on page 466 for a list of member functions

About Operator Sets

An operator set is a collection of application-defined relational operators, where each relational operator consists of both of the following:

- An application-defined *operator function* that conforms to the calling interface defined by the `ooQueryOperatorPtr` function pointer type.
- A *token* that represents the operator function in predicates.

When an operator set is assigned to the `ooUserDefinedOperators` global variable, the predicate query mechanism consults it to interpret any nonstandard tokens in a predicate.

An application normally uses the default operator set that is created and assigned to `ooUserDefinedOperators` when the application starts. An application can also create additional operator sets (for example, to define alternative sets of relational operators). However, only one operator set can be in effect at a time; if you create an operator set, you must assign it to the `ooUserDefinedOperators` global variable to make its operators available to Objectivity / DB.

An operator set is empty at creation, and you add relational operators to it by calling the `registerOperator` member function on the operator set. At any

point in the application, you can call the `clear` member function to remove all the currently registered operators from the set.

The `ooUserDefinedOperators` global variable is not thread-safe, so 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.

Reference Summary

Creating	`ooOperatorSet`
Adding and Removing Operator Functions	`registerOperator` `clear`

Reference Index

Constructors and Destructors

ooOperatorSet

Default constructor that constructs a new empty operator set.

```
ooOperatorSet();
```

Member Functions

registerOperator

Registers the specified operator function and its token with this operator set.

```
ooStatus registerOperator(
     const char *name,
     ooQueryOperatorPtr funcPtr);
```

Parameters

name

Token that is to represent the operator function in a predicate. This token may not begin or end with the following symbols or symbol combinations:

```
) ( && || ! ,  .
```

If the token is the same as an existing operator, the new operator will override the standard behavior.

funcPtr

Pointer to the operator function to be registered.

Returns

`oocSuccess` if successful; otherwise `oocError`.

See also

`ooQueryOperatorPtr`

clear

Clears all application-defined relational operators from this operator set.

```
clear();
```

ooQuery Class

Inheritance: **ooQuery**

The non-persistence-capable class `ooQuery` represents a *query object*, which evaluates whether a persistent object matches a given predicate string.

See:

- "Reference Index" on page 469 for a list of member functions

About Query Objects

You use a query object to filter arbitrary groups of objects that cannot be scanned directly with a predicate string. For example, you can use a query object to perform a predicate query over the group of objects referenced by a VArray of object references.

Reference Index

`evaluate`	Evaluates whether the referenced object matches this query object's predicate string.
`setup`	Sets up a query object for evaluating the specified type of objects against the specified predicate string.

Member Functions

evaluate

Evaluates whether the referenced object matches this query object's predicate string.

```
ooBoolean evaluate(ooHandle(ooObj) &objH);
```

Parameters

objH

Handle to the object to be evaluated.

Returns

`oocTrue` if the object is a match for the predicate string specified by `setup`; otherwise `oocFalse`.

setup

Sets up a query object for evaluating the specified type of objects against the specified predicate string.

```
ooStatus setup(
     char *predicate,
     ooTypeNumber typeN,
     ooOperatorSet *operatorSet = 0);
```

Parameters

predicate

Predicate string for the query object.

typeN

Type number of the class whose instances are to be evaluated.

operatorSet

The operator set to be consulted if *predicate* uses tokens for any application-defined relational operators. You must specify the operator set explicitly, even if it is the default operator set (that is, the current value of the `ooUserDefinedOperators` variable).

Specifying 0 causes all tokens in *predicate* to be interpreted as Objectivity / DB standard relational operators.

Returns

`oocSuccess` if successful; otherwise `oocError`.

ooRefHandle(*appClass*) Classes

Inheritance: **ooRef(ooObj)->ooRef(*appClass*)**

ooRef(ooObj)->ooRef(ooContObj)->ooRef(*appClass*)

Inheritance: **ooHandle(ooObj)->ooHandle(*appClass*)**

ooHandle(ooObj)->ooHandle(ooContObj)->ooHandle(*appClass*)

The abbreviation *ooRefHandle*(*appClass*) refers to two non-persistence-capable classes:

- ooRef(*appClass*), which represents an *object reference* to a persistent object of the application-defined class *appClass*.
- ooHandle(*appClass*), which represents a *handle* to a persistent object of the application-defined class *appClass*.

These two classes are documented together because they define almost identical sets of member functions (exceptions are listed in the "Reference Summary"). These classes provide the primary interface for operating on persistent objects of the application-defined persistence-capable class appClass and its derived classes. Note that *appClass* can be either a basic-object class (derived from ooObj) or a container class (derived from ooContObj).

See:

- "Reference Summary" on page 478 for an overview of member functions
- "Reference Index" on page 479 for a list of member functions

To use the *ooRefHandle*(*appClass*) classes, you must include and compile with files generated by the DDL processor, as described in "Obtaining Generated Class Definitions" on page 474.

(*ODMG*) You can use the ODMG standard class name d_Ref<*appClass*> interchangeably with ooRef(*appClass*).

About *appClass* Handles and References

When an application defines a persistence-capable class *appClass* and adds it to the federated-database schema, the DDL processor generates the corresponding handle and object-reference classes *ooRefHandle*(*appClass*). The application works with each persistent object of class *appClass* indirectly through one or more instances of *ooRefHandle*(*appClass*) that are set to reference the object. A handle or object reference to a persistent object serves as a type-safe smart pointer that:

- Identifies the persistent object to the application or to another object.
- Provides an interface for operating on the persistent object.
- Manages the memory pointer to the persistent object.
- Provides an indirect member-access operator (->) for accessing the persistent object's public member functions.

It is sometimes more appropriate to use a handle rather than an object reference, and vice versa; the choice is described in "Structure and Behavior" on page 473. A simple guideline is to use handles in function definitions and object references as data member types in persistence-capable class definitions.

Interface

The *ooRefHandle*(*appClass*) classes provide the primary interface for operating on a referenced instance of *appClass*.

If *appClass* is a basic-object class, the entire *ooRefHandle*(*appClass*) interface consists of member functions defined by the *ooRefHandle*(ooObj) base classes. These member functions are either inherited by the *ooRefHandle*(*appClass*) classes or redefined wherever type-specific parameters or behavior are required.

If *appClass* is a container class, the *ooRefHandle*(*appClass*) interface provides the same interface as *ooRefHandle*(ooContObj) classes, with a few redefined member functions for type-specific behavior or parameters. This interface includes:

- Public member functions defined by the *ooRefHandle*(ooContObj) classes for specialized container operations, such as finding the basic objects in a container.
- Public member functions defined by the *ooRefHandle*(ooObj) classes for general Objectivity / DB operations, such as opening, locking, printing object identifiers, and so on.

  Note: The member functions defined by the *ooRefHandle*(ooObj) classes for moving, copying, and versioning basic objects are disallowed for containers because the *ooRefHandle*(ooContObj) classes redefine them as private.

If other classes are derived from *appClass*, the *ooRefHandle*(*appClass*) classes themselves serve as base classes for the corresponding handle and object-reference classes. All such derived handle and object-reference classes provide the same interface as *ooRefHandle*(*appClass*), with the usual redefined member functions for type-specific behavior or parameters.

Structure and Behavior

Although both handles and object references provide a way to reference and operate on a persistent object, they are optimized for different purposes:

- Handles are optimized for accessing persistent objects in memory—that is, for performing multiple operations on a referenced object or repeatedly accessing the object's members.
- Object references are optimized for linking persistent objects—that is, for storing object identifiers persistently in reference attributes, in associations, or as elements of a collection.

Handles

Handles are optimized for efficient in-memory access because they can automatically obtain and manipulate pointers to referenced persistent objects. Thus, when a handle is set to reference a particular persistent object, the handle stores the object identifier for that object. The first time the persistent object is accessed through the handle, the handle is automatically *opened*—that is, the handle obtains a pointer to the object's representation in memory. This memory pointer enables the handle to access the referenced object quickly during subsequent operations performed through the handle. When the handle is *closed*, it invalidates the pointer but keeps the object identifier, so the application can reuse the handle (without resetting it) to access the same object.

Besides maintaining a pointer to the referenced persistent object, an open handle also *pins* the object's memory representation in the Objectivity / DB cache. Pinning guarantees that the persistent object is readily available in memory for as long as it is needed. Closing the handle removes its particular "pin"; when the last open handle to that persistent object is closed, the last pin is removed and the object itself is closed. Closing the last (or only) object on a buffer page permits Objectivity / DB to swap the page out of the cache as needed to make room for other open objects.

A handle to a persistent object, like any handle, has cache-related state that is associated with the Objectivity context in which it was created. Therefore, a handle:

- Cannot be stored persistently—for example, as an attribute value. In fact, the DDL processor does not accept ooHandle(*appClass*) as a data member type in a persistence-capable class definition.

- Cannot be passed between Objectivity contexts.

Object References

Object references are optimized for implementing persistent links because they are essentially wrappers for object identifiers. Thus, setting an object reference to a particular persistent object causes the object reference to store the object's identifier. The object reference never acquires a pointer to the persistent object in memory; instead, whenever the persistent object is accessed through the object reference, the operation is delegated to a temporary handle that provides the necessary pointer.

Because it has no bulky cache-specific state, an object reference to a persistent object:

- Can be stored persistently—for example, as an attribute value. The DDL processor accepts `ooRef(`*`appClass`*`)` as a data member type in a persistence-capable class definition.
- Can be passed between Objectivity contexts.

For convenience, an application can use an object reference (instead of a handle) to perform an operation on a referenced persistent object or to access one of the object's members. However, poor performance results when an object reference is used for multiple such operations on the same persistent object, because *each* operation causes a temporary handle to be created, used, and discarded. Performance may also be affected by swapping, because the object reference does not pin the persistent object's memory representation in the Objectivity / DB cache.

An object reference of type `ooRef(`*`appClass`*`)` is sometimes called a *standard object reference* because it contains the complete object identifier for the referenced object. An alternative for referencing a basic object under certain circumstances is the *short object reference* of type `ooShortRef(appClass)`, which saves space by storing object identifiers in a truncated format.

Obtaining Generated Class Definitions

To use the *`ooRefHandle`*`(`*`appClass`*`)` classes, you must include either the primary header file or the references header file generated by the DDL processor for *`appClass`*. Thus, if *`appClass`* is defined in the DDL file *`classDefFile`*`.ddl`, you must include one of the following files:

- The primary header file *`classDefFile`*`.h`
- The references header file *`classDefFile`*`_ref.h`

Furthermore, you must compile the method implementation file *`classDefFile`*`_ddl.cxx` with your application code files.

For more information about DDL-generated files and how to use them, see the Objectivity / C++ Data Definition Language book.

When *appClass* is a Template Class

When *appClass* is a persistence-capable template class with multiple parameters, the names of the generated handle and object-reference classes contain the symbol OO_COMMA to separate the template parameters. For example, for a persistence-capable template class Example<Float, Node>, the generated object-reference class is ooRef(Example<Float OO_COMMA Node>). This is because the macro syntax of the generated class name interprets embedded commas as separators between the as macro parameters instead of as separators between the template parameters.

Working With *appClass* Handles

NOTE For simplicity, this section describes how to work with handles. Except where noted, the same information applies to object references.

An application normally creates a handle as a local variable on the stack, rather than allocating it on the heap. A handle should not be declared as const, because its internal state is changed by any operation that accesses a persistent object through it. (Object references may be declared as const.) An application should not explicitly define subclasses of the *ooRefHandle*(*appClass*) classes; any necessary subclasses are generated automatically by the DDL processor if the application defines any subclasses of *appClass*.

A new handle is normally *null*—that is, it contains the value 0 instead of an object identifier for a persistent object. The application can then set the handle to reference a particular persistent object in any of the following ways:

- By creating a new persistent object with operator new of the *appClass* class and assigning the result to the handle.
- By finding an existing persistent object with the handle's lookupObj member function. (If *appClass* is a container class, you can use the handle's exist or open member function.)
- By passing the handle to a member function that sets it, such as the *linkName* member function of a persistent object that has a *linkName* association. The *linkName* member function finds the associated destination object and sets the specified handle to the found object.
- By assignment or initialization from another handle or object reference.

An object reference may be set in any of these ways, with the following exception—the result of `operator new` may not be assigned to an object reference.

A handle continues to reference the same persistent object until it is set to another persistent object or to null. Furthermore, multiple handles and object references can be set to the same persistent object. A handle of class `ooHandle(`*`appClass`*`)` can be set to an instance of *`appClass`* or any class derived from *`appClass`*.

An application operates on a persistent object by calling:

- Member functions of a handle that references the object. As indicated in "Reference Summary" on page 478, such member functions allow you to copy the referenced object, open it for update, and so on.

  To call a member function of a handle, you use the direct member-access operator (`.`). For example, `appH.lookupObj` calls the `lookupObj` member function of the handle `appH`.
- Member functions of the referenced object itself. As indicated in "Reference Summary" on page 84, these include generated member functions and member functions inherited from `ooObj` or `ooContObj`, as well as member functions defined by *`appClass`*.

  To call a member function of a referenced object, you use the handle's overloaded indirect member-access operator (`operator->`). For example, `appH->ooIsKindOf` calls the `ooIsKindOf` member function on the persistent object that is referenced by the handle `appH`.

Although most of a handle's member functions operate on the referenced persistent object, some functions operate on the handle itself. For example, you use:

- The assignment operator `operator=` to set a handle from another handle or from an object reference.
- The inherited comparison operators `operator==` and `operator!=` to test whether a handle references the same persistent object as another handle or object reference.
- The inherited member function `isNull` to test whether a handle is null. (Alternatively, you can use the overloaded `operator==` to compare a handle to 0.)

A handle preserves its reference to a persistent object across transaction boundaries, provided that the handle does not go out of scope and is not set to null as the result of an abort operation. Before reusing the handle in a new transaction, however, the application should call `isValid` to test whether the handle is still *valid*—that is, whether it still references an existing persistent object. A handle becomes invalid if it is set to null *or* if the referenced persistent object has been deleted by another process between transactions.

Specifying an *appClass* Handle to a Function

Objectivity/C++ functions that require a persistent object as input typically obtain the object through a parameter of type `const ooHandle(ooObj) &`. You can pass a handle of type `ooHandle(`*`appClass`*`)` through such a parameter because `ooHandle(`*`appClass`*`)` is derived from `ooHandle(ooObj)`.

Similarly, Objectivity/C++ functions that require containers typically obtain them through parameters of type of `const ooHandle(ooContObj) &`. If *`appClass`* is a container class, you can pass an *`appClass`* handle through such a parameter.

Opening and Closing an *appClass* Handle

NOTE This subsection applies only to handles, not to object references, which are in effect always closed.

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 object in memory until the handle is closed. A closed handle, which has an object identifier instead of a pointer, can reference either an open or a closed persistent object.

The most common way to open a persistent object through a handle is to do so implicitly by using the handle's indirect member-access operator (`operator->`) to access a member of the referenced object. Alternatively, a referenced persistent object can be opened by explicit request—for example, by calling the handle's `open` or `update` member function. Another way to explicitly open a persistent object is by finding it 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 (for example, because another operation opened it earlier in the transaction), the accessing handle gets a pointer to the existing memory representation and adds a pin.

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 it is set to reference another persistent object, or when the transaction that opened it commits or aborts. An application can close a handle explicitly by calling the handle's `close` member

function. Closing the last open handle to a particular persistent object unpins and closes the object.

Reference Summary

The following table summarizes just the member functions that are redefined by *ooRefHandle*(*appClass*) to provide type-specific parameters. For descriptions of inherited member functions:

- See the ooRefHandle(ooObj) classes (page 593) if *appClass* is a basic object class.
- See the ooRefHandle(ooContObj) classes (page 509) if *appClass* is a container class.

The summarized member functions are defined on both the object-reference class and the handle class. Two operators are defined on only the handle class, namely, operator* and operator appClass*.

Creating a Handle or Object Reference	ooHandle(appClass) ooRef(appClass)
Setting a Handle or Object Reference	operator= lookupObj
Accessing the Referenced Persistent Object	operator->() operator*() operator appClass*() ptr (*ODMG*)
Copying a Basic Object	copy
Testing a Handle	operator appClass*()
Finding a Persistent Object	lookupObj
ODMG Interface	operator d_Ref_Any ptr

Reference Index

Constructors and Destructors

ooHandle(*appClass*)

Default constructor that constructs a null handle.

```
ooHandle(appClass)();
```

ooHandle(*appClass*)

Constructs a handle that references the same persistent object as the specified object reference, handle, pointer, or ODMG generic reference.

```
1.   ooHandle(appClass)(
       const ooRefHandle(appClass) &objectRH);
2.   ooHandle(appClass)(const appClass *objectP);
(ODMG) 3.   ooHandle(appClass)(const d_Ref_Any &from);
```

Parameters

objectRH

Object reference or handle to an instance of *appClass*.

objectP

Pointer to an instance of *appClass*. A pointer *must* be the result of `operator new` on *appClass* or a derived class.

from

(*ODMG*) An ODMG generic reference to an instance of *appClass*. An error is signalled if *from* references an object that is not an instance of *appClass* or a class derived from *appClass*.

Discussion

Variants 1 and 3 allow a new handle to be constructed from an existing object reference, handle, or ODMG generic reference. If the new handle is constructed from an existing open handle, the new handle is open; in all other cases, the new handle is closed.

Variant 2, which constructs a handle from the specified pointer, has a narrower purpose—to obtain an open handle to a newly created persistent object so you can perform persistence operations on it, and so the object can eventually be unpinned when it is no longer needed in memory.

ooRef(*appClass*)

Default constructor that constructs a null object reference.

```
ooRef(appClass)();
```

ooRef(*appClass*)

Constructs an object reference that references the same persistent object as the specified object reference, handle, pointer, or ODMG generic reference.

```
1.   ooRef(appClass)(
       const ooRefHandle(appClass) &objectRH);
2.   ooRef(appClass)(const appClass *objectP);
(ODMG)3.   ooRef(appClass)(const d_Ref_Any &from);
```

Parameters

objectRH

Object reference or handle to an instance of *appClass*.

objectP

Pointer to an instance of *appClass*. The pointer may *not* be the result of `operator new`. Instead, the pointer must be the result of using either `operator appClass*` on a handle *or* `ptr` on an object reference or handle earlier in the same transaction. If the specified pointer was originally extracted from a handle, that handle must still exist and reference the same persistent object.

from

(*ODMG*) An ODMG generic reference to an instance of *appClass*. An error is signalled if *from* references an object that is not an instance of *appClass* or a class derived from *appClass*.

Discussion

Variants 1 and 3 allow a new object reference to be constructed from an existing object reference, handle, or ODMG generic reference.

Variant 2 has a narrower purpose, which is to allow you to resume persistence operations on an existing persistent object after manipulating it through a pointer. The use of this variant should be rare, however, because pointers are not normally used to manipulate persistent objects.

Operators

operator->

Indirect member-access operator; accesses a member of the referenced persistent object.

```
appClass *operator->();
```

Returns

Pointer to the referenced instance of *appClass*.

Discussion

The accessed persistent object is opened for read, if it is not already open.

If the referenced object is an instance of *appClass*, `operator->` can access any of the object's public members. If the referenced object is an instance of a derived class, `operator->` accesses only the members that are defined in *appClass*.

You use `operator->` in an expression *handle->member*, where *handle* is an instance of `ooHandle(`*appClass*`)` and *member* is the name of a public member defined on class *appClass*. As for any overloading of the C++ member-access operator (->), the expression *handle->member* is interpreted as (*handle*`.operator->())->`*member*. That is, the overloaded `operator->` returns a pointer to the referenced object, and then the ordinary C++ `operator->` selects the specified member of that object, returning the value of that member.

Although you can obtain the returned pointer (for example, from the `this` keyword in an accessed member function), you should view the pointer as an intermediate by-product of the access operation. The same is true for pointers to accessed data members obtained through expressions such as `&(`*handle->dataMember*`)`. You should not use such pointers in any further operations (for example, do not assign the returned pointer to a handle, object reference, or pointer variable).

Warning: The pointer returned by `operator->` is guaranteed valid for only a limited time:

- When you access an object through an object reference, the returned pointer is guaranteed valid only for the duration of the `->` operation.
- When you access an object through a handle, the returned pointer is guaranteed valid only as long as the handle exists and references the same object.

operator*

Handle class only. Dereference operator; returns the persistent object referenced by this handle.

```
appClass &operator*();
```

Returns

C++ reference to the persistent object referenced by this handle.

Discussion

This operator enables you to pass a handle to a function that accepts a persistent object by reference. This operator is analogous to the C++ `operator*` for dereferencing a pointer.

The persistent object referenced by this handle is opened for read, if it is not already open.

WARNING The returned reference is guaranteed valid only as long as the handle exists, remains open, and references the same object.

Example Assume that `Book` is a persistence-capable class. You use `operator*` to pass a book handle to `helperFunction`, which accepts a C++ reference to a `Book`.

```
void helperFunction(Book &aBook);

void processBook(ooHandle(Book) &bookH) {
   ...
   helperFunction(*bookH);
   ...
}
```

operator=

Assignment operator; sets this object reference or handle to reference the same persistent object as the specified object reference, handle, or pointer.

```
1.  ooRefHandle(appClass) &operator=(
      const ooRefHandle(appClass) &objectRH);
2.  ooRefHandle(appClass) &operator=(
      const ooShortRef(appClass) &shortObjR);
3.  ooRefHandle(appClass) &operator=(
      const appClass *objectP);
```

Parameters

objectRH

Object reference or handle to an instance of *appClass*.

shortObjR

Short object reference to an instance of *appClass* that resides in the same container as the object that is referenced by this object reference or handle. (If this object reference or handle is null, you can set it to a container with the `set_container` member function.)

A short object reference specifies just the lower half of an object identifier (corresponding to the object's logical page and slot numbers). The upper half of the object identifier (corresponding to the database and container) is taken from this object reference or handle.

objectP

0, or a nonnull pointer to an instance of *appClass*.

- If you are assigning to a handle, the specified pointer *must* be the result of `operator new` on *appClass* or a derived class.

- If you are assigning to an object reference, the specified pointer may *not* be the result of `operator new`. Instead, the pointer must be the result of using either `operator appClass*` on a handle *or* `ptr` on an object reference or handle earlier in the same transaction. If the specified pointer was originally extracted from a handle, that handle must still exist and reference the same persistent object.

Returns — This object reference or handle, after it has been set to reference the specified object.

Discussion — Variants 1 and 2 allow you to use the specified object reference or handle to produce another object reference or handle to the same persistent object. If you are assigning to a handle from an open handle, the returned handle is open; in all other cases, the returned handle is closed.

Variant 3 allows you to set this object reference or handle to null. Otherwise, assignment-from-pointer has two specific purposes, depending on whether you are assigning to a handle or to an object reference:

- Pointer-to-handle assignment enables you to obtain an open handle to a newly created persistent object, so you can perform persistence operations on it, and so the object can eventually be unpinned when it is no longer needed in memory.
- Pointer-to-object-reference assignment enables you to resume persistence operations on a persistent object after manipulating it through a pointer. This usage of variant 3 is rare, because pointers are not normally used to manipulate persistent objects.

operator d_Ref_Any

(*ODMG*) Conversion operator that returns an ODMG generic reference to the referenced persistent object.

```
operator d_Ref_Any() const;
```

operator *appClass**

Handle class only. Conversion operator that returns an *`appClass`* pointer to the referenced persistent object.

```
operator appClass*();
```

Returns — Pointer to the persistent object referenced by this handle. Returns a null pointer if the handle is a null handle.

Discussion — This conversion operator enables you to:

- Pass a handle to a function that accepts a pointer to a persistent object.
- Assign a handle to an *appClass** variable (for example, to pass to the overloaded operator delete).
- Use a handle as the conditional expression in an `if` or `while` statement to test whether the handle is null.

The persistent object referenced by this handle is opened for read, if it is not already open.

WARNING The returned pointer is guaranteed valid only as long as this handle exists, remains open, and references the same object.

An application generally relies on handles to provide memory management for persistent objects, and avoids the explicit use of pointers to such objects. On occasion, explicit use of pointers is required for performance reasons or for compatibility with functions that are indifferent to persistence (for example, in legacy code or a third-party library). However, you should be careful when manipulating a pointer from a handle because the validity of the pointer depends on the state of the handle. You should not use the returned pointer in other persistence operations (for example, do not pass it to any Objectivity/C++ member function other than the overloaded operator delete).

Example

The handle `objectH` is used as a conditional expression which evaluates to 0 if the handle is null.

```
ooHandle(myClass) objectH;
... // Set objectH to some object
if (objectH) {
... // Do something interesting if initialization was successful
}
```

See also

ptr
ooRef(ooObj)::operator int

Member Functions

copy

Creates a copy of the referenced basic object, clustering the new copy near the specified object.

```
1.  ooHandle(appClass) copy(
      const ooHandle(ooObj) &near) const;
2.  ooRef(appClass) &copy(
      const ooHandle(ooObj) &near,
      ooRef(appClass) &newCopy) const;
3.  ooHandle(appClass) &copy(
      const ooHandle(ooObj) &near,
      ooHandle(appClass) &newCopy) const;
```

Parameters

near

Handle to the object with which to cluster the new copy. *near* may be a handle to a database, a container, or a basic object:

- If *near* is a database handle, the new copy is stored in the default container of that database.
- If *near* is a container handle, the new copy is stored in that container.
- If *near* is a basic object handle, the new copy is stored in the same container as the referenced basic object. If possible, the copy will be put on the same page as the referenced basic object or on a nearby page.

newCopy

Object reference or handle to set to the new copy.

Returns

`oocSuccess` if successful; otherwise `oocError`.

Discussion

Copying applies only to basic objects. Therefore, you should call this member function only if *appClass* is derived from `ooObj` but not `ooContObj`. This member function signals an error if you attempt to copy a container.

When called without a *newcopy* parameter, `copy` allocates a new handle and returns it. Otherwise, `copy` returns the object reference or handle that is passed to it.

The application must be able to lock the container of the original object for read and the container of the new copy for update.

lookupObj

Finds the persistent object with the specified scope name (or the basic object matching the specified key structure) within the specified scope, and sets this object reference or handle to reference the found object.

```
1.  ooStatus lookupObj (
      const ooHandle(ooObj) &scope,
      const char *scopeName,
      const ooMode openMode = oocRead);
2.  ooStatus lookupObj (
      const ooHandle(ooObj) &scope,
      const ooKey &keyStruct,
      const ooMode openMode = oocRead);
```

Parameters

scope

Handle to an object that defines the scope of the lookup:

- (Variant 1) When the lookup is by scope name, *scope* specifies the scope object that defines the name scope to search. *scope* can reference the federated database, a database, a persistent container, a persistent basic object, or an autonomous partition.
- (Variant 1) When the lookup is by key structure, *scope* specifies the container to search. *scope* can reference the container itself or the database whose default container is to be searched.

scopeName

Scope name to look up in the scope specified by *scope*.

openMode

Intended level of access to the found object:

- Specify `oocRead` (the default) to open the object for read.
- Specify `oocUpdate` to open the object for update.
- (Variant 2 only) Specify `oocNoOpen` to set this object reference or handle to the object without opening it. `oocNoOpen` is valid only for scope-name lookup, because scope-named objects can be found without being opened, whereas keyed objects must be opened during the search.

keyStruct

Key structure specifying the key field and key field value to match.

Returns

`oocSuccess` if an object of the appropriate class is found; otherwise `oocError`.

Discussion

Scope-name lookup applies either to basic objects or containers. Therefore, you can call variant 1 regardless of whether *appClass* is derived from `ooObj` or `ooContObj`.

Keyed-object lookup applies only to basic objects. Therefore, you should call variant 2 only if *appClass* is derived from ooObj (but not ooContObj).

In any case, an object is found only if it is an instance of *appClass* or a class derived from *appClass*.

The application must be able to obtain a read lock on the hashed container used by the scope object.

ptr

(*ODMG*) Returns a C++ pointer to the referenced persistent object.

```
appClass *ptr();
```

Returns

Pointer to the referenced basic object or container.

Discussion

You use this member function to obtain a pointer to an application-defined basic object or container—for example, to pass to a function that accepts a pointer instead of a handle or object reference.

If ptr is called on an object reference, the referenced persistent object is opened for update. If ptr is called on a handle, the referenced persistent object is opened for read.

Warning: The returned pointer is guaranteed valid for only a limited time:

- If ptr is called on an object reference, the returned pointer is valid and the persistent object is pinned in memory until the end of the transaction.
- If ptr is called on a handle, the returned pointer is valid only as long as the handle exists, remains open, and references the same persistent object (equivalent to operator appClass*).

An application generally relies on handles to provide memory management for persistent objects, and avoids the explicit use of pointers to such objects. On occasion, explicit use of pointers is required for performance reasons or for compatibility with functions that are indifferent to persistence (for example, in legacy code or a third-party library). However, you should be careful when manipulating persistent objects through pointers:

- Pointers extracted from handles become invalid if the handles change or go out of scope.
- Pointers extracted from object references can cause the Objectivity / DB cache to run out of memory if too many objects are pinned until the end of the transaction.

You should not use the returned pointer in other persistence operations (for example, do not pass it to any Objectivity / C++ member function other than the overloaded operator delete).

ooRefHandle(ooAPObj) Classes

Inheritance: **ooRef(ooObj)->ooRef(ooAPObj)**

Inheritance: **ooHandle(ooObj)->ooHandle(ooAPObj)**

The abbreviation *ooRefHandle*(ooAPObj) refers to two non-persistence-capable classes:

- ooRef(ooAPObj), which represents an *object reference* to an autonomous partition.
- ooHandle(ooAPObj), which represents a *handle* to an autonomous partition.

The two classes ooRef(ooAPObj) and ooHandle(ooAPObj) are documented together because they define the same set of member functions. These member functions provide the primary interface for operating on autonomous partitions (instances of ooAPObj).

You can create and work with autonomous partitions only if you have bought and installed Objectivity / DB Fault Tolerant Option (Objectivity / FTO).

See:

- “Reference Summary” on page 493 for an overview of member functions
- “Reference Index” on page 494 for a list of member functions

About Autonomous-Partition Handles and References

An application works with an autonomous partition indirectly through one or more handles or object references—that is, through instances of *ooRefHandle*(ooAPObj) that are set to reference the desired partition. A handle or object reference to an autonomous partition both identifies the partition and provides the complete public interface for operating on it.

Handles and object references to autonomous partitions do not support indirect member access; the *ooRefHandle*(ooAPObj) classes provide no indirect member-access operator (->), and the ooAPObj class defines no public member functions other than a constructor and operator new.

You can work with an autonomous partition through either a handle or an object reference—the choice is arbitrary, except as described in "Structure and Behavior" on page 490. Most applications use handles rather than object references.

Interface

The *ooRefHandle*(ooAPObj) classes provide the primary interface for operating on a referenced autonomous partition. Part of this interface consists of member functions defined by *ooRefHandle*(ooAPObj) for specialized operations such as finding the containers that a partition controls. The other part of this interface consists of member functions defined by the *ooRefHandle*(ooObj) base classes for more general Objectivity / DB operations, such as opening, printing object identifiers, and so on. These member functions are either inherited by the *ooRefHandle*(ooAPObj) classes or redefined wherever type-specific parameters or behavior are required.

Some of the member functions defined by the *ooRefHandle*(ooObj) base classes are *not* available to instances of the *ooRefHandle*(ooAPObj) classes. These include member functions for moving, copying, versioning, scope-naming, and member-access operations, which apply only to basic objects or persistent objects. The disallowed member functions and operators are redefined as private members of the *ooRefHandle*(ooAPObj) classes.

Structure and Behavior

Handles and object references to autonomous partitions are essentially wrappers for autonomous-partition identifiers. For example, when you set a handle to reference a particular partition, the handle stores the object identifier of that partition. If the partition is then opened through the handle, Objectivity / DB uses the identifier to locate the partition's system-database file on disk. Subsequent member-function calls on the handle are applied to the instance of ooAPObj that represents the identified partition in memory.

In general, object references are optimized for implementing links among related persistent objects, while handles are optimized for memory management and member-access. When an autonomous partition is referenced, however, these optimizations are largely irrelevant, because partitions (unlike persistent objects):

- Cannot be linked (for example, through associations).
- Are not subject to memory management (they have no attributes for persistent data and are therefore not manipulated through pointers).

- Have no accessible members.

One significant exception is that a handle to an autonomous partition, like any handle, contains cache-related state that is associated with the Objectivity context in which it was created. Therefore, only object references (but not handles) can be passed between Objectivity contexts. Otherwise, a handle to a partition and an object reference to a partition are functionally equivalent.

Working With Autonomous-Partition Handles

NOTE For simplicity, this section describes how to work with handles. Except where noted, the same information applies to object references.

An application normally creates a handle as a local variable on the stack, rather than allocating it on the heap. A handle should not be declared as `const`, because its internal state may be changed when an autonomous partition is accessed through it. (Object references may be declared as `const`). Applications should not create subclasses of the *`ooRefHandle`*`(ooAPObj)` classes.

A new handle is normally *null*—that is, it contains the value 0 instead of the object identifier for an autonomous partition. The application can then set the handle to reference a particular partition in any of the following ways:

- By creating a new partition with `operator new` of the `ooAPObj` class and assigning the result to the handle.
- By finding an existing partition with the handle's `exist` or `open` member function.
- By passing the partition to a member function that sets it, such as the `controlledBy` member function of a container handle, which finds the partition that controls the container.
- By assignment or initialization from another handle or object reference.

A handle continues to reference the same autonomous partition until it is set to another partition or to null. Furthermore, multiple handles and object references can be set to the same partition.

An application operates on an autonomous partition by calling member functions on a handle that references it. To call a member function of a handle, you use the direct member-access operator (`.`). For example, `apH.name` calls the `change` member function of the handle `apH`.

As indicated in "Reference Summary" on page 493, an application can get and change a referenced partition's attributes, find its containers and database

images, and so on. For more information about operating on autonomous partitions, see Chapter 27, "Autonomous Partitions," in the Objectivity/C++ programmer's guide.

Although most of a partition handle's member functions operate on the referenced partition, some functions operate on the handle itself. For example, you use:

- The assignment operator `operator=` to set a handle from another handle or from an object reference.
- The inherited comparison operators `operator==` and `operator!=` to test whether a handle references the same partition as another handle or object reference.
- The inherited member function `isNull` to test whether a handle is null. (Alternatively, you can use the overloaded `operator==` to compare a handle to 0.)

A handle preserves its reference to an autonomous partition across transaction boundaries, provided that the handle does not go out of scope and is not set to null as the result of an abort operation. Before reusing the handle in a new transaction, however, the application should call `isValid` to test whether the handle is still *valid*—that is, whether it still references an existing autonomous partition. A handle becomes invalid if it is set to null *or* if the referenced partition has been deleted by another process between transactions.

Objectivity/C++ functions that require an autonomous partition as input normally obtain the partition through a parameter of type `const ooHandle(ooAPObj) &`. If a function manipulates other types of Objectivity/DB objects in addition to autonomous partitions, the parameter type may be specified as `const ooHandle(ooObj) &`. You can pass a partition handle to a parameter of this type, because `ooHandle(ooAPObj)` is derived from `ooHandle(ooObj)`. In practice, however, relatively few functions accept a partition handle where a more general handle is requested; these include `ooDelete` and functions that require a storage object for a name scope. In most cases, a function that requests a general-purpose handle operates only on basic objects or persistent objects, and signals an error if you specify a partition handle.

Any operation that affects an autonomous partition implicitly opens it if it is not already open; an application does not need to open an autonomous partition explicitly unless it is used as the entry point into the data or unless access must be guaranteed in advance. A handle is automatically closed when it is destroyed (for example, when it goes out of scope). However, closing one or more handles to a particular partition has no effect on that partition, which remains open until the transaction commits or aborts.

Reference Summary

The following table summarizes all the member functions that are available to instances of *ooRefHandle*(ooAPObj). Member functions indicated as *(inherited)* documented with the ooRefHandle(ooObj) classes (page 593).

Creating a Handle or Object Reference	ooHandle(ooAPObj) ooRef(ooAPObj)
Setting the Handle or Object Reference	operator= open exist
Comparing Handles and Object References	operator== *(inherited)* operator!= *(inherited)*
Opening and Closing the Autonomous Partition	open update openMode close
Modifying the Autonomous Partition	update change markOffline markOnline returnAll
Getting Information About the Autonomous Partition	name bootFileHost bootFilePath jnlDirHost jnlDirPath lockServerHost sysDBFileHost sysDBFilePath typeN typeName print *(inherited)* sprint *(inherited)*
Testing the Autonomous Partition	exist isAvailable isOffline

Testing the Handle or Object Reference	is_null *(inherited)* isNull *(inherited)* isValid operator int *(inherited)* operator ooObj* *(inherited)*
Finding Objects	exist containedIn containersControlledBy imagesContainedIn (*DRO*)

Reference Index

isOffline	Tests whether the referenced autonomous partition is offline (inaccessible to tools and applications).
isValid	Tests whether this object reference or handle is valid—that is, whether it references an existing autonomous partition.
jnlDirHost	Gets the network name of the host that contains the journal directory for the referenced autonomous partition.
jnlDirPath	Gets the fully qualified pathname of the journal directory for the referenced autonomous partition.
lockServerHost	Gets the network name of the host running the lock server for the referenced autonomous partition.
markOffline	Makes the referenced autonomous partition inaccessible to tools and applications.
markOnline	Makes the referenced autonomous partition accessible to tools and applications.
name	Gets the system name of the referenced autonomous partition.
ooHandle(ooAPObj)	Default constructor that constructs a null handle.
ooHandle(ooAPObj)	Constructs a handle that references the same autonomous partition as the specified object reference or handle.
ooRef(ooAPObj)	Default constructor that constructs a null object reference.
ooRef(ooAPObj)	Constructs an object reference that references the same autonomous partition as the specified object reference or handle.
open	Explicitly opens the referenced or specified autonomous partition, preparing the partition for the specified level of access.
openMode	Gets the current level of access to the referenced autonomous partition.
operator=	Assignment operator; sets this object reference or handle to reference the specified autonomous partition.

returnAll	Clears the referenced autonomous partition, returning all controlled containers to their home autonomous partitions.
sysDBFileHost	Gets the network name of the host containing the system-database file of the referenced autonomous partition.
sysDBFilePath	Gets the fully qualified pathname of the system-database file of the referenced autonomous partition.
typeN	Gets the type number of the autonomous-partition class ooAPObj.
typeName	Gets the name of the autonomous-partition class ooAPObj.
update	Opens the referenced autonomous partition for update access.

Constructors

ooHandle(ooAPObj)

Default constructor that constructs a null handle.

```
ooHandle(ooAPObj)();
```

ooHandle(ooAPObj)

Constructs a handle that references the same autonomous partition as the specified object reference or handle.

```
ooHandle(ooAPObj)(
     const ooRefHandle(ooAPObj) &existing);
```

Parameters *existing*

Object reference or handle to an existing autonomous partition.

ooRef(ooAPObj)

Default constructor that constructs a null object reference.

```
ooRef(ooAPObj)();
```

ooRef(ooAPObj)

Constructs an object reference that references the same autonomous partition as the specified object reference or handle.

```
ooRef(ooAPObj)(
     const ooRefHandle(ooAPObj) &existing);
```

Parameters *existing*

Object reference or handle to an existing autonomous partition.

Operators

operator=

Assignment operator; sets this object reference or handle to reference the specified autonomous partition.

```
ooRefHandle(ooAPObj) &operator=(
     const ooRefHandle(ooAPObj) &existing);
```

Parameters *existing*

Object reference or handle to an existing autonomous partition.

Returns This object reference or handle.

Member Functions

bootFileHost

Gets the network name of the data server host containing the boot file for the referenced autonomous partition.

```
char *bootFileHost() const;
```

Returns Pointer to a string containing the network name of the data server host. If an error occurs, a null pointer is returned and the error is signalled.

Discussion The returned string is statically allocated by the member function and is overwritten with each invocation. You should make a local copy of the returned string if you intend to use it later in the application.

bootFilePath

Gets the fully qualified pathname of the boot file for the referenced autonomous partition.

```
char *bootFilePath() const;
```

Returns Pointer to a string containing the pathname of the boot file. If an error occurs, a null pointer is returned and the error is signalled.

Discussion The returned string is statically allocated by the member function and is overwritten with each invocation. You should make a local copy of the returned string if you intend to use it later in the application.

change

Changes catalog information for the referenced autonomous partition.

```
ooStatus change(
     const char *lockServer = 0,
     const char *apFileHost = 0,
     const char *apFilePath = 0,
     const char *bootFileHost = 0,
     const char *bootFilePath = 0,
     const char *jnlDirHost = 0,
     const char *jnlDirPath = 0) const;
```

Parameters *lockServer*

Name of the new lock server host (the host that runs the lock server for this autonomous partition). Specify 0 (the default) to leave the lock server host unchanged.

apFileHost

Name of the data server host on which the autonomous partition's system-database file is to reside. Specify 0 (the default) to leave the autonomous-partition host unchanged. If you specify this parameter, you must also specify a nondefault value for *apFilePath*.

apFilePath

New pathname for the autonomous partition's system-database file on *apFileHost*, including the filename. Specify 0 (the default) to leave the autonomous-partition pathname unchanged.

bootFileHost

Name of the new data server host on which the autonomous-partition boot file is to reside. Specify 0 (the default) to leave the boot file host unchanged.

If you specify this parameter, you must also specify a nondefault value for *bootFilePath*.

bootFilePath

New pathname for the autonomous-partition boot file on *bootFilePath*, including the filename. Specify 0 (the default) to leave the boot file pathname unchanged.

jnlDirHost

New host machine for the autonomous-partition journal directory. Specify 0 (the default) to leave the journal-directory host unchanged. If you specify this parameter, you must also specify a nondefault value for *jnlDirPath*.

jnlDirPath

New path for the new autonomous-partition journal directory. Specify 0 (the default) to leave the journal-directory pathname unchanged.

Returns `oocSuccess` if successful; otherwise `oocError`.

Discussion You cannot change the system name of an autonomous partition. If you specify a new boot file location, the updated boot file is written to the new location, but the old boot file remains; you must delete this file using appropriate operating system commands. If you specify a new location for the autonomous partition's system-database file, the catalog is updated; however, you must use appropriate operating system commands to actually move the file.

You use this member function in a special-purpose application that consists of a single update transaction. The application must exit immediately after the transaction commits. This is because the new state of the autonomous partition is inconsistent with information cached by the executing application.

close

Internal use only. Objectivity / DB closes an autonomous partition automatically when the transaction that opened it commits or aborts.

```
ooStatus close() const;
```

containedIn

Finds the federated database that contains the referenced autonomous partition.

```
1.  ooHandle(ooFDObj) containedIn() const;
2.  ooRef(ooFDObj) &containedIn(
      ooRef(ooFDObj) &returnedFD) const;
```

```
3.   ooHandle(ooFDObj) &containedIn(
       ooHandle(ooFDObj) &returnedFD) const;
```

Parameters

returnedFD

Object reference or handle to be set to the federated database.

Returns

Object reference or handle to the federated database.

Discussion

When called without a parameter, `containedIn` allocates a new federated-database handle and returns it. Otherwise, `containedIn` returns the object reference or handle that was passed to it.

containersControlledBy

Initializes an object iterator to find the containers that are controlled by the referenced autonomous partition.

```
ooStatus containersControlledBy(
     ooItr(ooContObj) &controlledConts,
     const ooMode openMode = oocNoOpen) const;
```

Parameters

controlledConts

Object iterator for finding the controlled containers.

openMode

Intended level of access to the containers found by the iterator's `next` member function:

- `oocNoOpen` (the default) causes `next` to set the iterator to the next container without opening it.
- `oocRead` causes `next` to open the next container for read.
- `oocUpdate` causes `next` to open the next container for update.

Returns

`oocSuccess` if successful; otherwise `oocError`.

Discussion

The iterator finds only those containers that were explicitly transferred to the control of the referenced autonomous partition.

exist

Tests whether the specified autonomous partition exists in the federated database; if successful, sets this object reference or handle to reference the partition.

```
ooBoolean exist(
     const ooHandle(ooFDObj) &containingFD,
     const char *apSysName,
     const ooMode openMode = oocNoOpen);
```

Parameters

containingFD

Handle to the federated database.

apSysName

System name of the autonomous partition to be found.

openMode

Intended level of access to the autonomous partition, if it exists:

- Specify `oocNoOpen` (the default), to set this object reference or handle to the autonomous partition without opening it.
- Specify `oocRead` to open the autonomous partition for read.
- Specify `oocUpdate` to open the autonomous partition for update.

Returns

`oocTrue` if the specified autonomous partition exists, or `oocFalse` if the database does not exist or if it is not accessible.

Discussion

If the specified autonomous partition exists, this object reference or handle is set to reference it; otherwise, this object reference or handle is set to null.

If you specifically want to test for existence, you use the *openMode* parameter's default value (*oocNoOpen*). Otherwise, a return value of *oocFalse* could mean either that the partition doesn't exist, or that it does exist, but cannot be opened.

imagesContainedIn

(*DRO*) Initializes an object iterator to find all the database images that are contained in the referenced autonomous partition.

```
ooStatus imagesContainedIn(
     ooItr(ooDBObj) &containedDbs,
     const ooMode openMode = oocNoOpen) const;
```

Parameters

containedDBs

Object iterator for finding the database images in the autonomous partition.

openMode

Intended level of access to the database images found by the iterator's `next` member function:

- `oocNoOpen` (the default) causes `next` to set the iterator to the next database image without opening it.
- `oocRead` causes `next` to open the next database image for read.
- `oocUpdate` causes `next` to open the next database image for update.

Returns `oocSuccess` if successful; otherwise `oocError`.

isAvailable

Tests whether the current process can access the referenced autonomous partition.

```
ooBoolean isAvailable() const;
```

Returns `oocTrue` if the partition is available to the current process; otherwise `oocFalse`. This information can change at any time during the process.

isOffline

Tests whether the referenced autonomous partition is offline (inaccessible to tools and applications).

```
ooBoolean isOffline() const;
```

Returns `oocTrue` if the referenced autonomous partition is offline.

See also `markOffline`
`markOnline`
`ooOfflineMode` global type
`ooGetOfflineMode` global function
`ooSetOfflineMode` global function

isValid

Tests whether this object reference or handle is valid—that is, whether it references an existing autonomous partition.

```
ooBoolean isValid() const;
```

Returns `oocTrue` if the object reference or handle references an existing autonomous partition; `oocFalse` if this object reference or handle is null or has a stale identifier.

Discussion

You can use `isValid` to determine whether it is safe to use an object reference or handle that was set in a previous transaction. Such an object reference or handle still retains its reference to a partition; however, between transactions, that reference may have become invalid (for example, because another process has deleted the partition).

NOTE `isValid` checks only for the existence of a partition with a particular identifier, but has no way of knowing whether it is the same partition. It is possible, although very unlikely, for another process to have deleted the original partition and created a new one with the same identifier.

If your purpose is simply to test whether an object reference or handle has been initialized, it is more efficient to use `isNull`, which performs its test entirely in memory without having to access files on disk.

jnlDirHost

Gets the network name of the host that contains the journal directory for the referenced autonomous partition.

```
char *jnlDirHost() const;
```

Returns

Pointer to a string containing the journal-directory host name. If an error occurs, a null pointer is returned and the error is signalled.

Discussion

The string is statically allocated by the member function and is overwritten with each invocation. You should make a local copy of the returned string if you intend to use it later in the application.

jnlDirPath

Gets the fully qualified pathname of the journal directory for the referenced autonomous partition.

```
char *jnlDirPath() const;
```

Returns

Pointer to a string containing the journal-directory pathname.

Discussion

The string is statically allocated by the member function and is overwritten with each invocation. You should make a local copy of the returned string if you intend to use it later in the application.

lockServerHost

Gets the network name of the host running the lock server for the referenced autonomous partition.

```
char *lockServerHost() const;
```

Returns Pointer to a string containing the lock server host name. If an error occurs, a null pointer is returned.

Discussion The string is statically allocated by the member function and is overwritten with each invocation. You should make a local copy of the returned string if you intend to use it later in the application.

markOffline

Makes the referenced autonomous partition inaccessible to tools and applications.

```
ooStatus markOffline() const;
```

Returns `oocSuccess` if successful; otherwise `oocError`.

Discussion This member function must be invoked in a transaction that includes no other operations.

See also `isOffline`
`ooOfflineMode` global type
`ooGetOfflineMode` global function
`ooSetOfflineMode` global function

markOnline

Makes the referenced autonomous partition accessible to tools and applications.

```
ooStatus markOnline() const;
```

Returns `oocSuccess` if successful; otherwise `oocError`.

See also `isOffline`
`ooOfflineMode` global type
`ooGetOfflineMode` global function
`ooSetOfflineMode` global function

name

Gets the system name of the referenced autonomous partition.

```
char *name() const;
```

Returns Pointer to a string containing the system name.

Discussion The string is statically allocated by the member function and is overwritten with each invocation. You should make a local copy of the returned string if you intend to use it later in the application.

open

Explicitly opens the referenced or specified autonomous partition, preparing the partition for the specified level of access.

```
1.  ooStatus open(
      const ooMode openMode = oocRead);
2.  ooStatus open(
      const ooHandle(ooFDObj) &containingFD,
      const char *apSysName,
      const ooMode openMode = oocRead);
```

Parameters *openMode*

Intended level of access to the opened autonomous partition:

- Specify `oocRead` (the default) to open and implicitly lock the partition for read.
- Specify `oocUpdate` to open and implicitly lock the partition for update (read and write).

containingFD

Handle to the currently open federated database.

apSysName

System name of the autonomous partition to open.

Returns `oocSuccess` if successful; otherwise `oocError`.

Discussion Variant 1 opens the autonomous partition referenced by this object reference or handle.

Variant 2 finds and opens the autonomous partition with the specified system name, and sets this object reference or handle to reference it. An error is signalled if no partition exists with the specified system name or if the partition's system-database file cannot be found or accessed. This variant is especially useful

when you want to use the partition as an entry point into your data. For example, you might find and open a partition so you can iterate over the containers it controls.

Opening an autonomous partition makes it available to an application by locating and opening the partition's system-database file, provided that appropriate access permissions are set on it.

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 reference 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.

Any number of transactions can concurrently open the same partition in any mode.

You must be in an update transaction to open a partition for update. If necessary, you can promote a read transaction to an update transaction by promoting the open mode of the federated database.

openMode

Gets the current level of access to the referenced autonomous partition.

```
ooMode openMode() const;
```

Returns One of the following constants:

- `oocNoOpen`—the partition is not open in this transaction or an error has been signalled.
- `oocRead`—the partition is open for read in this transaction.
- `oocUpdate`—the partition is open for update in this transaction.

returnAll

Clears the referenced autonomous partition, returning all controlled containers to their home autonomous partitions.

```
ooStatus returnAll() const;
```

Returns `oocSuccess` if successful; otherwise `oocError`.

Discussion Clearing an autonomous partition clears it of the containers it controls. Specifically:

- Each container is returned to the control of the autonomous partition that contains the container's database.

- Each container is physically moved from the autonomous partition's system-database file to the file of the container's database.

sysDBFileHost

Gets the network name of the host containing the system-database file of the referenced autonomous partition.

```
char *sysDBFileHost() const;
```

Returns Pointer to a string containing the network name of the data server host. If an error occurs, a null pointer is returned and the error is signalled.

Discussion The returned string is statically allocated by the member function and is overwritten with each invocation. You should make a local copy of the returned string if you intend to use it later in the application.

sysDBFilePath

Gets the fully qualified pathname of the system-database file of the referenced autonomous partition.

```
char *sysDBFilePath() const;
```

Returns Pointer to a string containing the pathname of the autonomous partition's system-database file. If an error occurs, a null pointer is returned and the error is signalled.

Discussion The returned string is statically allocated by the member function and is overwritten with each invocation. You should make a local copy of the returned string if you intend to use it later in the application.

typeN

Gets the type number of the autonomous-partition class `ooAPObj`.

```
ooTypeNumber typeN() const;
```

Returns Type number of the autonomous-partition class `ooAPObj`.

typeName

Gets the name of the autonomous-partition class `ooAPObj`.

```
char *typeName() const;
```

Returns The string "`ooAPObj`".

Discussion The returned string must be treated as read only.

update

Opens the referenced autonomous partition for update access.

```
ooStatus update();
```

Returns `oocSuccess` if successful; otherwise `oocError`.

Discussion This member function is equivalent to calling open(`oocUpdate`).

ooRefHandle(ooContObj) Classes

Inheritance: **ooRef(ooObj)->ooRef(ooContObj)**

Inheritance: **ooHandle(ooObj)->ooHandle(ooContObj)**

The abbreviation *ooRefHandle*(ooContObj) refers to two non-persistence-capable classes:

- ooRef(ooContObj), which represents an *object reference* to a container.
- ooHandle(ooContObj), which represents a *handle* to a container.

These two classes are documented together because they define almost identical sets of member functions (exceptions are listed in the "Reference Summary"). These classes, along with the ooContObj class, define the behavior of all containers (instances of ooContObj and its derived classes).

See:

- "Reference Summary" on page 515 for an overview of member functions
- "Reference Index" on page 517 for a list of member functions

About Container Handles and References

An application works with a container indirectly through one or more handles or object references—that is, through instances of *ooRefHandle*(ooContObj) that are set to reference the desired container. A handle or object reference to a container serves as a type-safe smart pointer that:

- Identifies the container to the application or to another object.
- Provides an interface for operating on the container.
- Manages the memory pointer to the container.
- Provides an indirect member-access operator (->) for accessing the container's public member functions.

It is sometimes more appropriate to use a handle rather than an object reference, and vice versa; the choice is described in “Structure and Behavior” on page 510. A simple guideline is to use handles in function definitions and object references as data member types in persistence-capable class definitions.

Interface

The *ooRefHandle*(ooContObj) classes provide the primary interface for operating on a referenced container. Part of this interface consists of member functions defined by *ooRefHandle*(ooContObj) for specialized operations such as finding the basic objects in a container. The other part of this interface consists of member functions defined by the *ooRefHandle*(ooObj) base classes for more general Objectivity / DB operations, such as opening, locking, printing object identifiers, and so on. These member functions are either inherited by the *ooRefHandle*(ooContObj) classes or redefined wherever type-specific parameters or behavior are required.

Some of the member functions defined by the *ooRefHandle*(ooObj) base classes are *not* available to instances of the *ooRefHandle*(ooContObj) classes. These include member functions for moving, copying, and versioning operations, which apply only to basic objects. The disallowed member functions and operators are redefined as private members of the *ooRefHandle*(ooContObj) classes.

The *ooRefHandle*(ooContObj) classes themselves serve as base classes for other handle and object-reference classes. Some of these derived classes, such as *ooRefHandle*(ooGCContObj) and *ooRefHandle*(ooDefaultContObj), are predefined by Objectivity / C++. The other derived handle and object-reference classes—namely, the *ooRefHandle*(*appClass*) classes—are generated by the DDL processor for every application-defined container class *appClass*. All of the derived handle and object-reference classes provide the same interface as *ooRefHandle*(ooContObj), with a few redefined member functions for type-specific behavior or parameters; see ooRefHandle(appClass) (page 509).

Structure and Behavior

Although both handles and object references provide a way to reference and operate on a container, they are optimized for different purposes:

- Handles are optimized for accessing persistent objects in memory—that is, for performing multiple operations on a referenced object or repeatedly accessing the object’s members.
- Object references are optimized for linking persistent objects—that is, for storing object identifiers persistently in reference attributes, in associations, or as elements of a collection.

Handles

Handles are optimized for efficient in-memory access because they can automatically obtain and manipulate pointers to referenced objects. Thus, when a handle is set to reference a particular container, the handle stores the object identifier for that container. The first time the container is accessed through the handle, the handle is automatically *opened*—that is, the handle obtains a pointer to the container's representation in memory. This memory pointer enables the handle to access the referenced container quickly during subsequent operations performed through the handle. When the handle is *closed*, it invalidates the pointer but keeps the container's object identifier, so the application can reuse the handle (without resetting it) to access the same container.

Besides maintaining a pointer to the referenced container, an open handle also *pins* the container's memory representation in the Objectivity / DB cache; that is, the handle pins the persistent *container object* itself. (Among other things, the container object is where an application-defined container stores its persistent data.) Pinning guarantees that the container object is readily available in memory for as long as it is needed. Closing the handle removes its particular "pin"; when the last open handle to the container is closed, the last pin is removed and the container object is closed. Objectivity / DB is permitted to swap the page containing the closed container object out of the cache to make room for other open persistent objects. (This affects only the container object itself, and has no effect any open basic objects in the container.)

A handle to a container, like any handle, has cache-related state that is associated with the Objectivity context in which it was created. Therefore, a handle:

- Cannot be stored persistently—for example, as an attribute value. In fact, the DDL processor does not accept `ooHandle(ooContObj)` as a data member type in a persistence-capable class definition.
- Cannot be passed between Objectivity contexts.

Object References

Object references are optimized for implementing persistent links because they are essentially wrappers for object identifiers. Thus, when an object reference is set to reference a particular container, the object reference stores the object identifier of that container. The object reference never acquires a pointer to the container in memory; instead, whenever the container is accessed through the object reference, the operation is delegated to a temporary handle that provides the necessary pointer.

Because it has no bulky cache-specific state, an object reference to a container:

- Can be stored persistently—for example, as an attribute value. The DDL processor accepts `ooRef(ooContObj)` as a data member type in a persistence-capable class definition.

- Can be passed between Objectivity contexts.

For convenience, an application can use an object reference (instead of a handle) to perform an operation on a referenced container or to access one of the container's members. However, poor performance results when an object reference is used for multiple such operations on the same container, because *each* operation causes a temporary handle to be created, used, and discarded. Performance may also be affected by swapping, because the object reference does not pin the container's memory representation in the Objectivity / DB cache.

Working With Container Handles

NOTE For simplicity, this section describes how to work with handles. Except where noted, the same information applies to object references.

An application normally creates a handle as a local variable on the stack, rather than allocating it on the heap. A handle should not be declared as `const`, because its internal state is changed by any operation that accesses a container through it. (Object references may be declared as `const`.) Applications should not explicitly define subclasses of the *`ooRefHandle`*`(ooContObj)` classes; any necessary subclasses are generated automatically by the DDL processor when an application defines its own subclasses of `ooContObj`.

A new handle is normally *null*—that is, it contains the value 0 instead of an object identifier for a container. The application can then set the handle to reference a particular container in any of the following ways:

- By creating a new container with `operator new` of the `ooContObj` class and assigning the result to the handle.
- By finding an existing container with the handle's `exist`, `lookupObj`, or `open` member function.
- By passing the handle to a member function that sets it, such as the `containedIn` member function of a basic-object handle, which finds the container where the basic object is located.
- By assignment or initialization from another handle or object reference.

An object reference may be set in any of these ways, with the following exception—the result of `operator new` may not be assigned to an object reference.

A handle continues to reference the same container until it is set to another container or to null. Furthermore, multiple handles and object references can be set to the same container. A handle of class `ooHandle(ooContObj)` can be set to

reference any kind of container—that is, an instance of `ooContObj` or any predefined or application-defined class derived from `ooContObj`.

An application operates on a container by calling:

- Member functions of a handle that references the container. As indicated in "Reference Summary" on page 515, such member functions allow you to get the attributes of the referenced container, find its basic objects, and so on.

  To call a member function of a handle, you use the direct member-access operator (`.`). For example, `contH.name` calls the `name` member function of the container handle `contH`.

- Member functions of the referenced container itself (including member functions inherited from `ooObj`).

  To call a member function of a referenced container, you use the handle's overloaded indirect member-access operator (`operator->`). For example, `contH->ooIsKindOf` calls the `ooIsKindOf` member function on the container that is referenced by the handle `contH`.

Although most of a container handle's member functions operate on the referenced container, some functions operate on the handle itself. For example, you use:

- The assignment operator `operator=` to set a handle from another handle or from an object reference.
- The inherited comparison operators `operator==` and `operator!=` to test whether a handle references the same container as another handle or object reference.
- The inherited member function `isNull` to test whether a handle is null. (Alternatively, you can use the overloaded `operator==` to compare a handle to 0.)

A handle preserves its reference to a container across transaction boundaries, provided that the handle does not go out of scope and is not set to null as the result of an abort operation. Before reusing the handle in a new transaction, however, the application should call `isValid` to test whether the handle is still *valid*—that is, whether it still references an existing container. A handle becomes invalid if it is set to null *or* if the referenced container has been deleted by another process between transactions.

Specifying a Container Handle to a Function

Objectivity / C++ functions that require a container as input typically obtain the container through a parameter of type `const ooHandle(ooContObj) &`. You can pass any type of container handle through such a parameter, because `ooHandle(ooContObj)` is the base class for all container-handle classes. For example, you could pass a handle of type `ooHandle(ooGCContObj)` or

ooHandle(*appClass*), where *appClass* is an application-defined container class.

If a function can accept other types of Objectivity / DB objects in addition to containers, the function may do so through a parameter of type const ooHandle(ooObj) &. You can pass a container handle to a parameter of this type because ooHandle(ooContObj) is derived from ooHandle(ooObj). Many functions accept a container handle where a more general handle is requested, including:

- Functions that operate on any persistent object, such as the persistent-collection functions for managing elements.
- Functions that operate on (nearly) any Objectivity / DB object, such as ooDelete and various functions that scan, cluster, or define a scope name within the object.

Some functions do *not* accept a container handle where a more general handle is requested—for example, functions that manage the versioning of basic objects.

Opening and Closing a Container Handle

NOTE This subsection applies only to handles, not to object references, which are in effect always closed.

A handle is automatically opened when a container is opened through it. The open container is both locked and represented in memory; the open handle manages a pointer to the container, pinning the container in memory until the handle is closed. A closed handle, which has an object identifier instead of a pointer, can reference either an open or a closed container.

The most common way to open a container through a handle is to do so implicitly by using the handle's indirect member-access operator (operator->) to access a member of the referenced container. Alternatively, a referenced container can be opened by explicit request—for example, by calling the handle's open or update member function. Another way to explicitly open a container is by finding it with a function whose *openMode* parameter is either oocRead or oocUpdate. (Most functions that set a handle to a found container provide an *openMode* parameter for specifying the desired level of access through that handle.) In all cases, if the found or referenced container is already open (for example, because a basic object in it was opened earlier in the transaction), the accessing handle gets a pointer to the container's existing memory representation and adds a pin.

You obtain a closed handle to a container by finding the container with a function whose *openMode* parameter is set to oocNoOpen. Such operations

simply provide the handle with a container's object identifier without adding a pin, even if the container is already open through another handle.

Objectivity / DB automatically closes an open container handle when the handle is destroyed (for example, by going out of scope), when it is set to reference another container, or when the transaction that opened it commits or aborts. An application can close a handle explicitly by calling the handle's `close` member function.

Reference Summary

The following table summarizes all the member functions that are available to instances of *`ooRefHandle`*`(ooContObj)`. Member functions indicated as *(inherited)* are documented with the `ooRefHandle(ooObj)` classes (page 593).

The summarized member functions are defined on both the object-reference class and the handle class. Two operators are defined on only the handle class, namely, `operator*` and `operator ooContObj*`.

Creating a Handle or Object Reference	`ooHandle(ooContObj)` `ooRef(ooContObj)`

Setting the Handle or Object Reference	`operator=`
	`open`
	`exist`
	`lookupObj`

Comparing Handles and Object References	`operator==` *(inherited)* `operator!=` *(inherited)*

Accessing the Container	`operator->`
	`operator*`
	`operator ooContObj*`
	`ptr` (*ODMG*)

Opening, Closing, and Locking the Container	`open`
	`openMode`
	`lock` *(inherited)*
	`lockNoProp`
	`refreshOpen`
	`close`

Modifying the Container	`update` *(inherited)* `delete_object` *(inherited)*

Getting Information About the Container	name hash nPage numLogicalPages percentGrow typeN *(inherited)* typeName *(inherited)* print *(inherited)* sprint *(inherited)*
Testing the Container	exist isUpdated
Testing the Handle or Object Reference	is_null *(inherited)* isNull *(inherited)* isValid *(inherited)* operator int *(inherited)* operator ooContObj* operator ooObj* *(inherited)*
Working With Scope Names	nameObj *(inherited)* getObjName *(inherited)* unnameObj *(inherited)* getNameObj *(inherited)* getNameScope *(inherited)*
Finding Objects	exist open lookupObj contains containedIn controlledBy (*FTO*) getNameObj *(inherited)* getNameScope *(inherited)*
Converting Objects	convertObjects
Working With Autonomous Partitions (*FTO*)	transferControl controlledBy returnControl
ODMG Interface	operator d_Ref_Any ptr

Reference Index

ooRef(ooContObj)	Constructs an object reference that references the same container as the specified object reference, handle, pointer, or ODMG generic reference.
open	Explicitly opens the referenced or specified container, preparing the container for the specified level of access.
openMode	Gets the current level of access to the referenced container.
operator->	Indirect member-access operator; accesses a member of the referenced container.
operator*	Handle class only. Dereference operator; returns the container referenced by this handle.
operator=	Assignment operator; sets this object reference or handle to reference the same container as the specified object reference, handle, or pointer.
operator d_Ref_Any	(ODMG) Conversion operator that returns an ODMG generic reference to the referenced container.
operator ooContObj*	Handle class only. Conversion operator that returns a pointer to the referenced container.
percentGrow	Gets the growth factor for the referenced container.
ptr	(ODMG) Returns a C++ pointer to the referenced container.
refreshOpen	Reopens the referenced container, refreshing the view of the container in the MROW transaction reading it.
returnControl	(FTO) Releases the referenced container from the currently controlling autonomous partition and returns control to the autonomous partition of the container's database.
transferControl	(FTO) Transfers the referenced container into the control of the specified autonomous partition.

Constructors

ooHandle(ooContObj)

Default constructor that constructs a null handle.

```
ooHandle(ooContObj)();
```

ooHandle(ooContObj)

Constructs a handle that references the same container as the specified object reference, handle, pointer, or ODMG generic reference.

```
1.   ooHandle(ooContObj)(
       const ooRefHandle(ooContObj) &containerRH);
2.   ooHandle(ooContObj)(const ooContObj *containerP);
(ODMG) 3.   ooHandle(ooContObj)(const d_Ref_Any &from);
```

Parameters

containerRH

Object reference or handle to a container.

containerP

Pointer to a container. The pointer *must* be the result of `operator new` on a container class.

from

(*ODMG*) An ODMG generic reference to a container. An error is signalled if *from* references an object that is not an instance of `ooContObj` or a class derived from `ooContObj`.

Discussion

Variants 1 and 3 allow a new handle to be constructed from an existing object reference, handle, or ODMG generic reference. If the new handle is constructed from an existing open handle, the new handle is open; in all other cases, the new handle is closed.

Variant 2, which constructs a handle from the specified pointer, has a narrower purpose—to obtain an open handle to a newly created container so you can perform persistence operations on it, and so the container can eventually be unpinned when it is no longer needed in memory.

ooRef(ooContObj)

Default constructor that constructs a null object reference.

```
ooRef(ooContObj)();
```

ooRef(ooContObj)

Constructs an object reference that references the same container as the specified object reference, handle, pointer, or ODMG generic reference.

```
1.   ooRef(ooContObj)(
       const ooRefHandle(ooContObj) &containerRH);
```

```
            2.   ooRef(ooContObj)(const ooContObj *containerP);
      (ODMG)3.   ooRef(ooContObj)(const d_Ref_Any &from);
```

Parameters

containerRH

Object reference or handle to a container.

containerP

Pointer to a container. The pointer may *not* be the result of `operator new`. Instead, the pointer must be the result of using either `operator ooContObj*` on a handle *or* `ptr` on an object reference or handle earlier in the same transaction. If the specified pointer was originally extracted from a handle, that handle must still exist and reference the same container.

from

(*ODMG*) An ODMG generic reference to a container. An error is signalled if *`from`* references an object that is not an instance of `ooContObj` or a class derived from `ooContObj`.

Discussion

Variants 1 and 3 allow a new object reference to be constructed from an existing object reference, handle, or ODMG generic reference.

Variant 2 has a narrower purpose, which is to allow you to resume persistence operations on a container after manipulating it through a pointer. The use of this variant should be rare, however, because pointers are not normally used to manipulate containers.

Operators

operator->

Indirect member-access operator; accesses a member of the referenced container.

```
const ooContObj *operator->();
```

Returns

Pointer to the referenced container.

Discussion

The accessed container is opened for read, if it is not already open.

You use `operator->` in an expression *`handle->member`*, where *`handle`* is an instance of `ooHandle(ooContObj)` and *`member`* is the name of a public member defined on class `ooContObj`. As for any overloading of the C++ member-access operator (->), the expression *`handle->member`* is interpreted as (*`handle`*`.operator->())->`*`member`*. That is, the overloaded `operator->` returns a pointer to the referenced container, and then the ordinary C++

`operator->` selects the specified member of that container, returning the value of that member.

If the referenced object is an instance of `ooContObj`, `operator->` can access any of the object's public members. If the referenced object is an instance of a derived class, `operator->` accesses only the members that are defined in `ooContObj`.

operator*

Handle class only. Dereference operator; returns the container referenced by this handle.

```
ooContObj &operator*();
```

Returns C++ reference to the container referenced by this handle.

Discussion This operator enables you to pass a handle to a function that accepts a container by reference. This operator is analogous to the C++ `operator*` for dereferencing a pointer.

The container referenced by this handle is opened for read, if it is not already open.

Warning: The returned C++ reference is guaranteed valid only as long as the handle exists, remains open, and references the same container.

Example This example uses `operator*` to pass a container handle to `helperFunction`, which accepts a C++ reference to a container.

```
void helperFunction(ooContObj &aContainer);

void processContainer(ooHandle(ooContObj) &contH) {
   …
   helperFunction(*contH);
   …
}
```

operator=

Assignment operator; sets this object reference or handle to reference the same container as the specified object reference, handle, or pointer.

```
1.   ooRefHandle(ooContObj) &operator=(
       const ooRefHandle(ooContObj) &containerRH);
2.   ooRefHandle(ooContObj) &operator=(
       const ooContObj *containerP);
```

Parameters

containerRH

Object reference or handle to a container.

containerP

0, or a nonnull pointer to a container:

- If you are assigning to a handle, the specified pointer *must* be the result of `operator new` on `ooContObj` or an application-defined derived class.
- If you are assigning to an object reference, the specified pointer may *not* be the result of `operator new`. Instead, the pointer must be the result of using either `operator ooContObj*` on a handle *or* `ptr` on an object reference or handle earlier in the same transaction. If the specified pointer was originally extracted from a handle, that handle must still exist and reference the same object.

Returns

This object reference or handle, after it has been set to reference the specified container.

Discussion

Variant 1 allows you to use the specified object reference or handle to produce another object reference or handle to the same container. If you are assigning to a handle from an open handle, the returned handle is open; in all other cases, the returned handle is closed.

Variant 2 allows you to set this object reference or handle to null. Otherwise, assignment-from-pointer has two specific purposes, depending on whether you are assigning to a handle or to an object reference:

- Pointer-to-handle assignment enables you to obtain an open handle to a newly created container, so you can perform persistence operations on it, and so the container can eventually be unpinned when it is no longer needed in memory.
- Pointer-to-object-reference assignment enables you to resume persistence operations on a container after manipulating it through a pointer. This usage of variant 2 is rare, because pointers are not normally used to manipulate containers.

operator d_Ref_Any

(*ODMG*) Conversion operator that returns an ODMG generic reference to the referenced container.

```
operator d_Ref_Any() const;
```

operator ooContObj*

Handle class only. Conversion operator that returns a pointer to the referenced container.

```
operator ooContObj*();
```

Returns

Pointer to the container referenced by this handle. Returns a null pointer if this handle is a null handle.

Discussion

This conversion operator enables you to:

- Pass a handle to a function that accepts a pointer to a container.
- Assign a handle to an `ooContObj*` variable (for example, to pass to the overloaded `operator delete`).
- Use a handle as the conditional expression in an `if` or `while` statement to test whether the handle is null.

The container referenced by this handle is opened for read, if it is not already open.

WARNING The returned pointer is guaranteed valid only as long as the handle exists, remains open, and references the same container.

An application generally relies on handles to provide memory management for persistent objects, and avoids the explicit use of pointers to such objects. On occasion, explicit use of pointers is required for performance reasons or for compatibility with functions that are indifferent to persistence (for example, in legacy code or a third-party library). However, you should be careful when manipulating a pointer from a handle because the validity of the pointer depends on the state of the handle. You should not use the returned pointer in other persistence operations (for example, do not pass it to any Objectivity/C++ member function other than the overloaded `operator delete`).

Example

The container handle `contH` is used as a conditional expression which evaluates to 0 if the handle is null.

```
ooHandle(ooContObj) contH;
... // Set contH to some container
if (contH) {
... //Do something interesting if initialization was successful
}
```

See also

`ptr`

Member Functions

close

Explicitly closes this handle.

```
ooStatus close() const;
```

Returns `oocSuccess` if successful; otherwise `oocError`.

Discussion This member function is redundant for object references, which are, in effect, always closed. Therefore, you should use this member function only on handles.

Objectivity / DB automatically closes container handles when they go out of scope, when they are set to reference other containers, or when the transaction that opened them commits or aborts.

You can use the `close` member function to close a container handle explicitly. This informs Objectivity / DB that the application no longer requires access to the referenced container through this handle. Closing does not, however, affect any open objects in the container, nor does it release the lock on the container; locks are released only by committing or aborting a transaction.

When closed explicitly, a container handle retains the object identifier of the container to which it refers, so you can reopen it without reinitializing. Note, however, that a retained object identifier can become invalid between transactions (for example, because a concurrent process has deleted the corresponding container), and opening a handle with an invalid object identifier signals an error.

Closing a container handle invalidates its pointer to the container's representation in the Objectivity / DB cache. Closing the last open handle to a particular container unpins and closes the container object; closing the last open object on a buffer page permits Objectivity / DB to swap the page out of the cache as needed. Note that this affects only the persistent data defined for application-specific containers, and does not affect any open basic objects in the container.

containedIn

Finds the database that contains the referenced container.

```
1.  ooHandle(ooDBObj) containedIn() const;
2.  ooRef(ooDBObj) &containedIn(
      ooRef(ooDBObj) &database) const;
```

```
3.  ooHandle(ooDBObj)&containedIn(
      ooHandle(ooDBObj) &database) const;
```

Parameters *database*

Object reference or handle to be set to the found database.

Returns Object reference or handle to the found database.

Discussion When called without a *database* parameter, `containedIn` allocates a new database handle and returns it. Otherwise, `containedIn` returns the object reference or handle that is passed to it.

contains

Initializes an object iterator to find all basic objects stored in the referenced container.

```
ooStatus contains(
     ooItr(ooObj) &objI,
     const ooMode openMode = oocNoOpen) const;
```

Parameters *objI*

Object iterator for finding the contained basic objects.

openMode

Intended level of access to the basic objects found by the iterator's `next` member function:

- `oocNoOpen` (the default) causes `next` to set the iterator to the next basic object without opening it.
- `oocRead` causes `next` to open the next basic object for read.
- `oocUpdate` causes `next` to open the next basic object for update.

Returns `oocSuccess` if successful; otherwise `oocError`.

controlledBy

(*FTO*) Finds the autonomous partition, if any, that controls the referenced container.

```
1.  ooHandle(ooAPObj) controlledBy();
2.  ooRef(ooAPObj) &controlledBy(
      ooRef(ooAPObj) &partition) const;
3.  ooHandle(ooAPObj) &controlledBy(
      ooHandle(ooAPObj) &partition) const;
```

Parameters *partition*

Object reference or handle to set to the controlling autonomous partition.

Returns Object reference or handle to the controlling autonomous partition. If the container has not been transferred to the control of a partition, a null object reference or handle is returned.

Discussion When called without a parameter, `controlledBy` allocates a new autonomous-partition handle and returns it. Otherwise, `controlledBy` returns the object reference or handle that is passed to it.

See also `returnControl`
`transferControl`

convertObjects

Performs on-demand object conversion on any affected objects in the referenced container.

```
ooStatus convertObjects();
```

Returns `oocSuccess` if successful, or `oocError` if the federated database is opened only for read.

Discussion Object conversion is the process of making existing persistent objects consistent with class definition changes introduced by schema evolution. Certain schema evolution operations affect how instances of a class should be laid out in storage. After you perform such operations, existing objects of the changed classes are rendered out-of-date until they are converted to their new representations.

In general, you can allow each affected object to be converted automatically the first time it is accessed after schema evolution, potentially distributing the performance impact of conversion across many transactions. Alternatively, you can concentrate the performance impact of conversion into fewer transactions by converting all the affected objects in a container, a database, or a federated database *on demand*. You use this member function in an update transaction to convert the affected objects in a container on demand. This member function has no effect if the affected objects in the container have already been converted.

Note: On-demand object conversion cannot be used for schema operations that require an upgrade application; see `ooTrans::upgrade`.

The `convertObjects` member function automatically drops any index that is invalidated by a schema evolution change. Specifically, if you changed the type or deleted a data member that is a key field in a key description, the corresponding indexes are dropped.

See also Chapter 19, “Object Conversion,” in the Objectivity / C++ programmer’s guide

exist

Tests whether the specified container exists in the specified database; if successful, sets this object reference or handle to reference the container.

```
ooBoolean exist(
     const ooHandle(ooDBObj) &database,
     const char *contSysName,
     const ooMode openMode = oocNoOpen);
```

Parameters

database

Handle to the database to search.

contSysName

System name of the desired container.

openMode

Intended level of access to the container, if it exists:

- Specify `oocNoOpen` (the default), to set this object reference or handle to the container without opening it.
- Specify `oocRead` to open the container for read.
- Specify `oocUpdate` to open the container for update.

Returns `oocTrue` if the specified container exists, or `oocFalse` if the container does not exist or if it is not accessible.

Discussion If the specified container exists, this object reference or handle is set to reference it; otherwise, this object reference or handle is set to null.

If you specifically want to test for existence, you use the *openMode* parameter’s default value (*oocNoOpen*). Otherwise, a return value of *oocFalse* could mean either that the container doesn’t exist, or that it does exist, but cannot be opened.

NOTE This member function identifies a container using its system name. To find a container by its scope name, use `lookupObj`.

---

hash

Gets the hash value for the referenced container.

```
int32 hash() const;
```

Returns -1 if this object reference or handle is null or if the referenced container cannot be opened; otherwise, returns the referenced container's hash value:

- 0 indicates a nonhashed container.
- 1 or greater indicates a hashed container, and is also the clustering factor for keyed objects.

Discussion A container's hash value is set when the container is created. A hashed container and the objects in it can be used as scopes for naming objects; furthermore, a hashed container can contain keyed objects. A clustering factor is the number of sequentially keyed objects to be placed onto a page when keyed objects are created within the container.

See also `ooContObj::operator new`.

isUpdated

Tests whether the referenced container has already been updated and committed by another transaction.

```
ooBoolean isUpdated() const;
```

Returns `oocTrue` if the referenced container has been updated and committed by another transaction since being locked for read by the current MROW transaction; otherwise, `oocFalse`.

Discussion You can use this member function within an MROW transaction to determine whether to call `refreshOpen`.

lockNoProp

Explicitly locks the referenced container, without propagating locks to associated destination objects.

```
ooStatus lockNoProp(const ooLockMode lockMode) const;
```

Parameters *lockMode*

Type of lock to request:

- `oocLockRead` requests a read lock.
- `oocLockUpdate` requests an update lock.

Returns `oocSuccess` if the requested lock is obtained; otherwise `oocError`.

Discussion Objectivity / DB operations request and obtain locks implicitly as they are needed. You use this member function to obtain a lock explicitly when you want to reserve access to an object in advance, but you do *not* want to lock any

associated destination objects, even along associations that have lock propagation enabled.

Locking a container locks all the basic objects in it.

Whenever a lock is requested on a container, Objectivity/DB applies the transaction's concurrent access policy to determine whether the requested lock is compatible with other existing locks. An error is signaled if a requested lock cannot be obtained.

See also *ooRefHandle*(ooObj)::lock
ooLockMode global type

lookupObj

Finds the container with the specified scope name in the specified scope; if successful, sets this object reference or handle to reference the found container.

```
ooStatus lookupObj(
     const ooHandle(ooObj) &scope,
     const char *name,
     const ooMode openMode = oocRead) const;
```

Parameters

scope

Handle to the scope object that defines the name scope to search. *scope* can reference the federated database, a database, a persistent container, a persistent basic object, or an autonomous partition.

name

Scope name to look up in the scope specified by *scope*.

openMode

Intended level of access to the found container:

- Specify oocRead (the default) to open the object for read.
- Specify oocUpdate to open the object for update.
- Specify oocNoOpen to set this object reference or handle to the object without opening it.

Returns

oocSuccess if a container is found; otherwise oocError.

Discussion

Scope-named objects are found only if they are instances of ooContObj or application-defined classes derived from ooContObj.

name

Gets the system name of the referenced container.

```
char *name() const;
```

Returns — Pointer to a string containing the system name. If the container does not have a system name, the returned pointer is null.

Discussion — The string is statically allocated by the member function and is overwritten with each invocation. You should make a local copy of the returned string if you intend to use it later in the application.

nPage

Gets the current number of storage pages in the referenced container.

```
uint32 nPage();
```

Returns — The current number of storage pages in the container; returns 0 if this object reference or handle is null or if the container cannot be opened.

Discussion — You can use this function to help you calculate the size (in bytes) of a container within a database.

A storage page is the minimum unit of transfer to and from disk and across networks. The size of a storage page is configurable for each federated database and is set with the federated database is created. A federated database's storage pages are usually sized so that one or more typical persistent objects (called *small objects*) will fit within a single storage page; by definition, a *large object* spans multiple storage pages. You can get the page size of a federated database by calling the `pageSize` member function on a federated-database handle.

A subset of a container's storage pages are also logical pages; logical pages are storage pages that contain either one or more small objects, or the header information for a large object. If no large objects reside in a container, the number of storage pages will be close to the number of logical pages. A container with very large objects will have many more storage pages than logical pages. You can get the number of logical pages by calling the `numLogicalPages` member function.

numLogicalPages

Gets the current number of logical pages in the referenced container.

```
uint32 numLogicalPages();
```

Returns The current number of logical pages in the container; returns 0 if this object reference or handle is null or if the container cannot be opened.

Discussion You can use this function to monitor when the container will reach its maximum logical page limit (65535).

A logical page is a storage page that contains either one or more small objects, or the header information for a large object. Logical pages are numbered within each container; a logical page number appears as one of the fields within the object identifier of a persistent object.

Logical pages are a subset of a container's storage pages. If no large objects reside in a container, the number of logical pages will be close to the number of storage pages. A container with very large objects will have many fewer logical pages than storage pages. You can get the number of storage pages by calling the nPage member function.

open

Explicitly opens the referenced or specified container, preparing the container for the specified level of access.

```
1.  ooStatus open(const ooMode openMode = oocRead);
2.  ooStatus open(
      const ooHandle(ooDBObj) &database,
      const char *contSysName,
      const ooMode openMode = oocRead);
```

Parameters *openMode*

Intended level of access to the opened container:

- Specify `oocRead` (the default) to open the container for read. This implicitly locks the container (and the basic objects in it) for read.
- Specify `oocUpdate` to open the container for update (read and write). This implicitly locks the container (and the basic objects in it) for update.

database

Handle to the database in which to find the specified container.

contSysName

System name of the container to open.

Returns `oocSuccess` if successful; otherwise `oocError`.

Discussion Variant 1 assumes that this object reference or handle already references a container, and opens the referenced container.

Variant 2 finds and explicitly opens the container with the specified system name, and sets this object reference or handle to reference it. An error is signalled if no container exists with the specified system name in the specified database. This variant is especially useful when you want to use the container as an entry point into your data. For example, you might find and open a container so you can iterate over the basic objects in it.

Opening a container makes it available to an application by:

- Implicitly locking the container for read or update, as specified by *openMode*.
- Obtaining a representation of the container in memory, either by fetching storage pages from the database or reusing an existing memory representation that is guaranteed current. This memory representation includes pages describing the container and any data defined for it by the application (if the container is an instance of an application-defined container class).

Opening a container does not open any of the basic objects in it.

Opening a container for update additionally marks it as modified, causing any application-specific data to be written to the database when the transaction commits, whether or not that data was actually modified. 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.

The open operation fails if the container cannot be locked—for example, due to a lock conflict. Objectivity / DB applies the transaction's concurrent access policy to determine whether the requested lock is compatible with other existing locks. Once a lock is obtained, it is kept until the transaction either commits or aborts.

A container is opened automatically when you open one of the basic objects in it. Furthermore, using `operator->` to access a member of a container implicitly opens the container for read. You open a container explicitly when:

- You require update access so you can modify the container's application-specific data.
- You want to reserve either read or update access to the container in advance—for example, before starting a complex operation.

openMode

Gets the current level of access to the referenced container.

```
ooMode openMode() const;
```

Returns One of the following constants:

- oocNoOpen—the container is not open in this transaction.
- oocRead—the container is open for read in this transaction.
- oocUpdate—the container is open for update in this transaction.

percentGrow

Gets the growth factor for the referenced container.

```
uint32 percentGrow() const;
```

Returns The growth factor for the container; returns 0 if this object reference or handle is null or if the container cannot be opened.

Discussion A container's growth factor is set when the container is created. The growth factor is the amount by which the container may grow when it needs to accommodate more basic objects, expressed as a percentage of its current size.

See also ooContObj::operator new.

ptr

(*ODMG*) Returns a C++ pointer to the referenced container.

```
ooContObj *ptr();
```

Discussion You use this member function to obtain a pointer to a container—for example, to pass to a function that accepts a pointer instead of a handle or object reference.

If ptr is called on an object reference, the referenced container is opened for update. If ptr is called on a handle, the referenced container is opened for read.

Warning: The returned pointer is guaranteed valid for only a limited time:

- If ptr is called on an object reference, the returned pointer is valid and the container is pinned in memory until the end of the transaction.
- If ptr is called on a handle, the returned pointer is valid only as long as the handle exists, remains open, and references the same container (equivalent to operator ooContObj*).

An application generally relies on handles to provide memory management for persistent objects, and avoids the explicit use of pointers to such objects. On occasion, explicit use of pointers is required for performance reasons or for compatibility with functions that are indifferent to persistence (for example, in legacy code or a third-party library). However, you should be careful when manipulating persistent objects through pointers:

- Pointers extracted from handles become invalid if the handles change or go out of scope.
- Pointers extracted from object references can cause the Objectivity / DB cache to run out of memory if too many objects are pinned until the end of the transaction.

You should not use the returned pointer in other persistence operations (for example, do not pass it to any Objectivity / C++ member function other than the overloaded `operator delete`).

refreshOpen

Reopens the referenced container, refreshing the view of the container in the MROW transaction reading it.

```
ooStatus refreshOpen(
     const ooMode openMode,
     ooBoolean *pIsUpdated,
     ooBoolean closeHandles = oocFalse);
```

Parameters

openMode

Intended level of access to the reopened container:

- Specify `oocRead` to open the container for read. This implicitly requests a read lock on the container.
- Specify `oocUpdate` to open the container for update (read and write). This implicitly requests an update lock on the container.

pIsUpdated

Pointer to a value that, on return, indicates whether the referenced container has been updated and committed by another transaction since being locked for read by the current MROW transaction. The returned value corresponds to the result of the `isUpdated` member function.

closeHandles

Action to take if the transaction has any open handles to objects in the container:

- Specify `oocTrue` to close the all open handles.
- Specify `oocFalse` (the default) to signal an error and take no other action.

Returns

`oocSuccess` if successful; otherwise `oocError`.

Discussion

When you use an MROW transaction to read a container, one other transaction is allowed to concurrently update that container. If the updating transaction commits, your view is rendered out-of-date. You use the `refreshOpen` member

function to open the most recently committed version of an updated container within an MROW transaction. This means that each open object in the container is implicitly closed and then reopened the next time you access it.

returnControl

(*FTO*) Releases the referenced container from the currently controlling autonomous partition and returns control to the autonomous partition of the container's database.

```
ooStatus returnControl() const;
```

Returns `oocSuccess` if successful; otherwise `oocError`.

Discussion This member function physically moves the container back to the file of its database.

See also `controlledBy`
`transferControl`

transferControl

(*FTO*) Transfers the referenced container into the control of the specified autonomous partition.

```
ooStatus transferControl(
     const ooHandle(ooAPObj) &newControllingAP) const;
```

Parameters *`newControllingAP`*

Handle to the autonomous partition that is to control the container.

Returns `oocSuccess` if successful; otherwise `oocError`.

Discussion This member function physically moves the referenced container into the system database file of the specified autonomous partition.

See also `controlledBy`
`returnControl`

ooRefHandle(ooDBObj) Classes

Inheritance: **ooRef(ooObj)->ooRef(ooDBObj)**

Inheritance: **ooHandle(ooObj)->ooHandle(ooDBObj)**

The abbreviation *ooRefHandle*(ooDBObj) refers to two non-persistence-capable classes:

- ooRef(ooDBObj), which represents an *object reference* to a database.
- ooHandle(ooDBObj), which represents a *handle* to a database.

The two classes ooRef(ooDBObj) and ooHandle(ooDBObj) are documented together because they define the same set of member functions. These member functions provide the primary interface for operating on Objectivity / DB databases (instances of ooDBObj).

(*DRO*) The *ooRefHandle*(ooDBObj) classes also provide the primary interface for managing database images (also instances of ooDBObj), which you can create if you have bought and installed both Objectivity / DB Data Replication Option (Objectivity / DRO) and Objectivity / DB Fault Tolerant Option (Objectivity / FTO).

See:

- “Reference Summary” on page 541 for an overview of member functions
- “Reference Index” on page 543 for a list of member functions

About Database Handles and References

An application works with a database indirectly through one or more handles or object references—that is, through instances of *ooRefHandle*(ooDBObj) that are set to reference the desired database. A handle or object reference to a database both identifies the database and provides the complete public interface for operating on it.

Handles and object references to databases do not support indirect member access: the *ooRefHandle*(ooDBObj) classes provide no indirect member-access operator (->), and the ooDBObj class defines no public member functions other than a constructor and operator new.

You can work with a database through either a handle or an object reference—the choice is arbitrary, except as described in "Structure and Behavior" on page 538. Most applications use handles rather than object references.

Interface

The *ooRefHandle*(ooDBObj) classes provide the primary interface for operating on a referenced database. Part of this interface consists of member functions defined by *ooRefHandle*(ooDBObj) for specialized operations such as getting the number of containers in a database. The other part of this interface consists of member functions defined by the *ooRefHandle*(ooObj) base classes for more general Objectivity / DB operations, such as opening, locking, printing object identifiers, and so on. These member functions are either inherited by the *ooRefHandle*(ooDBObj) classes or redefined wherever type-specific parameters or behavior are required.

Some of the member functions defined by the *ooRefHandle*(ooObj) base classes are *not* available to instances of the *ooRefHandle*(ooDBObj) classes. These include member functions for moving, copying, versioning, scope-naming, and member-access operations, which apply only to basic objects or persistent objects. The disallowed member functions and operators are redefined as private members of the *ooRefHandle*(ooDBObj) classes.

Structure and Behavior

Handles and object references to databases are essentially wrappers for database identifiers. 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. Subsequent member-function calls on the handle operate on the instance of ooDBObj that represents the identified database in memory.

In general, object references are optimized for implementing links among related persistent objects, while handles are optimized for memory management and member-access. When a database is referenced, however, these optimizations are largely irrelevant, because databases (unlike persistent objects):

- Cannot be linked (for example, through associations).
- Are not subject to memory management (they have no attributes for persistent data and are therefore not manipulated through pointers).
- Have no accessible members.

One significant exception is that a handle to a database, like any handle, contains cache-related state that is associated with the Objectivity context in which it was created. Therefore, only object references (but not handles) can be passed between Objectivity contexts. Otherwise, a handle to a database and an object reference to a database are functionally equivalent.

Working With Database Handles

NOTE For simplicity, this section describes how to work with handles. Except where noted, the same information applies to object references.

An application normally creates a handle as a local variable on the stack, rather than allocating it on the heap. A handle should not be declared as `const`, because its internal state may be changed when a database is accessed through it. (Object references may be declared as `const`). Applications should not create subclasses of the *`ooRefHandle`*`(ooDBObj)` classes.

A new handle is normally *null*—that is, it contains the value 0 instead of the object identifier for a database. The application can then set the handle to reference a particular database in any of the following ways:

- By creating a new database with `operator new` of the `ooDBObj` class and assigning the result to the handle.
- By finding an existing database with the handle's `exist` or `open` member function.
- By passing the handle to a member function that sets it, such as the `containedIn` member function of a container handle, which finds the database where a container is located.
- By assignment or initialization from another handle or object reference.

A handle continues to reference the same database until it is set to another database or to null. Furthermore, multiple handles and object references can be set to the same database.

An application operates on a database by calling member functions on a handle that references it. To call a member function of a handle, you use the direct member-access operator (`.`). For example, `dbH.name` calls the `name` member function of the handle `dbH`.

As indicated in "Reference Summary" on page 541, an application can use the handle's member functions to get and change a referenced database's attributes, find its containers, tidy its disk space, and so on. If Objectivity / FTO and Objectivity / DRO are installed, the application can also create and manage

multiple images of a database. For more information about operating on databases and database images, see Chapter 8, “Storage Objects,” and Chapter 28, “Database Images,” respectively, in the Objectivity / C++ programmer’s guide.

Although most of a database handle’s member functions operate on the referenced database, some functions operate on the handle itself. For example, you use:

- The assignment operator `operator=` to set a handle from another handle or from an object reference.
- The inherited comparison operators `operator==` and `operator!=` to test whether a handle references the same database as another handle or object reference.
- The inherited member function `isNull` to test whether a handle is null. (Alternatively, you can use the overloaded `operator==` to compare a handle to 0.)

A handle preserves its reference to a database across transaction boundaries, provided that the handle does not go out of scope and is not set to null as the result of an abort operation. Before reusing the handle in a new transaction, however, the application should call `isValid` to test whether the handle is still *valid*—that is, whether it still references an existing database. A handle becomes invalid if it is set to null *or* if the referenced database has been deleted by another process between transactions.

Objectivity / C++ functions that require a database as input normally obtain the database through a parameter of type `const ooHandle(ooDBObj) &`. If a function manipulates other types of Objectivity / DB objects as well, the parameter type may be specified as `const ooHandle(ooObj) &`. You can pass a database handle to a parameter of this type, because `ooHandle(ooDBObj)` is derived from `ooHandle(ooObj)`. In practice, however, relatively few functions accept a database handle where a more general handle is requested; these include `ooDelete` and various functions that require a storage object for scanning, clustering, or as a name scope. In most cases, a function that requests a general-purpose handle operates only on basic objects or persistent objects, and signals an error if you specify a database handle.

Any operation that affects a database opens it implicitly if it is not already open; an application does not need to open a database explicitly unless it is used as the entry point into the data or unless access must be guaranteed in advance. A handle is automatically closed when it is destroyed (for example, when it goes out of scope). However, closing one or more handles to a particular database has no effect on that database, which remains open until the transaction commits or aborts.

Reference Summary

The following table summarizes all the member functions that are available to instances of *ooRefHandle*(ooDBObj). Member functions indicated as *(inherited)* documented with the ooRefHandle(ooObj) classes (page 593).

Creating a Handle or Object Reference	ooHandle(ooDBObj) ooRef(ooDBObj)
Setting the Handle or Object Reference	operator= open exist
Comparing Handles and Object References	operator== *(inherited)* operator!= *(inherited)*
Opening, Closing, and Locking the Database	open update openMode isReadOnly setReadOnly lock close
Modifying the Database	update setReadOnly change changePartition (*FTO*) tidy
Getting Information About the Database	name fileName pathName hostName numContObjs getImageFileName (*DRO*) getImageHostName (*DRO*) getImagePathName (*DRO*) getImageWeight (*DRO*) numImages (*DRO*) typeN typeName print *(inherited)* sprint *(inherited)*

Testing the Database	exist isReadOnly hasImageIn (*DRO*) isAvailable (*FTO*) isImageAvailable (*DRO*) isReplicated (*DRO*)
Testing the Handle or Object Reference	is_null *(inherited)* isNull *(inherited)* isValid operator int *(inherited)* operator ooObj* *(inherited)*
Finding Objects	exist open contains getDefaultContObj containedIn containingPartition (*FTO*) getTieBreaker (*DRO*)
Converting Objects	convertObjects
Working With Containers	contains getDefaultContObj numContObjs
Working With Autonomous Partitions	changePartition (*FTO*) containingPartition (*FTO*) isAvailable (*FTO*)
Working With Database Images	replicate (*DRO*) isReplicated (*DRO*) numImages (*DRO*) setImageWeight (*DRO*) setTieBreaker (*DRO*) negotiateQuorum (*DRO*) isNonQuorumRead (*DRO*) getAllowNonQuorumRead (*DRO*) setAllowNonQuorumRead (*DRO*) partitionsContainingImage (*DRO*) hasImageIn (*DRO*) getImageFileName (*DRO*) getImageHostName (*DRO*) getImagePathName (*DRO*) getImageWeight (*DRO*) getTieBreaker (*DRO*) isImageAvailable (*DRO*) deleteImage (*DRO*)

Reference Index

getImagePathName	(DRO) Gets the pathname of the directory that contains the specified image of the referenced database.
getImageWeight	(DRO) Gets the weight of the specified image of the referenced database.
getTieBreaker	(DRO) Finds the tie-breaker partition for the referenced database.
hasImageIn	(DRO) Tests whether the specified autonomous partition contains an image of the referenced database.
hostName	Gets the network name of the data server host that contains the referenced database.
isAvailable	(FTO) Tests whether the current process can access the referenced database.
isImageAvailable	(DRO) Tests whether the current process can access the specified image of the referenced database.
isNonQuorumRead	(DRO) Tests whether the application is currently reading the referenced database without having a quorum of images.
isReadOnly	Tests whether the referenced database is a read-only database.
isReplicated	(DRO) Tests whether the referenced database has more than one image.
isValid	Tests whether this object reference or handle is valid—that is, whether it references an existing database.
lock	Explicitly locks the referenced database.
name	Gets the system name of the referenced database.
negotiateQuorum	(DRO) Forces recalculation of the quorum for the referenced database.
numContObjs	Gets the number of containers in the referenced database.
numImages	(DRO) Gets the number of images of the referenced database.
ooHandle(ooDBObj)	Default constructor that constructs a null handle.

ooHandle(ooDBObj)	Constructs a handle that references the same database as the specified object reference or handle.
ooRef(ooDBObj)	Default constructor that constructs a null object reference.
ooRef(ooDBObj)	Constructs an object reference that references the same database as the specified object reference or handle.
open	Explicitly opens the referenced or specified database, preparing the database for the specified level of access.
openMode	Gets the current level of access to the referenced database.
operator=	Assignment operator; sets this object reference or handle to reference the same database as the specified object reference or handle.
partitionsContainingImage	(DRO) Initializes an object iterator to find all the autonomous partitions that contain an image of the referenced database.
pathName	Gets the pathname of the directory that contains the referenced database.
replicate	(DRO) Creates an image of the referenced database.
setAllowNonQuorumRead	(DRO) Specifies whether the application can read the referenced database when a quorum of images is not available.
setImageWeight	(DRO) Sets the weight of the specified image of the referenced database.
setReadOnly	Sets the access status of the referenced database so that it is either read-only or read-write.
setTieBreaker	(DRO) Sets the tie-breaker autonomous partition for the referenced database.
tidy	(administration) Consolidates fragmented storage space in the referenced database; used only in custom administration tools.
typeN	Gets the type number of the database class ooDBObj.

typeName	Gets the name of the database class ooDBObj.
update	Opens the referenced database for update access.

Constructors

ooHandle(ooDBObj)

Default constructor that constructs a null handle.

```
ooHandle(ooDBObj)();
```

ooHandle(ooDBObj)

Constructs a handle that references the same database as the specified object reference or handle.

```
ooHandle(ooDBObj)(
     const ooRefHandle(ooDBObj) &existing);
```

Parameters *existing*

Object reference or handle to an existing database.

ooRef(ooDBObj)

Default constructor that constructs a null object reference.

```
ooRef(ooDBObj)();
```

ooRef(ooDBObj)

Constructs an object reference that references the same database as the specified object reference or handle.

```
ooRef(ooDBObj)(
     const ooRefHandle(ooDBObj) &existing);
```

Parameters *existing*

Object reference or handle to an existing database.

Operators

operator=

Assignment operator; sets this object reference or handle to reference the same database as the specified object reference or handle.

```
ooRefHandle(ooDBObj) &operator=(
     const ooRefHandle(ooDBObj) &existing);
```

Parameters

existing

Object reference or handle to an existing database.

Returns

This object reference or handle.

Member Functions

change

(*administration*) Changes the location attributes of the referenced database.

```
ooStatus change(
     const char *dbSysName = 0,
     const char *dbHostName,
     const char *dbPathName,
     FILE* outputFile = stdout) const;
```

Parameters

dbSysName

New system name of the database. *This feature is currently not implemented.* Always pass the value of zero for this parameter.

dbHostName

Name of the data server host on which the database is to reside. You must specify this parameter, even when relocating the database file on the same host.

dbPathName

New pathname for the database file on *dbHostName*. You must include the database's filename as the last component of the pathname.

outputFile

Pointer to a transcript file in which to report the original and changed database attributes. The default is standard output.

Returns `oocSuccess` if successful; otherwise `oocError`.

Discussion Together, the parameters *`dbHostName`* and *`dbPathName`* determine the network address of the database file.

This member function updates the database's host and pathname in the federated database catalog, but does not actually move or rename the database file on your file system. You must use appropriate operating system commands to actually move the database file. The `change` member function is similar to the `oochangedb` tool, except that the tool also moves or renames the database file in the file system (see the Objectivity / DB administration book).

This member function requests an exclusive update lock on the referenced database. If it cannot obtain the lock, the member function both returns `oocError` and signals an error.

If the referenced database is read-only, you must use the `setReadOnly` member function to change it back to read-write before you can change its attributes.

(*DRO*) An error is signalled if the database is replicated (has more than one image).

Example This code sets the database's host name to `myHost` and pathname to `/mnt/john/design/adder.ecad.DB`.

```
ooHandle(ooDBObj) dbH;
...     // Set dbH to reference a database
dbH.change(0, "myHost", "/mnt/john/design/adder.ecad.DB");
```

changePartition

(*FTO*) Changes the autonomous partition that contains the referenced database.

```
ooStatus changePartition(
     const ooRefHandle(ooAPObj) &newPartition) const;
```

Parameters *`newPartition`*

Object reference or handle to the autonomous partition that is to contain the database.

Returns `oocSuccess` if successful; otherwise `oocError`.

Discussion This member function assigns the referenced database to the specified autonomous partition, removing the database from the current containing partition.

This member function updates the containment information in the catalogs of the autonomous partitions. If you also want to change the physical location of a database file, you must do so using the `oochangedb` tool.

You may not combine this operation with other updates on the database in the same transaction. You should consider executing `changePartition` in a separate transaction.

If the referenced database is read-only, you must use the `setReadOnly` member function to change it back to read-write before you can change its partition.

(*DRO*) If multiple images of the database exist, an error is signalled, and the current containing partition is left unchanged.

close

Internal use only. Objectivity / DB closes a database automatically when the transaction that opened it commits or aborts.

```
ooStatus close() const;
```

containedIn

Finds the federated database that contains the referenced database.

```
1.  ooHandle(ooFDObj) containedIn() const;
2.  ooRef(ooFDObj) &containedIn(
      ooRef(ooFDObj) &returnedFD) const;
3.  ooHandle(ooFDObj) &containedIn(
      ooHandle(ooFDObj) &returnedFD) const;
```

Parameters *returnedFD*

Object reference or handle to be set to the federated database.

Returns Object reference or handle to the federated database.

Discussion When called without a *returnedFD* parameter, `containedIn` allocates a new federated-database handle and returns it. Otherwise, `containedIn` returns the object reference or handle that was passed to it.

containingPartition

(*FTO*) Finds the autonomous partition that contains the referenced database.

```
1.  ooHandle(ooAPObj) containingPartition() const;
2.  ooRef(ooAPObj) &containingPartition(
      ooRef(ooAPObj) &returnedAP) const;
3.  ooHandle(ooAPObj) &containingPartition(
      ooHandle(ooAPObj) &returnedAP) const;
```

Parameters *returnedAP*

Object reference or handle to be set to the containing autonomous partition.

Returns An object reference or handle to the autonomous partition that contains the referenced database.

(*DRO*) If multiple images of the database exist, a null handle is returned and an error is signalled.

Discussion When called without a parameter, `containingPartition` allocates a new autonomous-partition handle and returns it. Otherwise, `containingPartition` returns the object reference or handle that was passed to it.

contains

(*FTO*) Initializes an object iterator to find all containers in the referenced database.

```
ooStatus contains(
     ooItr(ooContObj) &returnedConts,
     const ooMode openMode = oocNoOpen,
     const ooContainsFilter whichConts = oocAllObjs);
```

Parameters *returnedConts*

Object iterator for finding the database's containers.

openMode

Intended level of access to the containers found by the iterator's `next` member function:

- `oocNoOpen` (the default) causes `next` to set the iterator to the next container without opening it.
- `oocRead` causes `next` to open the next container for read.
- `oocUpdate` causes `next` to open the next container for update.

whichConts

Filters the containers to be found by the iterator:

- Specify `oocAllObjs` (the default) to initialize the iterator with all containers contained in the database.
- (*FTO*) Specify `oocNotTransferred` to initialize the iterator with containers that have the same controlling autonomous partition as the database.
- (*FTO*) Specify `oocTransferred` to initialize the iterator with those containers that have a different controlling autonomous partition than the database.

Returns `oocSuccess` if successful; otherwise `oocError`.

convertObjects

Performs on-demand object conversion on any affected objects in the referenced database.

```
ooStatus convertObjects();
```

Returns `oocSuccess` if successful, or `oocError` if the federated database is opened only for read.

Discussion Object conversion is the process of making existing persistent objects consistent with class definition changes introduced by schema evolution. Certain schema evolution operations affect how instances of a class should be laid out in storage. After you perform such operations, existing objects of the changed classes are rendered out-of-date until they are converted to their new representations.

In general, you can allow each affected object to be converted automatically the first time it is accessed after schema evolution, potentially distributing the performance impact of conversion across many transactions. Alternatively, you can concentrate the performance impact of conversion into fewer transactions by converting all the affected objects in a container, a database, or a federated database *on demand*. You use this member function in an update transaction to convert the affected objects in a database on demand. This member function has no effect if the affected objects in the database have already been converted.

Note: On-demand object conversion cannot be used for schema operations that require an upgrade application; see `ooTrans::upgrade`.

The `convertObjects` member function automatically drops any index that is invalidated by a schema evolution change. Specifically, if you changed the type or deleted a data member that is a key field in a key description, the corresponding indexes are dropped.

If the referenced database is read-only, you must use the `setReadOnly` member function to change it back to read-write before you can convert objects in it.

See also Chapter 19, “Object Conversion,” in the Objectivity / C++ programmer’s guide

deleteImage

(*DRO*) Deletes the specified autonomous partition’s image of the referenced database.

```
ooStatus deleteImage(
     const ooRefHandle(ooAPObj) &partition,
     ooBoolean deleteDBifLast = oocFalse) const;
```

Parameters

partition

Object reference or handle to the autonomous partition containing the database image.

deleteDBifLast

Specifies whether to delete the database image even if no other images remain:

- Specify `oocTrue` to delete the last image and the database itself.
- Specify `oocFalse` (the default) to preserve the last image and signal an error.

Returns

`oocSuccess` if successful; otherwise `oocError`.

Discussion

An error is signalled if either of the following are true:

- The referenced database has no image in the specified autonomous partition.
- If the referenced database is read-only. You must use the `setReadOnly` member function to change the database back to read-write before you can delete an individual image.

exist

Tests whether the specified database exists in the federated database; if successful, sets this object reference or handle to reference the database.

```
ooBoolean exist(
     const ooHandle(ooFDObj) &fdH,
     const char *dbSysName,
     const ooMode openMode = oocNoOpen);
```

Parameters

fdH

Handle to the federated database.

dbSysName

System name of the database to be found.

openMode

Intended level of access to the database, if it exists:

- Specify `oocNoOpen` (the default), to set this object reference or handle to the database without opening it.
- Specify `oocRead` to open the database for read.
- Specify `oocUpdate` to open the database for update.

Returns

`oocTrue` if the specified database exists, or `oocFalse` if the database does not exist or if it is not accessible.

Discussion If the specified database exists, this object reference or handle is set to reference it; otherwise, this object reference or handle is set to null.

If you specifically want to test for existence, you use the *openMode* parameter's default value (*oocNoOpen*). Otherwise, a return value of *oocFalse* could mean either that the database doesn't exist, or that it does exist, but cannot be opened.

fileName

Gets the fully qualified filename of the referenced database.

```
char *fileName() const;
```

Returns Pointer to a string containing the filename.

Discussion The returned filename includes the directory pathname as well as the simple name of the database file. Use `pathName` to obtain just the directory pathname.

The returned string is statically allocated by the member function and is overwritten with each invocation. You should make a local copy of the returned string if you intend to use it later in the application.

(*DRO*) An error is signalled if multiple images of the database exist. Use `getImageFileName` instead.

Example This example prints the filename of the database referenced by `dbH`. On UNIX, the output of this example might be: `/mnt/john/design/testDb.testFd.DB`.

```
ooHandle(ooFDObj) fdH = fdH.open("testFd", oocRead);
ooHandle(ooDBObj) dbH = dbH.open(fdH, "testDb", oocRead);

printf("filename: %s\n", dbH.fileName());
```

getAllowNonQuorumRead

(*DRO*) Tests whether this application is allowed to read the referenced database when a quorum of images is not available.

```
ooBoolean getAllowNonQuorumRead() const;
```

Returns `oocTrue` if nonquorum reads are allowed; otherwise `oocFalse`.

See also `isNonQuorumRead`
`setAllowNonQuorumRead`

getDefaultContObj

Finds the default container of the referenced database.

```
1.  ooHandle(ooContObj) getDefaultContObj(
      const ooMode openMode = oocNoOpen) const;
2.  ooRef(ooContObj) &getDefaultContObj(
      ooRef(ooContObj) &returnedCont,
      const ooMode openMode = oocNoOpen) const;
3.  ooHandle(ooContObj) &getDefaultContObj(
      ooHandle(ooContObj) &returnedCont,
      const ooMode openMode = oocNoOpen) const;
```

Parameters

returnedCont

Object reference or handle to be set to the default container.

openMode

Intended level of access to the default container:

- Specify `oocRead` (the default) to open the default container for read.
- Specify `oocUpdate` to open the default container for update.
- Specify `oocNoOpen` to set the object reference or handle reference to the default container without opening it.

Returns

Object reference or handle to the default container of the database.

Discussion

When called without a *returnedCont* parameter, this member function allocates a new container handle and returns it. Otherwise, this member function returns the object reference or handle that was passed to it.

getImageFileName

(*DRO*) Gets the fully qualified filename of the specified image of the referenced database.

```
const char *getImageFileName(
     const ooRefHandle(ooAPObj) &partition) const;
```

Parameters

partition

Object reference or handle to the autonomous partition that contains the desired database image.

Returns

Pointer to a string containing the database image's filename.

Discussion The returned filename includes the directory pathname as well as the simple name of the database image file. Use `getImagePathName` to obtain just the directory pathname.

The returned string is statically allocated by the member function and is overwritten with each invocation. You should make a local copy of the returned string if you intend to use it later in the application.

An error is signalled if the specified autonomous partition does not contain an image of the referenced database.

getImageHostName

(*DRO*) Gets the name of the data server host that contains the specified image of the referenced database.

```
const char *getImageHostName(
     const ooRefHandle(ooAPObj) &partition) const;
```

Parameters *partition*

Object reference or handle to the autonomous partition that contains the desired database image.

Returns Pointer to a string containing the network name of the data server host where the database image file is located.

Discussion The string is statically allocated by the member function and is overwritten with each invocation. You should make a local copy of the returned string if you intend to use it later in the application.

An error is signalled if the specified autonomous partition does not contain an image of the referenced database.

getImagePathName

(*DRO*) Gets the pathname of the directory that contains the specified image of the referenced database.

```
const char *getImagePathName(
     const ooRefHandle(ooAPObj) &partition) const;
```

Parameters *partition*

Object reference or handle to the autonomous partition that contains the desired database image.

Returns Pointer to a string containing the directory pathname.

Discussion The returned pathname does not include the simple name of the database file; use `getImageFileName` to obtain a path name that includes the filename.

The string is statically allocated by the member function and is overwritten with each invocation. You should copy the string if you wish to use it later in your application.

An error is signalled if the specified autonomous partition does not contain an image of the referenced database.

getImageWeight

(*DRO*) Gets the weight of the specified image of the referenced database.

```
uint32 getImageWeight(
     const ooRefHandle(ooAPObj) &partition) const;
```

Parameters *partition*

Object reference or handle to the autonomous partition that contains the desired database image.

Returns The weight of the database image; returns 0 if the referenced database is not replicated in the specified autonomous partition.

getTieBreaker

(*DRO*) Finds the tie-breaker partition for the referenced database.

```
ooHandle(ooAPObj) getTieBreaker() const;
```

Returns Handle to the tie-breaker partition if one exists; otherwise, a null handle.

Discussion This member function allocates an autonomous-partition handle and returns it.

hasImageIn

(*DRO*) Tests whether the specified autonomous partition contains an image of the referenced database.

```
ooBoolean hasImageIn(
     const ooRefHandle(ooAPObj) &partition) const;
```

Parameters *partition*

Object reference or handle to the autonomous partition to be searched.

Returns `oocTrue` if the partition contains an image of the referenced database; otherwise `oocFalse`.

hostName

Gets the network name of the data server host that contains the referenced database.

```
char *hostName() const;
```

Returns Pointer to a string containing the network name of the data server host.

Discussion The returned string is statically allocated by the member function and is overwritten with each invocation. You should make a local copy of the returned string if you intend to use it later in the application.

(*DRO*) An error is signalled if multiple images of the database exist. Use `getImageHostName` instead.

Example This example prints the host name for the database referenced by `dbH`. The output of this example might be: `myMachine`.

```
ooHandle(ooFDObj) fdH;
ooHandle(ooDBObj) dbH;

fdH.open("testFd", oocRead);
dbH.open(fdH, "testDb", oocRead);

printf("hostname: %s\n", dbH.hostName());
```

isAvailable

(*FTO*) Tests whether the current process can access the referenced database.

```
ooBoolean isAvailable() const;
```

Returns `oocTrue` if the current process can access the database; otherwise `oocFalse`.

(*FTO*) If the database is not replicated, it is accessible if the process can access its containing partition.

(*DRO*) If the database is replicated, it is accessible if the process can access a quorum of its images.

isImageAvailable

(*DRO*) Tests whether the current process can access the specified image of the referenced database.

```
ooBoolean isImageAvailable(
     const ooRefHandle(ooAPObj) &partition) const;
```

Parameters *partition*

Object reference or handle to the autonomous partition containing the desired image.

Returns `oocTrue` if the image in the specified autonomous partition is available; otherwise `oocFalse`.

isNonQuorumRead

(*DRO*) Tests whether the application is currently reading the referenced database without having a quorum of images.

```
ooBoolean isNonQuorumRead() const;
```

Returns `oocTrue` if the application is reading the referenced database without having a quorum of images; otherwise `oocFalse`.

Discussion Nonquorum reading of a database is possible only after an application has called the `setAllowNonQuorumRead` member function for the database during the current transaction.

WARNING If this member function returns `oocTrue`, your application may be reading stale data from the database.

See also `getAllowNonQuorumRead`
`setAllowNonQuorumRead`

isReadOnly

Tests whether the referenced database is a read-only database.

```
ooBoolean isReadOnly() const;
```

Returns `oocTrue` if this object reference or handle references a read-only database; otherwise `oocFalse`.

Discussion A read-only database can be opened only for read; any attempt to implicitly or explicitly open the database for update will fail as if there were a lock conflict.

See also `setReadOnly`

isReplicated

(*DRO*) Tests whether the referenced database has more than one image.

```
ooBoolean isReplicated() const;
```

Returns `oocTrue` if the referenced database is replicated; otherwise `oocFalse`.

isValid

Tests whether this object reference or handle is valid—that is, whether it references an existing database.

```
ooBoolean isValid() const;
```

Returns `oocTrue` if this object reference or handle references an existing database; `oocFalse` if this object reference or handle is null or has a stale identifier, or if the application cannot obtain a read lock on the database to be checked.

Discussion You can use `isValid` to determine whether it is safe to use an object reference or handle that was set in a previous transaction. Such an object reference or handle still retains its reference to a database; however, between transactions, that reference may have become invalid (for example, because another process has deleted the database).

NOTE `isValid` checks only for the existence of a database with a particular identifier, but has no way of knowing whether it is the same database. It is possible, although very unlikely, for another process to have deleted the original database and created a new one with the same identifier.

If your purpose is simply to test whether an object reference or handle has been initialized, it is more efficient to use `isNull`, which performs its test entirely in memory without having to access files on disk.

lock

Explicitly locks the referenced database.

```
ooStatus lock(const ooLockMode lockMode) const;
```

Parameters *`lockMode`*

Type of lock to request:

- Specify `oocLockRead` to request a read lock.
- Specify `oocLockUpdate` to request an update lock.

Returns `oocSuccess` if the requested lock is obtained; otherwise `oocError`.

Discussion Objectivity / DB operations request and obtain locks implicitly as they are needed. You use this member function to obtain a lock explicitly when you want to reserve access to a database in advance.

Explicitly locking a database essentially limits the level of concurrent access to its containers, allowing them to be read, but guaranteeing they will not change while the lock is held. That is, explicitly locking a database for either read or update:

- Prevents any other transaction from concurrently opening the database or a container in it for update.
- Allows any other transaction to concurrently open the database or a container in it for read.

A database cannot be locked if it, or any container in it, is already opened for update.

NOTE Holding an update lock on a database does *not* guarantee update access to the database's individual containers. For example, if transaction T1 locks a database for update, and then a standard (non-MROW) transaction T2 opens a container in the database for read, T1 cannot concurrently get an update lock on the container being read.

When a database is locked for read, other transactions can concurrently lock it for read. When a database is locked for update, MROW transactions can concurrently lock it for read, but standard transactions cannot. Two transactions cannot lock the same database for update.

name

Gets the system name of the referenced database.

```
char *name() const;
```

Returns Pointer to a string containing the system name.

Discussion The string is statically allocated by the member function and is overwritten with each invocation. You should make a local copy of the returned string if you intend to use it later in the application.

negotiateQuorum

(*DRO*) Forces recalculation of the quorum for the referenced database.

```
ooStatus negotiateQuorum(ooMode openMode);
```

Parameters *openMode*

Intended level of access to this database:

- Specify `oocRead` to open and implicitly lock the database for read.
- Specify `oocUpdate` to open and implicitly lock the database for update (read and write).

Returns `oocSuccess` if successful; otherwise `oocError`.

Discussion If the referenced database has any images in previously unavailable partitions, this member function adds those images back into the quorum for this database. The reacquired images are resynchronized—that is, they are updated to be consistent with the images in the quorum.

If your application includes a reinitialization procedure that is executed when an autonomous partition that was down is brought back up, that procedure should call this member function for every database with an image in the restored autonomous partition.

This member function returns `oocError` and performs no action if any of the following is true:

- If the application does not have access to a quorum of images for the referenced database.
- If any partition containing an image for this database is still unavailable—for example, if the partition's lock server or AMS has not been restarted yet.

If the referenced database is not replicated, this member function returns `oocSuccess` and performs no other action.

You may not recalculate a quorum if nonquorum reading has been enabled for the referenced database (see `setAllowNonQuorumRead`). To recalculate a quorum in this case, you must end the current transaction (to disable nonquorum reading), and then call `negotiateQuorum` from within a new transaction.

numContObjs

Gets the number of containers in the referenced database.

```
unsigned long numContObjs() const;
```

Returns Number of containers.

numImages

(*DRO*) Gets the number of images of the referenced database.

```
uint32 numImages() const;
```

Returns Number of images.

Discussion For any existing database, there is always at least one image.

open

Explicitly opens the referenced or specified database, preparing the database for the specified level of access.

```
1.  ooStatus open(
      const ooMode openMode = oocRead);
2.  ooStatus open(
      const ooHandle(ooFDObj) &fdH,
      const char *dbSysName,
      const ooMode openMode = oocRead);
```

Parameters *openMode*

Intended level of access to the opened database:

- Specify `oocRead` (the default) to open and implicitly lock the database for read.
- Specify `oocUpdate` to open and implicitly lock the database for update (read and write).

fdH

Handle to the currently open federated database.

dbSysName

System name of the database to open.

Returns `oocSuccess` if successful; otherwise `oocError`.

Discussion Variant 1 assumes that this object reference or handle already references a database, and opens the referenced database.

Variant 2 finds and explicitly opens the database with the specified system name, and sets this object reference or handle to reference it. An error is signalled if no database exists with the specified system name or if the database file cannot be found or accessed. This variant is especially useful when you want to use the database as an entry point into your data. For example, you might find and open a database so you can iterate over the containers in it.

Opening a database makes it available to an application by locating and opening the database file, provided that appropriate access permissions are set on it.

It is normally not necessary to open databases explicitly because they are usually opened automatically by operations that access them or their contents. For example, once you have a reference to a database, creating a container in the database automatically opens it for update. In general, you open a referenced database explicitly only when you want to guarantee access to the database in advance—for example, before starting a complex operation.

Any number of transactions can concurrently open the same database in any mode. However, a database cannot be opened for update if another transaction already has a read or update lock on it.

You must be in an update transaction to open a database for update. If necessary, you can promote a read transaction to an update transaction by promoting the open mode of the federated database.

See also `update`

openMode

Gets the current level of access to the referenced database.

```
ooMode openMode() const;
```

Returns One of the following constants:

- `oocNoOpen`—the database is not open in this transaction.
- `oocRead`—the database is open for read in this transaction.
- `oocUpdate`—the database is open for update in this transaction.

partitionsContainingImage

(*DRO*) Initializes an object iterator to find all the autonomous partitions that contain an image of the referenced database.

```
ooStatus partitionsContainingImage(
    ooItr(ooAPObj) &apI) const;
```

Parameters *apI*

Object iterator for finding the containing autonomous partitions.

Returns `oocSuccess` if successful; otherwise `oocError`.

Discussion

If the referenced database is not replicated in any other autonomous partition, the iterator finds the partition in which the database resides. The iterator's `next` member function finds each autonomous partition without opening or locking it.

pathName

Gets the pathname of the directory that contains the referenced database.

```
char *pathName() const;
```

Returns

Pointer to a string containing the pathname of the directory.

Discussion

The returned pathname does not include the simple name of the database file; use `fileName` to obtain a path name that includes the filename.

The string is statically allocated by the member function and is overwritten with each invocation. You should copy the string if you wish to use it later in your application.

(*DRO*) An error is signalled if multiple images of the database exist. Use `getImagePathName` instead.

Example

This example prints the directory pathname of the database referenced by `dbH`. On UNIX, the output of this example might be: `/mnt/john/design`.

```
ooHandle(ooFDObj) fdH;
ooHandle(ooDBObj) dbH;

fdH.open("testFd", oocRead);
dbH.open(fdH, "testDb", oocRead);

printf("pathname: %s\n", dbH.pathName());
```

replicate

(*DRO*) Creates an image of the referenced database.

```
ooStatus replicate(
     const ooRefHandle(ooAPObj) &partition,
     const char *hostName = 0,
     const char *pathName = 0,
     uint32 weight = 1) const;
```

Parameters

partition

Object reference or handle to the autonomous partition in which to create the new database image. This partition may not already contain an image of this database.

hostName

Name of the data server host on which to create the new database image. If this parameter is omitted, the host will be the same as that hosting the partition's system database file. The host machine must be running AMS.

pathName

Fully qualified pathname (including the filename) of the new database image. The format of the pathname must follow the naming conventions of the specified host. If this parameter is omitted, the image's file is created with a name based on this database's system name and is placed in the same directory as the partition's system database file.

weight

Weight of the new database image. *weight* must be 1 or greater.

Returns `oocSuccess` if successful; otherwise `oocError`.

Discussion If neither *hostName* or *pathName* is specified, the database image is created in the same directory as the autonomous partition's system database file, and the image's file name is generated from the database's system name.

An error is signalled if any of the following are true:

- If the referenced database already has an image in the specified autonomous partition.
- If AMS is not running on the data server hosts where the original and the new database images reside. For information about AMS, see the Objectivity / DB administration book.
- If the referenced database has been made read-only with the `setReadOnly` member function. You must change the database back to read-write before you can add a new image.

setAllowNonQuorumRead

(*DRO*) Specifies whether the application can read the referenced database when a quorum of images is not available.

```
ooStatus setAllowNonQuorumRead(
     ooBoolean value = oocTrue) const;
```

Parameters *value*

Specify `oocTrue` (the default) to allow this application to read the referenced database even if a quorum is not available; specify `oocFalse` to prevent this application from reading the database when a quorum is not available.

Returns `oocSuccess` if successful; otherwise `oocError`.

Discussion By default, nonquorum reading is disabled for all databases. You must call this member function explicitly to permit nonquorum reading for a particular database during a particular transaction. Nonquorum reading is automatically disabled at the end of the transaction in which this member function is called.

WARNING If you enable nonquorum reading, your application may read stale data from the database.

See also `getAllowNonQuorumRead`
`isNonQuorumRead`

setImageWeight

(*DRO*) Sets the weight of the specified image of the referenced database.

```
ooStatus setImageWeight(
     const ooRefHandle(ooAPObj) &partition,
     uint32 weight) const;
```

Parameters *partition*

Object reference or handle to the autonomous partition containing the desired image.

weight

Weight to be set for the database image. *weight* must be 1 or greater.

Returns `oocSuccess` if successful; otherwise `oocError`.

Discussion An error is signalled if either of the following are true:

- The referenced database is not replicated in the specified autonomous partition.
- If the referenced database is read-only. You must use the `setReadOnly` member function to change the database back to read-write before you can set an image's weight.

setReadOnly

Sets the access status of the referenced database so that it is either read-only or read-write.

```
ooStatus setReadOnly(ooBoolean value) const;
```

Returns `oocSuccess` if successful; otherwise `oocError`.

Parameters *value*

Specify `oocTrue` to mark the database as read-only; specify `oocFalse` to change the database back to read-write.

Discussion A read-only database can be opened only for read. Any attempt to implicitly or explicitly open the database for update will fail as if there were a lock conflict. Using a read-only database can improve performance by allowing the application to grant read locks and deny update locks without consulting the lock server.

You can change a read-only database back to read-write only if no application or tool is currently reading either that database or any other read-only database in the same federated database. You must change the database back to read-write before you can perform any other operation on it.

(*DRO*) If one image of a database is made read-only, all images are automatically made read-only. Similarly, if one image is changed back to read-write, all images are changed back to read-write.

See also `isReadOnly`

setTieBreaker

(*DRO*) Sets the tie-breaker autonomous partition for the referenced database.

```
ooStatus setTieBreaker(
     const ooRefHandle(ooAPObj) &partition) const;
```

Parameters *partition*

Object reference or handle to the tie-breaker autonomous partition. This partition may not already contain an image of this database. If *partition* is 0, any existing tie-breaker is eliminated.

Returns `oocSuccess` if successful; otherwise `oocError`.

tidy

(*administration*) Consolidates fragmented storage space in the referenced database; used only in custom administration tools.

```
ooStatus tidy(
     FILE* outputFile = stdout,
     const char *hostName = 0,
     const char *pathName = 0) const;
```

Parameters

outputFile

Pointer to the transcript file in which to write the generated report. The default is standard output.

hostName

Name of the data server host on which to create the temporary file.

pathName

Pathname of the directory in which to create the temporary file.

Returns

`oocSuccess` if successful; otherwise `oocError`.

Discussion

This member function creates a temporary database file during execution and therefore requires free disk space equal to the size of the database file being tidied. The temporary file is created in the directory that contains the database file unless you specify *hostName* and *pathName*.

If either *hostName* or *pathName* is zero or empty (""), the temporary file is created in the default directory.

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.

You must not abort the transaction after calling this member function.

This member function performs the same function as the `ootidy` tool (see the Objectivity / DB administration book).

typeN

Gets the type number of the database class `ooDBObj`.

```
ooTypeNumber typeN() const;
```

Returns

Type number of the database class `ooDBObj`.

typeName

Gets the name of the database class `ooDBObj`.

```
char *typeName() const;
```

Returns

The string "`ooDBObj`".

Discussion

The returned string must be treated as read-only.

update

Opens the referenced database for update access.

```
ooStatus update();
```

Returns `oocSuccess` if successful; otherwise `oocError`.

Discussion This member function is equivalent to calling open(`oocUpdate`).

See also open

ooRefHandle(ooFDObj) Classes

Inheritance: **ooRef(ooObj)->ooRef(ooFDObj)**

Inheritance: **ooHandle(ooObj)->ooHandle(ooFDObj)**

The abbreviation *ooRefHandle*(ooFDObj) refers to two non-persistence-capable classes:

- ooRef(ooFDObj), which represents an *object reference* to the federated database.
- ooHandle(ooFDObj), which represents a *handle* to the federated database.

The two classes ooRef(ooFDObj) and ooHandle(ooFDObj) are documented together because they define the same set of member functions. These member functions provide the primary interface for operating on federated databases (instances of ooFDObj).

See:

- "Reference Summary" on page 575 for an overview of member functions
- "Reference Index" on page 576 for a list of member functions

About Federated-Database Handles and References

An application that is connected to a particular federated database can work with that federated database indirectly through one or more handles or object references—that is, through instances of *ooRefHandle*(ooFDObj) that are set to reference the federated database being accessed. Such handles and object references both identify the federated database and provide the complete public interface for operating on it.

Handles and object references to federated databases do not support indirect member access; the *ooRefHandle*(ooFDObj) classes provide no indirect

member-access operator (->), and the ooFDObj class defines no public member functions other than a constructor and operator new.

You can work with a federated database through either a handle or an object reference—the choice is arbitrary, except as described in "Structure and Behavior" on page 572. Most applications use handles rather than object references.

Interface

The *ooRefHandle*(ooFDObj) classes provide the primary interface for operating on a federated database. Part of this interface consists of member functions defined by *ooRefHandle*(ooFDObj) for specialized operations such as tidying. The other part of this interface consists of member functions defined by the *ooRefHandle*(ooObj) base classes for more general Objectivity / DB operations, such as opening, locking, printing object identifiers, and so on. These member functions are either inherited by the *ooRefHandle*(ooFDObj) classes or redefined wherever type-specific parameters or behavior are required.

Some of the member functions defined by the *ooRefHandle*(ooObj) base classes are *not* available to instances of the *ooRefHandle*(ooFDObj) classes. These include member functions for moving, copying, versioning, scope-naming, and member-access operations, which apply only to basic objects or persistent objects. The disallowed member functions and operators are redefined as private members of the *ooRefHandle*(ooFDObj) classes.

Structure and Behavior

Handles and object references to a federated database are essentially wrappers for the federated database's identifier. Thus, when an application opens a federated database through a handle, the handle stores that federated database's identifier. The handle uses the identifier to find the instance of ooFDObj that represents the federated database in memory.

In general, object references are optimized for implementing links among related persistent objects, while handles are optimized for memory management and member-access. When a federated database is referenced, however, these optimizations are largely irrelevant, because federated databases (unlike persistent objects):

- Cannot be linked (for example, through associations).
- Are not subject to memory management (they have no attributes for persistent data and are therefore not manipulated through pointers).
- Have no accessible members.

One significant exception is that a handle to a federated database, like any handle, contains cache-related state that is associated with the Objectivity context

in which it was created. Therefore, only object references (but not handles) can be passed between Objectivity contexts. Otherwise, a handle to a federated database and an object reference to a federated database are functionally equivalent.

Working With Federated-Database Handles

NOTE For simplicity, this section describes how to work with handles. Except where noted, the same information applies to object references.

An application normally creates a handle as a local variable on the stack, rather than allocating it on the heap. A handle should not be declared as `const`, because its internal state may be changed when a federated database is accessed through it. (Object references may be declared as `const`). Applications should not create subclasses of the *`ooRefHandle`*`(ooFDObj)` classes.

A new handle is normally *null*—that is, it contains the value 0 instead of a federated database identifier. The application can set the handle to reference a federated database in any of the following ways:

- By connecting to the federated database with the handle's `exist` or `open` member function. (Within a particular application, the *same* federated database must be specified in all such operations.)
- By passing the handle to a member function that sets it, such as the `containedIn` member function of a database handle, which finds the federated database from a database in it.
- By assignment or initialization from another handle or object reference.

A handle continues to reference the federated database unless it is set to null; it cannot be set to any other federated database during the application. Multiple handles and object references can be set to the same federated database.

An application operates on the federated database by calling member functions on a handle that references it. To call a member function of a handle, you use the direct member-access operator (`.`). For example, `fdH.name` calls the `dumpCatalog` member function of the handle `fdH`.

As indicated in "Reference Summary" on page 575, an application can get and change the federated database's attributes, find its databases, tidy its disk space, and so on.

Although most of the handle's member functions operate on the referenced federated database, some functions operate on the handle itself. For example, you use:

- The assignment operator `operator=` to set a handle from another handle or from an object reference.
- The inherited comparison operators `operator==` and `operator!=` to test whether a handle is equal to another handle or object reference.
- The inherited member function `isNull` to test whether a handle is null. (Alternatively, you can use the overloaded `operator==` to compare a handle to 0.)

A handle preserves its reference to a federated database across transaction boundaries, provided that the handle does not go out of scope and is not set to null as the result of an abort operation.

Objectivity/C++ functions that require a federated database as input normally obtain the federated database through a parameter of type `const ooHandle(ooFDObj) &`. If a function manipulates other types of Objectivity/DB objects as well, the parameter type may be specified as `const ooHandle(ooObj) &`. You can pass a federated-database handle to a parameter of this type, because `ooHandle(ooFDObj)` is derived from `ooHandle(ooObj)`. In practice, however, relatively few functions accept a federated-database handle where a more general handle is requested; these include the functions that require a storage object for scanning or as a name scope. In most cases, a function that requests a general-purpose handle operates only on basic objects or persistent objects, and signals an error if you specify a federated-database handle.

An application must call the `open` member function on a federated-database handle to explicitly open the federated database at the beginning of every transaction; doing so verifies the connection and designates the transaction as either a read or update transaction. The handle is automatically closed when it is destroyed (for example, by going out of scope). However, closing one or more handles to the federated database has no particular effect on the federated database, which remains open until the transaction commits or aborts.

Reference Summary

The following table summarizes all the member functions that are available to instances of *ooRefHandle*(ooFDObj). Member functions indicated as *(inherited)* documented with the ooRefHandle(ooObj) classes (page 593).

Creating a Handle or Object Reference	ooHandle(ooFDObj) ooRef(ooFDObj)
Setting the Handle or Object Reference	operator= open exist
Comparing Handles and Object References	operator== *(inherited)* operator!= *(inherited)*
Opening, Closing, and Locking the Federated Database	open update openMode lock close
Modifying the Federated Database	update change
Getting Information About the Federated Database	name number pageSize lockServerName dumpCatalog typeN typeName print *(inherited)* sprint *(inherited)*
Testing the Federated Database	exist
Testing the Handle or Object Reference	is_null *(inherited)* isNull *(inherited)* isValid operator int *(inherited)* operator ooObj* *(inherited)*
Finding Objects	exist open bootAP contains

Converting Objects	convertObjects setConversion upgradeObjects
Working With Databases	contains tidy
Working With Autonomous Partitions (*FTO*)	bootAP contains

Reference Index

name	Gets the system name of the referenced federated database.
number	(administration) Gets the identifier of the referenced federated database.
ooHandle(ooFDObj)	Default constructor that constructs a null handle.
ooHandle(ooFDObj)	Constructs a handle that references the same federated database as the specified object reference or handle.
ooRef(ooFDObj)	Default constructor that constructs a null object reference.
ooRef(ooFDObj)	Constructs an object reference that references the same federated database as the specified object reference or handle.
open	Opens the specified federated database for the specified level of access and sets this object reference or handle to reference the opened federated database.
operator=	Assignment operator; sets this object reference or handle to reference the specified federated database.
openMode	Gets the current level of access to the referenced federated database.
pageSize	(administration) Gets the storage page size of the referenced federated database.
setConversion	Registers a conversion function for objects of the specified class in the referenced federated database.
tidy	(administration) Consolidates fragmented storage space in all the databases in the referenced federated database; used only in custom administration tools.
typeN	Gets the type number of the federated-database class ooFDObj.
typeName	Gets the name of the federated-database class ooFDObj.
update	Opens the referenced federated database for update access.
upgradeObjects	Performs object conversion throughout the referenced federated database; used only in a special-purpose upgrade application.

Constructors and Destructors

ooHandle(ooFDObj)

Default constructor that constructs a null handle.

```
ooHandle(ooFDObj)();
```

ooHandle(ooFDObj)

Constructs a handle that references the same federated database as the specified object reference or handle.

```
ooHandle(ooFDObj)(
     const ooRefHandle(ooFDObj) &federation);
```

Parameters *federation*

Object reference or handle to the federated database.

ooRef(ooFDObj)

Default constructor that constructs a null object reference.

```
ooRef(ooFDObj)();
```

ooRef(ooFDObj)

Constructs an object reference that references the same federated database as the specified object reference or handle.

```
ooRef(ooFDObj)(
     const ooRefHandle(ooFDObj) &federation);
```

Parameters *federation*

Object reference or handle to the federated database.

Operators

operator=

Assignment operator; sets this object reference or handle to reference the specified federated database.

```
ooRefHandle(ooFDObj) &operator=(
     const ooRefHandle(ooFDObj) federation);
```

Parameters *federation*

Object reference or handle to the federated database.

Returns This object reference or handle.

Member Functions

bootAP

(*FTO*) Finds the boot autonomous partition for the current application.

```
1.   ooHandle(ooAPObj) bootAP() const;
2.   ooRef(ooAPObj) &bootAP(
       ooRef(ooAPObj) &partition) const;
3.   ooHandle(ooAPObj) &bootAP(
       ooHandle(ooAPObj) &partition) const;
```

Parameters *partition*

Object reference or handle to be set to the boot autonomous partition.

Returns Object reference or handle to the boot autonomous partition, if successful; otherwise, returns a null handle.

Discussion When called without a parameter, `bootAP` allocates a new autonomous-partition handle and returns it. Otherwise, `bootAP` returns the object reference or handle that was passed to it.

change

(*administration*) Changes the attributes of the referenced federated database; used only in custom administration tools.

```
ooStatus change(
     const char *bootFilePath = 0,
     const char *lockServer = 0,
     const uint32 fdNumber = 0,
     FILE* outputFile = stdout) const;
```

Parameters

bootFilePath

New pathname for the boot file of the federated database. You must include the filename as the last component of the pathname. Specify 0 (the default) to leave the boot file path unchanged.

lockServer

Name of the new lock server host (the host that runs the lock server for this federated database). Specify 0 (the default) to leave the lock server host unchanged.

fdNumber

New federated database identifier. Specify 0 (the default) to leave the identifier unchanged.

outputFile

Pointer to the transcript file in which to report the original and changed federated-database attributes. The default value is standard output.

Returns

`oocSuccess` if successful; otherwise `oocError`.

Discussion

You cannot change the system name of the federated database or its storage page size. If you specify a new boot file location, the updated boot file is written to the new location, but the old boot file remains; you must delete this file using appropriate operating system commands.

You use this member function in a special-purpose application that consists of a single update transaction. The 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.

You should call `ooNoLock` so that the application will run in single-user mode. You then stop the lock server before running the application and restart it after the application completes.

WARNING You must guarantee that no other transaction has access to the federated database or data corruption could result.

Example This example changes the lock server host for the federated database to `moon`.

```
ooTrans trans;
ooHandle(ooFDObj) fdH;
…
ooInit();
ooNoLock();
trans.start();
fdH.open("Documentation", oocUpdate);
fdH.change(0, "moon"); // change the lock server
trans.commit();
exit(0);
```

close

Closes any open Objectivity / DB objects in the referenced federated database.

```
ooStatus close() const;
```

Returns `oocSuccess` if successful; otherwise `oocError`.

Discussion This member function closes all Objectivity / DB objects that have been opened in the current transaction. Closing persistent objects may be useful for purposes of memory management, because Objectivity / DB can swap the pages of closed objects out of the Objectivity / DB cache. Note, however, that locks on the closed objects are retained until the transaction commits or aborts.

You do not need to explicitly reopen the federated database before performing further operations within the same transaction. However, you cannot open a different federated database within a process even if you have closed the first one.

contains

Initializes an object iterator to find all the databases *or* all the autonomous partitions in the referenced federated database.

```
1.   ooStatus contains(
       ooItr(ooDBObj) &containedDBs,
       const ooMode openMode = oocNoOpen) const;
```

```
2.  ooStatus contains(
      ooItr(ooAPObj) &containedAPs,
      const ooMode openMode = oocNoOpen) const;
```

Parameters

containedDBs

Object iterator for finding the databases in the federated database.

(*FTO*) If the federated database has more than one autonomous partition, the iterator finds the databases contained in all the autonomous partitions.

containedAPs

(*FTO*) Object iterator for finding the autonomous partitions of the federated database.

openMode

Intended level of access to the databases or partitions found by the iterator's `next` member function:

- `oocNoOpen` (the default) causes `next` to set the iterator to the next found object without opening it.
- `oocRead` causes `next` to open the next found object for read.
- `oocUpdate` causes `next` to open the next found object for update.

Returns

`oocSuccess` if successful; otherwise `oocError`.

convertObjects

Performs on-demand object conversion on any affected objects in the referenced federated database.

```
ooStatus convertObjects(ooBoolean purge_schema = oocFalse);
```

Parameters

purge_schema

Specifies whether to remove schema-evolution history after all objects are converted. By default, this history is not purged. **Warning:** Purging the schema may delete information that is required for distributing schema changes to deployed federated databases (see Discussion below). You should take a backup of the federated database before purging its schema.

Returns

`oocSuccess` if successful, or `oocError` if the federated database is opened for read-only access.

Discussion

Object conversion is the process of making existing persistent objects consistent with class definition changes introduced by schema evolution. Certain schema evolution operations affect how instances of a class should be laid out in storage. After you perform such operations, existing objects of the changed classes are rendered out-of-date until they are converted to their new representations.

In general, you can allow each affected object to be converted automatically the first time it is accessed after schema evolution, potentially distributing the performance impact of conversion across many transactions. Alternatively, you can concentrate the performance impact of conversion into fewer transactions by converting all the affected objects in a container, a database, or a federated database *on demand*.

You use this member function in an update transaction to convert the affected objects in a federated database on demand. This member function has no effect if the affected objects in the federated database have already been converted.

NOTE On-demand object conversion cannot be used for schema operations that require an upgrade application; see `ooTrans::upgrade`.

The `convertObjects` member function automatically drops any index that is invalidated by a schema evolution change. Specifically, if you changed the type or deleted a data member that is a key field in a key description, the corresponding indexes are dropped.

If you make schema changes in a development federated database and then distribute these to deployed federated databases (using the `ooschemadump` and `ooschemaupgrade` tools), 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.

See also Chapter 19, “Object Conversion,” in the Objectivity / C++ programmer’s guide

dumpCatalog

(*administration*) Prints out a list of the files associated with the referenced federated database; primarily used in administration tools.

```
ooStatus dumpCatalog(
     FILE *outputFile = stdout,
     const ooFileNameFormat format = oocHostLocal,
     const ooBoolean printLabels = oocTrue) const;
```

Parameters

outputFile

Pointer to a transcript file in which to print out the list of files. The default is standard output.

format

Format in which to print each filename:

- Specify `oocHostLocal` (the default) to use *`host:localPath`*—for example, `object:/mnt/ed/design/up.FDB`).
- Specify `oocNative` to use full network pathnames—for example, `/net/object/usr/mnt/ed/dsgn/up.FDB`.

printLabels

Specifies whether to identify each file listed in the output. By default, each filename is labeled; specify `oocFalse` to suppress labels.

Returns

`oocSuccess` if successful; otherwise `oocError`.

Discussion

This member function lists all of the files associated with the federated database, including the system database file, the boot file, the journal directory, files used by the lock server, and all database files. (*FTO*) In a partitioned federated database, the output includes system database files and boot files of all autonomous partitions. (*DRO*) If replication is used, the output includes all database image files.

This member function performs the same function as the `oodumpcatalog` tool (see the Objectivity / DB administration book).

exist

Tests for the existence of the specified federated database; if successful, sets this object reference or handle to reference the federated database and optionally opens it.

```
ooBoolean exist(
     const char *bootFilePath,
     const ooMode openMode = oocNoOpen);
```

Parameters *bootFilePath*

Path to the boot file of the federated database or an autonomous partition.

(*FTO*) If the specified file is the boot file of an autonomous partition, that partition is the boot autonomous partition for the application.

openMode

Mode in which to open the federated database if it exists:

- Specify `oocNoOpen` (the default), to set this object reference or handle to the federated database without opening it.
- Specify `oocRead` to open the federated database for read.
- Specify `oocUpdate` to open the federated database for update.

Returns `oocTrue` if the specified federated database exists, or `oocFalse` if the federated database does not exist or if it is not accessible.

Discussion If the specified federated database exists, this object reference or handle is set to the federated database; otherwise, this object reference or handle is set to null.

If you specifically want to test for existence, you use the *openMode* parameter's default value (*oocNoOpen*). Otherwise, a return value of *oocFalse* could mean either that the federation doesn't exist, or that it does exist, but cannot be opened.

See also open

isValid

Checks whether this object reference or handle is valid—that is, whether it references the federated database.

```
ooBoolean isValid() const;
```

Returns `oocTrue` if the object reference or handle references the federated database; `oocFalse` if this object reference or handle is null.

lock

Explicitly locks the referenced federated database.

```
ooStatus lock(const ooLockMode lockMode) const;
```

Parameters *lockMode*

Type of lock to request:

- Specify `oocLockRead` to request a read lock.
- Specify `oocLockUpdate` to request an update lock.

Returns `oocSuccess` if the requested lock is obtained; otherwise `oocError`.

Discussion Objectivity / DB operations request and obtain locks implicitly as they are needed. You use this member function to obtain a lock explicitly when you want to reserve access to a federated database in advance.

Explicitly locking a federated database for read essentially allows concurrent read transactions and prevents concurrent update transactions against the same federated database. More specifically, locking a federated database for read in a read transaction:

- Prevents any other transaction from concurrently opening the federated database for update.
- Allows any other transaction to concurrently open the federated database (and its databases and containers) for read.

Explicitly locking a federated database for update in an update transaction prevents all concurrent access; an exclusive lock is placed on the federated database, which prevents any other transaction from opening it for read or update.

A federated database cannot be locked:

- For read, if another transaction has already opened it for update.
- For update, if another transaction has already locked it for read or opened it for update.

lockServerName

Gets the network name of the host running the lock server for the referenced federated database.

```
char *lockServerName() const;
```

Returns Pointer to a string containing the lock server host name.

Discussion The string is statically allocated by the member function and is overwritten with each invocation. You should make a local copy of the returned string if you intend to use it later in the application.

name

Gets the system name of the referenced federated database.

```
char *name() const;
```

Returns Pointer to a string containing the system name.

Discussion The string is statically allocated by the member function and is overwritten with each invocation. You should make a local copy of the returned string if you intend to use it later in the application.

number

(*administration*) Gets the identifier of the referenced federated database.

```
uint32 number() const;
```

Returns Integer representing the federated-database identifier.

open

Opens the specified federated database for the specified level of access and sets this object reference or handle to reference the opened federated database.

```
ooStatus open(
     const char *bootFilePath,
     const ooMode openMode = oocRead,
     ooBoolean recover = oocFalse);
```

Parameters

bootFilePath

Path to the boot file of the federated database or autonomous partition to be opened. You can omit this parameter if you set the `OO_FD_BOOT` environment variable to the path. You can specify this path with or without a host name. If you specify it as a host path, use the format *`host`*`::`*`path`*.

(*FTO*) If the specified file is the boot file of an autonomous partition, that partition is the boot autonomous partition for the application.

openMode

Intended level of access to the federated database:

- Specify `oocRead` (the default) to open the federated database for read and to designate the transaction as a read transaction.
- Specify `oocUpdate` to open the federated database for update (read and write) and to designate the transaction as an update transaction.

recover

Specifies whether to perform local automatic recovery when opening the federated database. If you specify `oocTrue`, Objectivity / DB rolls back any incomplete *local* transactions against the federated database (transactions that were started by other applications running on the same host).

Returns `oocSuccess` if successful; otherwise `oocError`.

Discussion

You must call `open` at the beginning of each transaction in an application. In the first transaction, opening the federated database helps to initialize Objectivity / DB with schema information and storage page size. In subsequent transactions, this operation has minimal performance impact because it simply verifies that the same federated database is being accessed. *Only one* federated database can be open in a process.

Any number of transactions may concurrently open the same federated database for read or update access, subject to existing locks (see the `lock` member function):

- A federated database can be opened for read but not update if another transaction has already locked it for read.
- A federated database cannot be opened in any mode if another transaction has already locked it for update.

If you intend to modify any object in a federated database, you must open the federated database for update, or the changes you make will be lost. Within a transaction, you can promote the open mode from read to update by calling open again with *`openMode`* set to `oocUpdate`. You do not need to close the federated database first. However, you may not demote the open mode from update to read; if you try, `oocSuccess` is returned, but the open mode remains unchanged.

You use the `recover` parameter to enable automatic recovery for the application. For performance reasons, you should arrange for this parameter to be set to `oocTrue` only once in an application (during the first transaction). For more information about automatic recovery, see the Objectivity / DB administration book.

See also `exist`

openMode

Gets the current level of access to the referenced federated database.

```
ooMode openMode() const;
```

Returns

One of the following constants:

- `oocNoOpen`—the federated database is closed in this transaction.
- `oocRead`—the federated database is open for read in this transaction.
- `oocUpdate`—the federated database is open for update in this transaction.

pageSize

(*administration*) Gets the storage page size of the referenced federated database.

```
uint32 pageSize() const;
```

Returns Integer representing the federated database's storage page size. This size was set when the federated database was created.

setConversion

Registers a conversion function for objects of the specified class in the referenced federated database.

```
ooStatus setConversion(
     const char *className,
     ooConvertFunction convFunction) const;
```

Parameters

className

Class of objects to be converted by the conversion function.

convFunction

Pointer to an application-defined conversion function.

Discussion A conversion function augments any kind of object conversion performed during or after schema evolution. You can register a conversion function in any application that is to trigger object conversion. When a conversion function is registered for a particular evolved class, the application invokes the function each time an object of that class is converted. A conversion function typically gets data member values from an existing unconverted object and then sets new data member values in the object's converted representation.

In a given application, you can register no more than one conversion function for each evolved persistence-capable class. Registering a second conversion function for a class replaces the previously registered function.

Returns `oocSuccess` if successful; otherwise `oocError`.

See also Chapter 19, "Object Conversion," in the Objectivity/C++ programmer's guide

tidy

(*administration*) Consolidates fragmented storage space in all the databases in the referenced federated database; used only in custom administration tools.

```
ooStatus tidy(
     FILE *outputFile = stdout,
     const char *hostName = 0,
     const char *pathName = 0) const;
```

Parameters

outputFile

Pointer to the transcript file in which to write the generated report. The default is standard output.

hostName

Name of the data server host on which to create the temporary file. Specify 0 to create the file on the current host.

pathName

Pathname of the directory on *hostName* in which to create the temporary file.

Returns

Returns a non-zero value for `ooStatus` if successful.

Discussion

This member function creates a temporary file for intermediate data during execution and therefore requires free disk space equal to the size of the largest database file in the federation. The temporary file is created in the directory that contains the federated database's system database file unless you specify *hostName* and *pathName*.

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 system data that was cached during the transaction, and commiting the transaction discards the obsolete cached data.

You must not abort the transaction after calling the `tidy` member function.

WARNING To prevent database corruption, make sure no other transactions are concurrently accessing a federated database being tidied.

This member function performs the same function as the `ootidy` tool (see the Objectivity / DB administration book).

typeN

Gets the type number of the federated-database class `ooFDObj`.

```
ooTypeNumber typeN() const;
```

Returns

Type number of the federated-database class `ooFDObj`.

typeName

Gets the name of the federated-database class ooFDObj.

```
char *typeName() const;
```

Returns The string "ooFDObj".

Discussion The returned string must be treated as read-only.

update

Opens the referenced federated database for update access.

```
ooStatus update();
```

Returns oocSuccess if successful; otherwise oocError.

Discussion This member function is equivalent to calling open(oocUpdate).

upgradeObjects

Performs object conversion throughout the referenced federated database; used only in a special-purpose upgrade application.

```
ooStatus upgradeObjects(ooBoolean purge_schema = oocFalse);
```

Parameters *purge_schema*

Specifies whether to remove schema-evolution history after objects are converted. By default, this history is not purged. **Warning:** Purging the schema may delete information that is required for distributing schema changes to deployed federated databases (see Discussion below). You should take a backup of the federated database before purging its schema.

Returns oocSuccess if successful, or oocError if no classes have been changed by schema evolution.

Discussion Certain schema-evolution operations require that you create a special-purpose upgrade application to perform object conversion. An upgrade application takes the place of on-demand or deferred conversion. You can, however, register one or more conversion functions in an upgrade application.

An upgrade application is identified by a invoking ooTrans::upgrade. You then call the upgradeObjects member function in an update transaction.

If you make schema changes in a development federated database and then distribute these to deployed federated databases (using the ooschemadump and ooschemaupgrade tools), 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.

See also Chapter 19, "Object Conversion," in the Objectivity / C++ programmer's guide

ooRefHandle(ooObj) Classes

Inheritance: **ooRef(ooObj)**

Inheritance: **ooHandle(ooObj)**

The abbreviation *ooRefHandle*(ooObj) refers to two non-persistence-capable classes:

- ooRef(ooObj), which represents an *object reference* to an Objectivity / DB object.
- ooHandle(ooObj), which represents a *handle* to an Objectivity / DB object.

These two classes are documented together because they define almost identical sets of member functions (exceptions are listed in the "Reference Summary"). These classes, along with the ooObj class, define persistence behavior for various kinds of *Objectivity/DB objects* (instances of ooObj and its derived classes).

See:

- "Reference Summary" on page 601 for an overview of member functions
- "Reference Index" on page 603 for a list of member functions

(*ODMG*) You can use either of the ODMG standard class names d_Ref<ooObj> or d_Ref<d_Object> interchangeably with ooRef(ooObj).

About Handles and Object References

An application works with an Objectivity / DB object indirectly through one or more handles or object references that are set to reference the object. An object of a particular type is usually referenced by a handle or object reference of the corresponding type—for example, a standard container is referenced by instances of *ooRefHandle*(ooContObj), a database is referenced by instances of *ooRefHandle*(ooDBObj), a basic object of an application-defined class

appClass is referenced by instances of *ooRefHandle*(*appClass*), and so on. All of the various type-specific handle and object-reference classes are derived from the *ooRefHandle*(ooObj) classes.

In general, every handle or object reference serves to:

- 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.

It is sometimes more appropriate to use a handle rather than an object reference, and vice versa; the choice is described in "Structure and Behavior" on page 597. A simple guideline is to use handles in function definitions and object references as data member types in persistence-capable class definitions.

Inheritance Hierarchy

The *ooRefHandle*(ooObj) classes are the base classes for all type-specific handle and object-reference classes, forming a pair of inheritance hierarchies that parallel the class derivation from ooObj. That is:

- Just as class ooObj is the base class for all Objectivity / DB objects, ooHandle(ooObj) is the base class for all handle classes, and ooRef(ooObj) is the base class for all object-reference classes.
- For every class *className* that derives from ooObj, a corresponding handle class ooHandle(*className*) derives from ooHandle(ooObj), and a corresponding object-reference class ooRef(*className*) derives from ooRef(ooObj).

The *ooRefHandle*(ooObj) classes are therefore the base classes for:

- Handles and object references to storage objects and autonomous partitions (instances of the ooRefHandle(ooContObj), ooRefHandle(ooDBObj), ooRefHandle(ooFDObj), and ooRefHandle(ooAPObj) classes).
- Handles and object references to persistent objects, including:
  - Instances of the ooRefHandle(appClass) classes, which are generated by the DDL processor for every application-defined persistence-capable class *appClass* derived from ooObj or ooContObj.
  - Instances of predefined classes such as *ooRefHandle*(ooMap), *ooRefHandle*(ooGeneObj), and *ooRefHandle*(ooGCContObj), which exist for every persistence-capable class defined by Objectivity / C++.

NOTE The predefined handle and reference classes for most persistence-capable classes are not documented separately; you can think of them as a kind of ooRefHandle(appClass).

Interface

The *ooRefHandle*(ooObj) base classes define three overlapping interfaces:

- The core interface for operating on Objectivity / DB objects of any type. This interface includes member functions for general Objectivity / DB operations, such as opening, locking, printing object identifiers, and so on.
- An interface for operating on persistent objects (basic object or container only). This interface extends the core interface to include an indirect member-access operator (->) and member functions for managing scope names.
- The complete interface for operating on basic objects. This interface extends the persistent-object interface to include member functions for moving, copying, and versioning operations.

Every derived handle or object-reference class acquires the member functions of the applicable interface by either inheriting them or redefining them wherever type-specific parameters or behavior are required. (Because these member functions are not virtual, they are hidden but not overridden in the derived classes that redefine them.)

Furthermore, the derived handle and object-reference classes disallow any inapplicable member functions by redefining them as private members. For example, the *ooRefHandle*(ooContObj) classes redefine the inherited copy member function as private because the copy operation is intended only for basic objects.

The majority of member functions defined by *ooRefHandle*(ooObj) are inherited by the handle and object-reference classes for persistent objects; comparatively few member functions are reimplemented or disallowed. In contrast, the ooRefHandle(ooDBObj), ooRefHandle(ooFDObj), and ooRefHandle(ooAPObj) classes disallow or reimplement a majority of the *ooRefHandle*(ooObj) member functions, and so diverge considerably from the base interface.

Usage

Handles and object references of type *ooRefHandle*(ooObj) can reference instances of ooObj or of any class derived from ooObj. Such handles and object references provide an alternative to using type-specific handles and object

references. For example, a basic object of an application-defined class `RentalFleet` can be referenced by instances of *`ooRefHandle`*`(RentalFleet)` and by instances of *`ooRefHandle`*`(ooObj)`.

Because of the interfaces of the various handle and object-reference classes, instances of the *`ooRefHandle`*`(ooObj)` classes are used in two distinct ways:

- As *persistent-object handles and object references*, which can be set to reference either basic objects or containers. This is the more common usage.
- As *general-purpose handles and object references*, which can be set to reference any type of Objectivity / DB object. This usage is fairly rare.

Persistent-Object Handles and Object References

Instances of the *`ooRefHandle`*`(ooObj)` base classes are normally used for referencing just persistent objects (basic objects and containers) rather than all types of Objectivity / DB objects. This is because:

- Most member functions defined by *`ooRefHandle`*`(ooObj)` are implemented for operating on persistent objects. Thus, a persistent-object handle of type `ooHandle(ooObj)` can perform the same persistence operations as can a handle of the type-specific class `ooHandle(`*`appClass`*`)`, when referencing an *`appClass`* object.
- Most Objectivity / C++ functions that use a *`ooRefHandle`*`(ooObj) &` parameter perform operations that are available only to persistent objects or only to basic objects, but not to the federated database, databases, or autonomous partitions.

Your application can use variables of type *`ooRefHandle`*`(ooObj)` as persistent-object handles and object references when the class of the referenced persistent object can't be known until run time.

General-Purpose Handles and Object References

Instances of the *`ooRefHandle`*`(ooObj)` classes are used as general-purpose handles and object references by certain Objectivity / C++ functions that accept or return several kinds of referenced Objectivity / DB object. These functions use such handles and object references as parameters or return values. For example, because any kind of Objectivity / DB object can be a scope object for defining scope names, the functions that set, get, and remove scope names have a parameter of type `ooHandle(ooObj) &` to specify the scope object.

NOTE Although a parameter of type *`ooRefHandle`*`(ooObj) &` syntactically accepts a handle or object reference of any type, the function itself need not operate on all types of referenced object. For example, `ooDelete` has a parameter of type `ooHandle(ooObj) &` for specifying the object to be deleted. This function does

not delete federated databases, however, so it signals an error if it receives a handle to a federated database.

It is convenient and appropriate to use a general-purpose handle or object reference when both of the following conditions are met:

- The type of the referenced object can't be known until runtime.
- The specific type isn't relevant because only core persistence operations are to be performed on the referenced object.

In all other cases, your application should avoid using variables of type *ooRefHandle*(ooObj) as general-purpose handles and object references. This is because a general-purpose handle or object reference to a database, federated database, or autonomous partition cannot perform the same persistence operations on the referenced object as can a handle or object reference of the appropriate type-specific class.

Instead, your application should:

- Use variables of type *ooRefHandle*(ooObj) only as persistent-object handles and object references.
- Reference the federated database, databases, and autonomous partitions with type-specific handles and object references.

Structure and Behavior

Regardless of type, all handles and object references provide a way to reference and operate on Objectivity / DB objects. The key difference between handles and object references is:

- Handles contain both an identifier for the referenced object and cache-related state that is associated with the Objectivity context in which it was created.
- Object references contain an identifier for the referenced object, but no bulky cache-specific state.

Therefore, only object references (but not handles) can be passed between Objectivity contexts and saved persistently.

When databases, federated databases, or autonomous partitions are referenced, handles are nearly equivalent to object references—both are essentially wrappers for a referenced object's identifier, as described in the "Structure and Behavior" sections of ooRefHandle(ooDBObj), ooRefHandle(ooFDObj), and ooRefHandle(ooAPObj).

When persistent objects are referenced, handles and object references are optimized for different purposes:

- Handles are optimized for accessing persistent objects in memory—that is, for performing multiple operations on a referenced object or repeatedly accessing the object's members.
- Object references are optimized for linking persistent objects—that is, for storing object identifiers persistently in reference attributes, in associations, or as elements of a collection.

Handles to Persistent Objects

Handles are optimized for efficient in-memory access because they can automatically obtain and manipulate pointers to referenced persistent objects. Thus, when a handle is set to reference a particular persistent object, the handle stores the object identifier for that object. The first time the persistent object is accessed through the handle, the handle is automatically *opened*—that is, the handle obtains a pointer to the object's representation in memory. This memory pointer enables the handle to access the referenced object quickly during subsequent operations performed through the handle. When the handle is *closed*, it invalidates the pointer but keeps the object identifier, so the application can reuse the handle (without resetting it) to access the same object.

Besides maintaining a pointer to the referenced persistent object, an open handle also *pins* the object's memory representation in the Objectivity / DB cache. Pinning guarantees that the persistent object is readily available in memory for as long as it is needed. Closing the handle removes its particular "pin"; when the last open handle to that persistent object is closed, the last pin is removed and the object itself is closed. Closing the last (or only) object on a buffer page permits Objectivity / DB to swap the page out of the cache as needed.

Object References to Persistent Objects

Object references are optimized for implementing persistent links because they are essentially wrappers for object identifiers. Thus, setting an object reference to a particular persistent object causes the object reference to store the object's identifier. The object reference never acquires a pointer to the persistent object in memory; instead, whenever the persistent object is accessed through the object reference, the operation is delegated to a temporary handle that provides the necessary pointer.

For convenience, an application can use an object reference (instead of a handle) to perform an operation on a referenced persistent object or to access one of the object's members. However, poor performance results when an object reference is used for multiple such operations on the same persistent object, because *each* operation causes a temporary handle to be created, used, and discarded. Performance may also be affected by swapping, because the object reference does

not pin the persistent object's memory representation in the Objectivity / DB cache.

An object reference of type `ooRef(ooObj)` is sometimes called a *standard object reference* because it contains the complete object identifier for the referenced object. An alternative for referencing a basic object under certain circumstances is the *short object reference* of type `ooShortRef(ooObj)`, which saves space by storing object identifiers in a truncated format.

Working With Persistent-Object Handles

NOTE For simplicity, this section describes how to work with handles to persistent objects. Except where noted, the same information applies to object references.

An application normally creates a persistent-object handle as a local variable on the stack, rather than allocating it on the heap. A handle should not be declared as `const`, because its internal state is changed by any operation that accesses a persistent object through it. (Object references may be declared as `const`.) An application should not explicitly define subclasses of the *`ooRefHandle`*`(ooObj)` classes; any necessary subclasses are generated automatically by the DDL processor if the application defines any subclasses of `ooObj`.

A new persistent-object handle is normally *null*—that is, it contains the value 0 instead of an object identifier. The application can then set the handle to reference a particular persistent object in any of the following ways:

- By creating a new persistent object with `operator new` of the desired class and assigning the result to the handle.
- By finding an existing persistent object with the handle's `lookupObj` member function.
- By passing the handle to a member function that sets it, such as the *`linkName`* member function of a persistent object that has a *`linkName`* association. The *`linkName`* member function finds the associated destination object and sets the specified handle to the found object.
- By assignment or initialization from another handle or object reference.

An object reference may be set in any of these ways, with the following exception—the result of `operator new` may not be assigned to an object reference.

A handle continues to reference the same persistent object until it is set to another persistent object or to null. Furthermore, multiple handles and object references can be set to the same persistent object.

An application operates on a persistent object by calling:

- Member functions of a handle that references the object. As indicated in "Reference Summary" on page 601, such member functions allow you to assign a scope name to the referenced object, open it for update, and so on.

  To call a member function of a handle, you use the direct member-access operator (`.`). For example, `objH.lookupObj` calls the `lookupObj` member function of the handle `objH`.

- Member functions of the referenced object itself.

  To call a member function of a referenced object, you use the handle's overloaded indirect member-access operator (`operator->`). For example, `objH->ooIsKindOf` calls the `ooIsKindOf` member function on the persistent object that is referenced by the handle `objH`. Only members inherited from `ooObj` can be accessed from a persistent-object handle.

Although most of a handle's member functions operate on the referenced persistent object, some functions operate on the handle itself. For example, you use:

- The assignment operator `operator=` to set a handle from another handle or from an object reference.
- The comparison operators `operator==` and `operator!=` to test whether a handle references the same persistent object as another handle or object reference.
- The member function `isNull` to test whether a handle is null. (Alternatively, you can use the overloaded `operator==` to compare a handle to 0.)

A handle preserves its reference to a persistent object across transaction boundaries, provided that the handle does not go out of scope and is not set to null as the result of an abort operation. Before reusing the handle in a new transaction, however, the application should call `isValid` to test whether the handle is still *valid*—that is, whether it still references an existing persistent object. A handle becomes invalid if it is set to null *or* if the referenced persistent object has been deleted by another process between transactions.

Opening and Closing a Persistent-Object Handle

NOTE This subsection applies only to handles, not to object references, which are in effect always closed.

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 object in memory

until the handle is closed. A closed handle, which has an object identifier instead of a pointer, can reference either an open or a closed persistent object.

The most common way to open a persistent object through a handle is to do so implicitly by using the handle's indirect member-access operator (operator->) to access a member of the referenced object. Alternatively, a referenced persistent object can be opened by explicit request—for example, by calling the handle's open or update member function. Another way to explicitly open a persistent object is by finding it 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 (for example, because another operation opened it earlier in the transaction), the accessing handle gets a pointer to the existing memory representation and adds a pin.

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 it is set to reference another persistent object, or when the transaction that opened it commits or aborts. An application can close a handle explicitly by calling the handle's close member function. Closing the last open handle to a particular persistent object unpins and closes the object.

Reference Summary

The summarized member functions are defined on both the object-reference class and the handle class. Two operators are defined on only the handle class, namely, operator* and operator ooObj*. One operator is defined on only the object-reference class, namely, operator int.

Creating a Handle or Object Reference	ooRef(ooObj) ooHandle(ooObj)
Setting the Handle or Object Reference	operator= lookupObj set_container
Comparing Handles and Object References	operator== operator!=

Accessing the Persistent Object	operator-> operator* operator ooObj* ptr (*ODMG*)
Opening, Closing, and Locking the Persistent Object	open openMode update close lock lockNoProp
Modifying the Persistent Object	delete_object (*ODMG*) update
Copying or Moving the Basic Object	copy move
Getting Information About the Objectivity/DB Object	print sprint typeN typeName
Testing the Handle or Object Reference	openMode is_null (*ODMG*) isNull isValid operator int operator ooObj*
Working With Scope Names	nameObj lookupObj getObjName unnameObj getNameObj getNameScope
Finding Objects	lookupObj containedIn getNameObj getNameScope

Versioning the Basic Object	getDefaultVers setDefaultVers getNextVers getPrevVers getVersStatus setVersStatus
ODMG Interface	operator d_Ref_Any delete_object is_null ptr

Reference Index

isValid	Tests whether this object reference or handle is valid—that is, whether it references an existing Objectivity/DB object.
lock	Explicitly locks the referenced persistent object; propagates locks along associations that have lock propagation enabled.
lockNoProp	Explicitly locks the referenced persistent object, without propagating locks to associated destination objects.
lookupObj	Finds the persistent object with the specified scope name (or the basic object matching the specified key structure) within the specified scope, and sets this object reference or handle to reference the found object.
move	Moves the referenced basic object to a different container.
nameObj	Names the referenced persistent object in the specified scope.
ooHandle(ooObj)	Default constructor that constructs a null handle.
ooHandle(ooObj)	Constructs a handle that references the same Objectivity/DB object as the specified object reference, handle, pointer, or ODMG generic reference.
ooRef(ooObj)	Default constructor that constructs a null object reference.
ooRef(ooObj)	Constructs an object reference that references the same Objectivity/DB object as the specified object reference, handle, pointer, or ODMG generic reference.
open	Explicitly opens the referenced persistent object, preparing the object for the specified level of access.
openMode	Tests whether this handle is open, and, if so, gets the current level of access to the referenced basic object.
operator->	Indirect member-access operator; accesses a member of the referenced persistent object.
operator*	Handle class only. Dereference operator; returns the persistent object referenced by this handle.
operator=	Assignment operator; sets this object reference or handle to reference the same Objectivity/DB object as the specified object reference, handle, or pointer.
operator==	Equality operator; tests whether this object reference or handle has the same value as the specified item.

operator!=	Inequality operator; tests whether this object reference or handle has a different value from the specified item.
operator d_Ref_Any	(ODMG) Conversion operator that returns an ODMG generic reference to the referenced object.
operator int	Object-reference class only. Conversion operator that tests whether this object reference is null.
operator ooObj*	Handle class only. Conversion operator that returns a C++ pointer to the Objectivity/DB object that is referenced by this handle.
print	Prints the object identifier of the referenced Objectivity/DB object.
ptr	(ODMG) Returns a C++ pointer to the referenced persistent object.
set_container	Provides this object reference or handle with container information so you can assign a short object reference to it.
setDefaultVers	Sets the referenced basic object as the default version of its genealogy.
setVersStatus	Enables or disables versioning for the referenced basic object by setting the object's versioning mode.
sprint	Returns a string containing the object identifier of the referenced Objectivity/DB object.
typeN	Gets the type number of the class of the referenced Objectivity/DB object.
typeName	Gets the name of the class of the referenced Objectivity/DB object.
unnameObj	Deletes the name in the specified scope for the referenced persistent object.
update	Opens the referenced persistent object for update access.

Constructors

ooHandle(ooObj)

Default constructor that constructs a null handle.

```
ooHandle(ooObj)();
```

ooHandle(ooObj)

Constructs a handle that references the same Objectivity / DB object as the specified object reference, handle, pointer, or ODMG generic reference.

```
1.   ooHandle(ooObj)(const ooRefHandle(ooObj) &objectRH);
2.   ooHandle(ooObj)(const ooObj *objectP);
(ODMG) 3.   ooHandle(ooObj)(const d_Ref_Any &from);
```

Parameters

objectRH

Object reference or handle to any Objectivity / DB object (basic object, container, database, federated database, or autonomous partition).

objectP

Pointer to a basic object, container, database, or autonomous partition. The pointer *must* be the result of `operator new` on `ooObj` or a derived class.

from

(*ODMG*) An ODMG generic reference to a basic object or container.

Discussion

Variants 1 and 3 allow a new handle to be constructed from an existing object reference, handle, or ODMG generic reference. If the new handle is constructed from an existing open handle, the new handle is open; in all other cases, the new handle is closed.

Variant 2, which constructs a handle from the specified pointer, has a narrower purpose—to obtain an open handle to a newly created Objectivity / DB object so you can perform persistence operations on the object and so the object can eventually be unpinned when it is no longer needed in memory.

ooRef(ooObj)

Default constructor that constructs a null object reference.

```
ooRef(ooObj)();
```

ooRef(ooObj)

Constructs an object reference that references the same Objectivity / DB object as the specified object reference, handle, pointer, or ODMG generic reference.

```
1.   ooRef(ooObj)(const ooRefHandle(ooObj) &objectRH);
2.   ooRef(ooObj)(const ooObj *objectP);
(ODMG) 3.   ooRef(ooObj)(const d_Ref_Any &from);
```

Parameters

objectRH

Object reference or handle to any Objectivity / DB object (basic object, container, database, federated database, or autonomous partition).

objectP

Pointer to a persistent object (basic object or container). The pointer may *not* be the result of `operator new`. Instead, the pointer must be the result of using either `operator ooObj*` on a handle *or* `ptr` on an object reference or handle earlier in the same transaction. If the specified pointer was originally extracted from a handle, that handle must still exist and reference the same persistent object.

from

(*ODMG*) An ODMG generic reference to a persistent object (basic object or container).

Discussion

Variants 1 and 3 allow a new object reference to be constructed from an existing object reference, handle, or ODMG generic reference.

Variant 2 has a narrower purpose, which is to allow you to resume persistence operations on a persistent object after manipulating it through a pointer. The use of this variant should be rare, however, because pointers are not normally used to manipulate persistent objects.

Operators

operator->

Indirect member-access operator; accesses a member of the referenced persistent object.

```
ooObj *operator->();
```

Returns

Pointer to the referenced basic object or container.

Discussion

The accessed persistent object is opened for read, if it is not already open.

You use `operator->` in an expression *`handle->member`*, where *`handle`* is an instance of `ooHandle(ooObj)` and *`member`* is the name of a public member defined on class `ooObj`. As for any overloading of the C++ member-access operator (->), the expression *`handle->member`* is interpreted as (*`handle`*`.operator->())->`*`member`*. That is, the overloaded `operator->` returns a pointer to the referenced object, and then the ordinary C++ `operator->` selects the specified member of that object, returning the value of that member.

If the referenced object is an instance of `ooObj`, `operator->` can access any of the object's public members. If the referenced object is an instance of a derived class, `operator->` accesses only the members that are defined in `ooObj`.

operator*

Handle class only. Dereference operator; returns the persistent object referenced by this handle.

```
ooObj &operator*();
```

Returns C++ reference to the persistent object referenced by this handle.

Discussion This operator enables you to pass a handle to a function that accepts a basic object or container by reference. This operator is analogous to the C++ `operator*` for dereferencing a pointer.

The persistent object referenced by this handle is opened for read, if it is not already open.

WARNING The returned reference is guaranteed valid only as long as the handle exists, remains open, and references the same object.

Example This examples uses `operator*` to pass a handle to `helperFunction`, which accepts a C++ reference to a persistent object.

```
void helperFunction(ooObj &anObject);

void processObject(ooHandle(ooObj) &objH) {
   …
   helperFunction(*objH);
   …
}
```

operator=

Assignment operator; sets this object reference or handle to reference the same Objectivity / DB object as the specified object reference, handle, or pointer.

```
1.   ooRefHandle(ooObj) &operator=(
       const ooRefHandle(ooObj) &objectRH);
2.   ooRefHandle(ooObj) &operator=(
       const ooShortRef(ooObj) &shortObjR);
3.   ooRefHandle(ooObj) &operator=(const ooObj *objectP);
```

Parameters

objectRH

Object reference or handle to an Objectivity / DB object (basic object, container, database, federated database, or autonomous partition).

shortObjR

Short object reference to a basic object that resides in the same container as the object that is referenced by this object reference or handle. (If this object reference or handle is null, you can set it to a container with the `set_container` member function.)

A short object reference specifies just the lower half of an object identifier (corresponding to the object's logical page and slot numbers). The upper half of the object identifier (corresponding to the database and container) must be supplied by this object reference or handle.

objectP

0, or a nonnull pointer to one of several types of Objectivity / DB object:

- If you are assigning to a handle, the pointer may reference a basic object, container, database, or autonomous partition. The specified pointer *must* be the result of `operator new` on `ooObj` or a derived class.
- If you are assigning to an object reference, the pointer may reference a persistent object (basic object or container). The specified pointer may *not* be the result of `operator new`. Instead, the pointer must be the result of using either `operator ooObj*` on a handle *or* `ptr` on an object reference or handle earlier in the same transaction. If the specified pointer was originally extracted from a handle, that handle must still exist and reference the same persistent object.

Returns

This object reference or handle, after it has been set to reference the specified object.

Discussion

Variants 1 and 2 allow you to use the specified object reference or handle to produce another object reference or handle to the same object. If you are assigning to a handle from an open handle, the returned handle is open; in all other cases, the returned handle is closed.

Variant 3 allows you to set this object reference or handle to null. Otherwise, assignment-from-pointer has two specific purposes, depending on whether you are assigning to a handle or to an object reference:

- Pointer-to-handle assignment enables you to obtain an open handle to a newly created Objectivity / DB object, so you can perform persistence operations on it, and so the object can be unpinned when it is no longer needed in memory.
- Pointer-to-object-reference assignment enables you to resume persistence operations on a persistent object after manipulating it through a pointer. This

usage of variant 3 is rare, because pointers are not normally used to manipulate persistent objects.

operator==

Equality operator; tests whether this object reference or handle has the same value as the specified item.

```
1.  ooBoolean operator==(
      const ooRefHandle(ooObj) &compare) const;
2.  ooBoolean operator==(
      const ooObj *compare) const;
3.  ooBoolean operator==(
      const ooShortRef(ooObj) &compare) const;
4.  ooBoolean operator==(int zero) const;
```

Parameters

compare

Any of the following:

- Object reference or handle to an Objectivity / DB object (basic object, container, database, federated database, or autonomous partition).
- Pointer to a basic object or container.
- Short object reference to a basic object. A short object reference must refer to a basic object that resides in the same container as the object referenced by this object reference or handle.

zero

Literal 0. This value allows you to use `operator==` as an alternative for `isNull`.

Returns

Variants 1, 2, and 3 return `oocTrue` if this object reference or handle contains the same identifier as *compare*; otherwise `oocFalse`.
Variant 4 returns `oocTrue` if this object reference or handle is null; otherwise `oocFalse`.

Discussion

Variants 1, 2, and 3 test whether this object reference or handle references the same object as *compare*. Variant 4 tests whether this object reference or handle is null.

See also

`isNull` (as an alternative to variant 4)

operator!=

Inequality operator; tests whether this object reference or handle has a different value from the specified item.

```
1.  ooBoolean operator!=(
      const ooRefHandle(ooObj) &compare) const;
2.  ooBoolean operator!=(
      const ooObj *compare) const;
3.  ooBoolean operator!=(
      const ooShortRef(ooObj) &compare) const;
4.  ooBoolean operator!=(int zero) const;
```

Parameters

compare

Any of the following:

- Object reference or handle to an Objectivity / DB object (basic object, container, database, federated database, or autonomous partition).
- Pointer to a basic object or container.
- Short object reference to a basic object. A short object reference must refer to a basic object that resides in the same container as the object referenced by this object reference or handle.

zero

Literal 0.

Returns

Variants 1, 2, and 3 return `oocTrue` if this object reference or handle contains a different identifier than *compare*; otherwise `oocFalse`.
Variant 4 returns `oocTrue` if this object reference or handle is nonnull; otherwise `oocFalse`.

Discussion

Variants 1, 2, and 3 test whether this object reference or handle references a different object than *compare*. Variant 4 tests whether this object reference or handle is nonnull.

operator d_Ref_Any

(*ODMG*) Conversion operator that returns an ODMG generic reference to the referenced object.

```
operator d_Ref_Any() const;
```

operator int

Object-reference class only. Conversion operator that tests whether this object reference is null.

```
operator int() const;
```

Returns 0 if this object reference is null; otherwise, returns a nonzero integer.

Discussion This conversion operator enables you to use an object reference as the conditional expression in an `if` or `while` statement to test whether the object reference is null.

Example
```
ooRef(ooObj) objectR;
... // Set objectR to some object
if (objectR) {
... //Do something interesting if initialization was successful
}
```

See also `operator ooObj*`

operator ooObj*

Handle class only. Conversion operator that returns a C++ pointer to the Objectivity / DB object that is referenced by this handle.

```
operator ooObj*();
```

Returns The return value depends on what this handle references—specifically:

- A null pointer, if this handle is null.
- A nonnull pointer, if this handle references a database, federated database, or autonomous partition. This pointer should be used only in expressions that test for null.
- Otherwise, a pointer to the persistent object referenced by this handle.

Discussion This conversion operator enables you to:

- Pass a handle to a function that accepts a pointer to a persistent object.
- Assign a handle to an `ooObj*` variable (for example, to pass to the overloaded `operator delete`).
- Use a handle as the conditional expression in an `if` or `while` statement to test whether the handle is null.

If this handle references a persistent object, the object is opened for read (unless it is already open).

WARNING The returned pointer is guaranteed valid only as long as this handle exists, remains open, and references the same object.

An application generally relies on handles to provide memory management for persistent objects, and avoids the explicit use of pointers to such objects. On occasion, explicit use of pointers is required for performance reasons or for compatibility with functions that are indifferent to persistence (for example, in legacy code or a third-party library). However, you should be careful when manipulating a pointer from a handle because the validity of the pointer depends on the state of the handle. You should not use the returned pointer in other persistence operations (for example, do not pass it to any Objectivity / C++ member function other than the overloaded `operator delete`).

Example The handle `objectH` is used as a conditional expression which evaluates to 0 if the handle is null.

```
ooHandle(ooObj) objectH;
... // Set objectH to some object
if (objectH) {
... // Do something interesting if initialization was successful
}
```

See also `operator int`
`ptr`

Member Functions

close

Explicitly closes this handle.

```
ooStatus close() const;
```

Returns `oocSuccess` if successful; otherwise `oocError`.

Discussion This member function is redundant for object references, which are, in effect, always closed. Therefore, you should use this member function only on handles.

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.

You can use the `close` member function to close a handle explicitly. This 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.

A closed handle retains the object identifier of the Objectivity / DB object to which it refers, so you can reopen it without reinitializing. Note, however, that a retained object identifier can become invalid between transactions (for example, because a concurrent process has deleted the corresponding object), and opening a handle with an invalid object identifier signals an error.

Closing a handle to a persistent object invalidates the pointer to the object's representation in the Objectivity / DB cache. Closing the last open handle to a particular persistent object unpins and closes that object; closing the last open object on a buffer page permits Objectivity / DB to swap the page out of the cache as needed.

containedIn

Finds the container that contains the referenced basic object.

```
1.  ooHandle(ooContObj) containedIn() const;
2.  ooRef(ooContObj) &containedIn(
      ooRef(ooContObj) &container) const;
3.  ooHandle(ooContObj) &containedIn(
      ooHandle(ooContObj) &container) const;
```

Parameters

container

Object reference or handle to set to the found container.

Returns

Object reference or handle to the found container.

Discussion

When called without a *container* parameter, `containedIn` allocates a new container handle and returns it. Otherwise, `containedIn` returns the object reference or handle that is passed to it.

copy

Creates a copy of the referenced basic object, clustering the new copy near the specified object.

```
1.  ooHandle(ooObj) copy(
      const ooHandle(ooObj) &near) const;
2.  ooRef(ooObj) &copy(
      const ooHandle(ooObj) &near,
      ooRef(ooObj) &newCopy) const;
```

```
3.  ooHandle(ooObj) &copy(
      const ooHandle(ooObj) &near,
      ooHandle(ooObj) &newCopy) const;
```

Parameters

near

Handle to the object with which to cluster the new copy. *near* may be a handle to a database, a container, or a basic object:

- If *near* is a database handle, the new copy is stored in the default container of that database.
- If *near* is a container handle, the new copy is stored in that container.
- If *near* is a basic object handle, the new copy is stored in the same container as the referenced basic object. If possible, the copy will be put on the same page as the referenced basic object or on a nearby page.

newCopy

Object reference or handle to set to the new copy.

Returns

Object reference or handle to the new copy.

Discussion

When called without a *newCopy* parameter, `copy` allocates a new handle and returns it. Otherwise, `copy` returns the object reference or handle that is passed to it.

You can copy basic objects only, not containers. The application must be able to lock the container of the original object for read and the container of the new copy for update.

delete_object

(*ODMG*) Deletes the referenced persistent object.

```
void delete_object();
```

Discussion

This member function is equivalent to `ooObj::operator delete`.

getDefaultVers

Finds the default version of the referenced basic object.

```
ooStatus getDefaultVers(
     ooRefHandle(ooObj) &default) const;
```

Parameters

default

Object reference or handle to set to the default version.

Returns

`oocSuccess` if successful; otherwise `oocError`.

Discussion This member function identifies the genealogy of the referenced basic object, and finds the default version in the genealogy, provided that the referenced object was created *after* the default version was set. (Versions created before a default version is set cannot know about the default version.)

If a default version for the genealogy is found, `getDefaultVers` sets the specified object reference or handle to reference it. If no default version exists, or if the default version was specified after the referenced version was created, the specified object reference or handle is set to null.

See also `setDefaultVers`

getNameObj

Initializes an object iterator to find all objects named in the scope of the referenced persistent object.

```
ooStatus getNameObj(
     ooItr(ooObj) &objI) const;
```

Parameters *objI*

Object iterator for finding the named objects.

Returns `oocSuccess` if successful; otherwise `oocError`.

Discussion The application must be able to obtain a read lock on the hashed container used by the scope object.

getNameScope

Initializes an object iterator to find all scope objects in the federated database that define a scope name for the referenced persistent object.

```
ooStatus getNameScope(
     ooItr(ooObj) &objI) const;
```

Parameters *objI*

Object iterator for finding the scope objects.

Returns `oocSuccess` if successful; otherwise `oocError`.

Discussion The application must be able to obtain read locks on all the hashed containers used by the scope objects.

getNextVers

Initializes an object iterator to find the next version(s) of the referenced basic object.

```
ooStatus getNextVers(
     ooItr(ooObj) &nextVers,
     const ooMode openMode = oocNoOpen) const;
```

Parameters

nextVers

Object iterator for finding the next version(s).

openMode

Intended level of access to the versions found by the iterator's `next` member function:

- `oocNoOpen` (the default) causes `next` to set the iterator to the next version without opening it.
- `oocRead` causes `next` to open the next version for read.
- `oocUpdate` causes `next` to open the next version for update.

  Warning: If versioning is enabled for one or more found objects, specifying `oocUpdate` means that `next` will create a new version of each such object.

Returns

`oocSuccess` if successful; otherwise `oocError`.

Discussion

This member function initializes the specified iterator to find the next version (if linear versioning is used) or versions (if branch versioning is used) of the referenced object.

If no next version(s) exist, the iterator is set to null, so that invoking the iterator's `next` member function will return `oocFalse`.

See also

`getPrevVers`

getObjName

Gets the name defined in the specified scope for the referenced persistent object.

```
char *getObjName(
     const ooHandle(ooObj) &scope) const;
```

Parameters

scope

Handle to the scope object that defines the name scope to be searched. The scope object can be the federated database, a database, a persistent container, a persistent basic object, or an autonomous partition.

Returns Pointer to a string containing the scope name. If the referenced object does not have a scope name in the specified scope, the returned pointer is null.

Discussion The string is statically allocated by the member function and overwritten with each invocation. You should make a local copy of the returned string if you intend to use it later in the application.

The application must be able to get a read lock on the hashed container used by the scope object.

See also `lookupObj`

getPrevVers

Finds the previous version of the referenced basic object.

```
ooStatus getPrevVers(
     ooRefHandle(ooObj) &previous) const;
```

Parameters *previous*

Object reference or handle to set to the previous version.

Returns `oocSuccess` if successful; otherwise `oocError`.

Discussion If a previous version is found, `getPrevVers` sets the specified object reference or handle to reference it. Otherwise, the specified object reference or handle is set to null.

See also `getNextVers`

getVersStatus

Gets the current versioning mode of the referenced basic object.

```
ooVersMode getVersStatus() const;
```

Returns One of the following constants:

- `oocNoVers`—the versioning behavior of the object is disabled.
- `oocLinearVers`—linear versioning is enabled for the object.
- `oocBranchVers`—branch versioning is enabled for the object.

See also `setVersStatus`
`ooVersMode` global type

is_null

(*ODMG*) Tests whether this object reference or handle is null.

```
int is_null();
```

Returns Non-zero integer (true) if this object reference or handle is null; otherwise 0 (false).

isNull

Tests whether this object reference or handle is null.

```
ooBoolean isNull() const;
```

Returns `oocTrue` if this object reference or handle is null; otherwise `oocFalse`.

Discussion You can use this member function as an alternative to comparing this handle or object reference to 0 with `operator==`.

See also `operator==`

isValid

Tests whether this object reference or handle is valid—that is, whether it references an existing Objectivity / DB object.

```
ooBoolean isValid() const;
```

Returns `oocTrue` if this object reference or handle references an existing Objectivity / DB object; `oocFalse` if this object reference or handle is null or has a stale object identifier, or if the application cannot obtain a read lock on the container and database to be checked.

Discussion You can use `isValid` to determine whether it is safe to use an object reference or handle that was set in a previous transaction. Such an object reference or handle still retains its reference to an Objectivity / DB object; however, between transactions, that reference may have become invalid (for example, because another process has deleted the referenced object).

NOTE `isValid` checks only for the existence of a referenced object, but does *not* check whether the class of the referenced object corresponds to the class of the object reference or handle. An object reference or handle can have a valid but incorrect object identifier after another process has deleted the originally referenced object and created another object in the same location, where it is addressible by the same object identifier. Therefore, if `isValid` returns `oocTrue`, you can verify the

referenced object's class (for example, using the typeN member function), but you must write application-specific code to verify the identity of the referenced object.

If your purpose is simply to test whether an object reference or handle has been initialized, it is more efficient to use isNull, which performs its test entirely in memory without having to access files on disk.

lock

Explicitly locks the referenced persistent object; propagates locks along associations that have lock propagation enabled.

```
ooStatus lock(const ooLockMode lockMode) const;
```

Parameters

lockMode

Type of lock to request:

- Specify oocLockRead to request a read lock.
- Specify oocLockUpdate to request an update lock.

Returns

oocSuccess if all requested locks are obtained; otherwise oocError.

Discussion

Objectivity / DB operations request and obtain locks implicitly as they are needed. You use this member function to obtain a lock explicitly when:

- You want to reserve access to an object in advance—for example, before starting a complex operation.
- You want to lock an entire composite object (a group of associated objects whose associations have lock propagation enabled). When you explicitly lock an object with such associations, the associated destination objects are locked as well. Lock propagation is enabled by behavior specifiers in an association's definition.

Because containers are the fundamental unit of locking within Objectivity / DB, locking a basic object causes its container to be locked. This effectively locks all the basic objects in the same container.

Whenever a lock is requested on a container, Objectivity / DB applies the transaction's concurrent access policy to determine whether the requested lock is compatible with other existing locks. An error is signaled if a requested lock cannot be obtained.

See also

lockNoProp
ooLockMode global type

lockNoProp

Explicitly locks the referenced persistent object, without propagating locks to associated destination objects.

```
ooStatus lockNoProp(const ooLockMode lockMode) const;
```

Parameters

lockMode

Type of lock to request:

- `oocLockRead` requests a read lock.
- `oocLockUpdate` requests an update lock.

Returns

`oocSuccess` if the requested lock is obtained; otherwise `oocError`.

Discussion

Objectivity / DB operations request and obtain locks implicitly as they are needed. You use this member function to obtain a lock explicitly when you want to reserve access to an object in advance, but you do *not* want to lock any associated destination objects, even along associations that have lock propagation enabled.

Because containers are the fundamental unit of locking within Objectivity / DB, locking a basic object causes its container to be locked. This effectively locks all the basic objects in the same container.

Whenever a lock is requested on a container, Objectivity / DB applies the transaction's concurrent access policy to determine whether the requested lock is compatible with other existing locks. An error is signaled if a requested lock cannot be obtained.

See also

`lock`
`ooLockMode` global type

lookupObj

Finds the persistent object with the specified scope name (or the basic object matching the specified key structure) within the specified scope, and sets this object reference or handle to reference the found object.

```
1.   ooStatus lookupObj (
       const ooHandle(ooObj) &scope,
       const char *scopeName,
       const ooMode openMode = oocRead);
2.   ooStatus lookupObj (
       const ooHandle(ooObj) &scope,
       const ooKey &keyStruct,
       const ooMode openMode = oocRead);
```

Parameters *scope*

Handle to an object that defines the scope of the lookup:

- (Variant 1) When the lookup is by scope name, *scope* specifies the scope object that defines the name scope to search. *scope* can reference the federated database, a database, a persistent container, a persistent basic object, or an autonomous partition.
- (Variant 2) When the lookup is by key structure, *scope* specifies the container to search. *scope* can reference the container itself or the database whose default container is to be searched.

scopeName

Scope name to look up in the scope specified by *scope*.

openMode

Intended level of access to the found object:

- Specify `oocRead` (the default) to open the object for read.
- Specify `oocUpdate` to open the object for update.
- (Variant 1 only.) Specify `oocNoOpen` to set this object reference or handle to the object without opening it. `oocNoOpen` is valid only for scope-name lookup, because scope-named objects can be found without being opened, whereas keyed objects must be opened during the search.

keyStruct

Key structure specifying the key field and key field value to match.

Returns `oocSuccess` if an object is found; otherwise `oocError`.

Discussion The application must be able to obtain a read lock on the hashed container used by the scope object.

See also `getObjName`

move

Moves the referenced basic object to a different container.

```
ooStatus move(const ooHandle(ooObj) &target);
```

Parameters *target*

Handle of the object with which to cluster the basic object. *target* may be a handle to a database, a container, or a persistent basic object:

- If *target* is a database handle, the basic object is moved to the default container of that database.

- If *target* is a container handle, the basic object is moved to that container.
- If *target* is a basic object handle, the moved basic object is put into the same container as the specified target object. If possible, the moved object will be put on the same page as the target object or on a nearby page.

Returns `oocSuccess` if successful; otherwise `oocError`.

Discussion You can move basic objects only, not containers. The application must be able to obtain an update lock on the object's container and the container to which the object will be moved.

WARNING This member function changes the object identifier of the moved object, and all references containing the old identifier become invalid. When you move an object you should, within the same transaction, update references to the object within all relevant attributes, persistent collections, name scopes, indexes, and unidirectional associations. (Objectivity/DB automatically maintains referential integrity for bidirectional associations). See "Moving a Basic Object" in the Objectivity/C++ programmer's guide for further information.

nameObj

Names the referenced persistent object in the specified scope.

```
ooStatus nameObj(
     const ooHandle(ooObj) &scope,
     const char *name) const;
```

Parameters *scope*

Handle to the scope object that defines the name scope for *name*. The scope object can be the federated database, a database, a persistent container, a persistent basic object, or an autonomous partition. If *scope* references a persistent container, that container must be hashed. If *scope* references a persistent basic object, that object must be in a hashed container.

name

Scope name to assign. This name:

- Must be a null-terminated string that can contain any non-null character.
- Must be unique within the name scope defined by *scope*.
- May contain up to 487 characters.

Returns `oocSuccess` if successful; otherwise `oocError`.

Discussion

You may give an object only a single name within a given scope. If the object already has a scope name in the specified scope, you must remove that name with `unnameObj` before you can assign a new name.

The application must be able to obtain an update lock on the hashed container used by the scope object.

See also

`lookupObj`

open

Explicitly opens the referenced persistent object, preparing the object for the specified level of access.

```
ooStatus open(const ooMode openMode = oocRead);
```

Parameters

openMode

Intended level of access to the opened object:

- Specify `oocRead` (the default) to open the object for read. This implicitly requests a read lock on the relevant container.
- Specify `oocUpdate` to open the object for update (read and write). This implicitly requests an update lock on the relevant container.

Returns

`oocSuccess` if successful; otherwise `oocError`.

Discussion

Opening a basic object makes it available to your application by:

- Implicitly locking the basic object's container for read or update, as specified by *openMode*.
- Obtaining a representation of the basic object in memory, either by fetching the page(s) containing the object from the database or by reusing an existing memory representation that is guaranteed current.
- *(If* `open` *is called on a handle)* Providing this handle with a memory pointer to the opened object.

Opening a container makes it available to an application by:

- Implicitly locking the container for read or update, as specified by *openMode*.
- Obtaining a representation of the container in memory, as for basic objects. The representation for a container includes its page map and any persistent data (if the container is an instance of an application-defined container class). Opening a container does not open any of the basic objects in it.

Opening an object for update additionally marks it as modified, causing it to be written to the database when the transaction commits, whether or not the object was actually modified. You must be in an update transaction to open an object

for update. If necessary, you can promote a read transaction to an update transaction by promoting the open mode of the federated database.

The `open` operation fails if the relevant container cannot be locked—for example, due to a lock conflict. Objectivity/DB applies the transaction's concurrent access policy to determine whether the requested lock is compatible with other existing locks on the container. Once a lock is obtained, it is kept until the transaction either commits or aborts.

You rarely need to explicitly open an object for read because accessing the object through an object reference or handle accomplishes this implicitly (see `operator->`). You open an 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.

If the referenced object is a basic object for which versioning is enabled (see `setVersStatus`), opening it for update causes a new version to be created, and sets this object reference or handle to reference the new version.

See also `update`

openMode

Tests whether this handle is open, and, if so, gets the current level of access to the referenced basic object.

```
ooMode openMode() const;
```

Returns One of the following constants:

- `oocNoOpen`—this handle is not open in this transaction (although the referenced basic object may be open through another handle).
- `oocRead`—this handle is open, and the referenced basic object is open for read in this transaction.
- `oocUpdate`—this handle is open, and the referenced basic object is open for update in this transaction.

print

Prints the object identifier of the referenced Objectivity/DB object.

```
void print(FILE *outputFile = stdout) const;
```

Parameters *outputFile*

Pointer to the file in which to print the object identifier. The default is standard output.

Discussion The object identifier for a basic object or container is printed in #*D*-*C*-*P*-*S* format, which identifies the database (*D*), container (*C*), logical page number (*P*), and logical slot number (*S*) of the object—for example, #2-3-3-12. (For a container, the page and slot numbers refer to the location of the container object itself.) The object identifier for a database, autonomous partition, or federated database is printed as *I*-0-0-0, where *I* is the integer identifier of the object—for example, #2-0-0-0.

ptr

(*ODMG*) Returns a C++ pointer to the referenced persistent object.

```
ooObj *ptr();
```

Returns Pointer to the referenced basic object or container.

Discussion You use this member function to obtain a pointer to a basic object or container—for example, to pass to a function that accepts a pointer instead of a handle or object reference.

If `ptr` is called on an object reference, the referenced persistent object is opened for update. If `ptr` is called on a handle, the referenced persistent object is opened for read.

Warning: The returned pointer is guaranteed valid for only a limited time:

- If `ptr` is called on an object reference, the returned pointer is valid and the persistent object is pinned in memory until the end of the transaction.
- If `ptr` is called on a handle, the returned pointer is valid only as long as the handle exists, remains open, and references the same persistent object (equivalent to `operator ooObj*`).

An application generally relies on handles to provide memory management for persistent objects, and avoids the explicit use of pointers to such objects. On occasion, explicit use of pointers is required for performance reasons or for compatibility with functions that are indifferent to persistence (for example, in legacy code or a third-party library). However, you should be careful when manipulating persistent objects through pointers:

- Pointers extracted from handles become invalid if the handles change or go out of scope.

- Pointers extracted from object references can cause the Objectivity / DB cache to run out of memory if too many objects are pinned until the end of the transaction.

You should not use the returned pointer in other persistence operations (for example, do not pass it to any Objectivity / C++ member function other than the overloaded `operator delete`).

set_container

Provides this object reference or handle with container information so you can assign a short object reference to it.

```
ooStatus set_container(
     const ooRefHandle(ooObj) &object);
```

Parameters

object

Object reference or handle to a basic object or container.

Returns

`oocSuccess` if successful; otherwise `oocError`.

Discussion

You use this member function to set a null object reference or handle before assigning a short object reference to it. A short object reference specifies just the lower half of an object identifier (corresponding to the object's logical page and slot numbers); `set_container` supplies the upper half of the object identifier (corresponding to the database and container), and sets the page and slot components to null. That is:

- If you specify a container, this object reference or handle is set to the upper half of the object identifier for the container.
- If you specify a basic object, this object reference or handle is set to the upper half of the object identifier for the object.

WARNING After `set_container` is invoked, this object reference or handle is not valid for referencing either a basic object or a container, because it contains only partial identifier information.

setDefaultVers

Sets the referenced basic object as the default version of its genealogy.

```
ooStatus setDefaultVers() const;
```

Returns

`oocSuccess` if successful; otherwise `oocError`.

Discussion

If the referenced basic object does not already belong to a genealogy, this member function implicitly creates one (an instance of `ooGeneObj`) and adds the referenced object to it. All subsequently created versions of the same object are automatically added to this genealogy.

If you want all versions of an object to belong to a genealogy, you should set the object to be the initial default version as soon as you enable versioning for it. You can use this member function to reset the default version of a genealogy at any time.

See also `getDefaultVers`

setVersStatus

Enables or disables versioning for the referenced basic object by setting the object's versioning mode.

```
ooStatus setVersStatus(const ooVersMode versMode) const;
```

Parameters *versMode*

Versioning behavior to set for the referenced object:

- Specify `oocNoVers` to disable versioning for the object.
- 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.

Returns `oocSuccess` if successful; otherwise `oocError`.

Discussion

When versioning is enabled for a basic object, a new version of the object is created each time the object is opened for update. By default, the new version is a bit-wise copy of the opened object. An application can customize the copy operation for versioning instances of a basic-object class by overriding the `ooNewVersInit` member function in that class.

See also `getVersStatus`
`ooObj::ooNewVersInit`
`ooVersMode` global type

sprint

Returns a string containing the object identifier of the referenced Objectivity / DB object.

```
char *sprint(char *buffer = 0) const;
```

Parameters *buffer*

String in which to return the object identifier. If you omit this parameter, `sprint` statically allocates a new string.

Returns String representing the object identifier.

Discussion The object identifier for a basic object or container is printed in #*D*-*C*-*P*-*S* format, which identifies the database (*D*), container (*C*), logical page number (*P*), and logical slot number (*S*) of the object—for example, #2-3-3-12. (For a container, the page and slot numbers refer to the location of the container object itself.) The object identifier for a database, autonomous partition, or federated database is printed as *I*-0-0-0, where *I* is the integer identifier of the object—for example, #2-0-0-0.

Each successive invocation of this member function without the parameter overwrites the statically allocated string. You should make a local copy of the returned string if you intend to use it later in the application.

typeN

Gets the type number of the class of the referenced Objectivity / DB object.

```
ooTypeNumber typeN() const;
```

Returns Type number for the referenced basic object, container, database, federated database, or autonomous partition.

Discussion Every class in the `ooObj` inheritance hierarchy has a uniqe type number that identifies the class within a federated-database schema. This member function gets the type number of the referenced object's class.

See also `ooTypeN` global macro
`ooTypeNumber` global type
`ooObj::ooGetTypeN`

typeName

Gets the name of the class of the referenced Objectivity / DB object.

```
char *typeName() const;
```

Returns String containing the class name of the referenced basic object, container, database, federated database, or autonomous partition.

WARNING Do not modify the returned string in any manner. Doing so may result in unexpected program errors. This string is used internally by Objectivity/DB.

See also `ooTypeN` global macro
`ooObj::ooGetTypeName`

unnameObj

Deletes the name in the specified scope for the referenced persistent object.

```
ooStatus unnameObj(
     const ooHandle(ooObj) &scope,
     const char *name = 0) const;
```

Parameters *scope*

Handle to the scope object that defines the scope name to be deleted. The scope object can be the federated database, a database, a persistent container, a persistent basic object, or an autonomous partition.

name

This parameter is ignored.

Returns `oocSuccess` if successful; otherwise `oocError`.

Discussion The application must be able to obtain an update lock on the persistent object, the scope object, and the hashed container used by the scope object.

See also `lookupObj`, `nameObj`

update

Opens the referenced persistent object for update access.

```
ooStatus update();
```

Returns `oocSuccess` if successful; otherwise `oocError`.

Discussion This member function is equivalent to calling `open(oocUpdate)`.

ooShortRef(*appClass*) Class

Inheritance: **ooShortRef(ooObj)->ooShortRef(*appClass*)**

The non-persistence-capable class ooShortRef(*appClass*) represents a *short object reference* to an instance of the application-defined basic-object class *appClass*.

See:

- "Reference Index" on page 633 for a list of member functions

To use the ooShortRef(*appClass*) class, you must include and compile with files generated by the DDL processor, as described in "Obtaining Generated Class Definitions" on page 632.

About Short Object References

When an application defines a basic-object class *appClass* and adds it to the federated-database schema, the DDL processor generates the corresponding short-object-reference class ooShortRef(*appClass*). Short object references enable you to save space when storing persistent references to basic objects in attributes and associations. Short object references are alternatives to standard object references (instances of ooRef(appClass)); both kinds of object reference store the object identifier of a referenced basic object, with the following difference:

- A standard object reference stores all components of the object identifier, including the database, container, logical page, and logical slot numbers.
- A short object reference truncates the object identifier, storing just its lower half (the logical page and slot numbers).

Short object references occupy about half the space of standard object references, so they can be useful when disk usage is a concern for a federated database that must maintain a large number of references to basic objects. However, a

truncated object identifier provides only enough information to locate an object within its container. Consequently, a short object reference must used as an attribute or in an association of a persistent object, which determines the necessary container and database information.

You can use a short object reference of type `ooShortRef(ooObj)` to reference a basic object of any type. Normally, however, a basic object of a particular type is referenced by a short object reference of the corresponding type—that is, a basic object of an application-defined class *`appClass`* is referenced by instances of `ooShortRef(appClass)`.

Obtaining Generated Class Definitions

To use the `ooShortRef(`*`appClass`*`)` class, you must include either the primary header file or the references header file generated by the DDL processor for *`appClass`*. Thus, if *`appClass`* is defined in the DDL file *`classDefFile`*`.ddl`, you must include one of the following files:

- The primary header file *`classDefFile`*`.h`
- The references header file *`classDefFile`*`_ref.h`

Furthermore, you must compile the method implementation file *`classDefFile`*`_ddl.cxx` with your application code files.

For more information about DDL-generated files and how to use them, see the Objectivity/C++ Data Definition Language book.

When *appClass* is a Template Class

When *`appClass`* is a persistence-capable template class with multiple parameters, the name of the generated short-object-reference class contains the symbol `OO_COMMA` to separate the template parameters. For example, for a persistence-capable template class `Example<Float, Node>`, the generated class is `ooShortRef(Example<Float OO_COMMA Node>)`. This is because the macro syntax of the generated class name interprets embedded commas as separators between the as macro parameters instead of as separators between the template parameters.

Working With Short Object References

An application normally creates a short object reference implicitly as a data member or a short inline association of a persistent object. An application should not explicitly define subclasses of the `ooShortRef(`*`appClass`*`)` classes; any necessary subclasses are generated automatically by the DDL processor if the application defines any subclasses of *`appClass`*.

A new short object reference is normally *null*—that is, it contains the value 0 instead of an object identifier. The application can then set the short object reference a particular basic object in any 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.

Unlike standard object references, short object references cannot be used as smart pointers. That is, you cannot use a short object reference to operate on the referenced basic object or access its members. Rather, the member functions inherited from the `ooShortRef(ooObj)` base class are limited to operations that apply to short object references themselves, such as comparing, testing, and printing.

You can assign a short object reference to a standard object reference or handle. The upper half of the object identifier is typically supplied by calling `set_container` on the standard object reference or handle.

Reference Index

The following table summarizes just the member functions that are redefined by *`ooRefHandle`*`(`*`appClass`*`)` to provide type-specific parameters. For descriptions of inherited member functions, see the `ooShortRef(ooObj)` classes (page 637).

`ooShortRef(appClass)`	Default constructor that constructs a null short object reference.
`ooShortRef(appClass)`	Constructs a new short object reference that references the same basic object as the specified short object reference, standard object reference, handle, or pointer.
`operator=`	Assignment operator; sets this short object reference to the same basic object as the specified short object reference, standard object reference, handle, or pointer.

Constructors and Destructors

ooShortRef(*appClass*)

Default constructor that constructs a null short object reference.

```
ooShortRef(appClass)();
```

ooShortRef(*appClass*)

Constructs a new short object reference that references the same basic object as the specified short object reference, standard object reference, handle, or pointer.

```
1.  ooShortRef(appClass)(
      const ooShortRef(appClass) &object);
2.  ooShortRef(appClass)(
      const ooRefHandle(appClass) &object);
3.  ooShortRef(appClass)(
      const appClass *objectP);
```

Parameters

object

Short object reference, standard object reference, or handle to an instance of *appClass*.

objectP

Pointer to an instance of *appClass*. The pointer may *not* be the result of `operator new` on *appClass*. Instead, the pointer must be the result of using `operator appClass*` on a handle *or* `ptr` on an object reference or handle earlier in the same transaction. If the specified pointer was originally extracted from a handle, that handle must still exist and reference the same object.

Discussion

If you use Variant 2 to construct a short object reference from a standard object reference or handle, the database and container portion of the object identifier are discarded.

Variant 3 allows you to resume persistence operations on a basic object after manipulating it through a pointer. The use of this variant should be rare, however, because pointers are not normally used to manipulate basic objects.

Operators

operator=

Assignment operator; sets this short object reference to the same basic object as the specified short object reference, standard object reference, handle, or pointer.

```
1.  ooShortRef(appClass) &operator=(
      const ooShortRef(appClass) &object);
2.  ooShortRef(appClass) &operator=(
      const ooReHandle(appClass) &object);
```

```
3.  ooShortRef(appClass) &operator=(
      const appClass *objectP);
```

Parameters

object

Short object reference, standard object reference, or handle to an instance of *appClass*.

objectP

0, or a nonnull pointer to an instance of *appClass*. The pointer may *not* be the result of `operator new` on *appClass*. Instead, the pointer must be the result of using `operator appClass*` on a handle *or* `ptr` on an object reference or handle earlier in the same transaction. If the specified pointer was originally extracted from a handle, that handle must still exist and reference the same object.

Returns

This short object reference.

Discussion

If you use Variant 2 to assign a standard object reference or handle to a short object reference, the database and container portion of the object identifier are discarded.

Variant 3 allows you to set this short object reference to null. Otherwise, assignment-from-pointer enables you to resume persistence operations on a basic object after manipulating it through a pointer. The use of this variant should be rare, however, because pointers are not normally used to manipulate basic objects.

ooShortRef(ooObj) Class

Inheritance: **ooShortRef(ooObj)**

The non-persistence-capable class `ooShortRef(ooObj)` is the base class for all classes of *short object references*.

See:

- "Reference Summary" on page 638 for an overview of member functions
- "Reference Index" on page 639 for a list of member functions

About Short Object References

Short object references enable you to save space when storing persistent references to basic objects in attributes and associations. Short object references are alternatives to standard object references (instances of `ooRef(ooObj)` and its derived classes); both kinds of object reference store the object identifier of a referenced basic object, with the following difference:

- A standard object reference stores all components of the object identifier, including the database, container, logical page, and logical slot numbers.
- A short object reference truncates the object identifier, storing just its lower half (the logical page and slot numbers).

Short object references occupy about half the space of standard object references, so they can be useful when disk usage is a concern for a federated database that must maintain a large number of references to basic objects. However, a truncated object identifier provides only enough information to locate a basic object within its container. Consequently, a short object reference must be used as an attribute or in an association of a persistent object, which determines the necessary container and database information:

- If the referencing persistent object is a basic object, the referenced object is assumed to be in the same container.

- If the referencing persistent object is a container, the referenced object is assumed to be in that container.

You can use a short object reference of type `ooShortRef(ooObj)` to reference a basic object of any type. Normally, however, a basic object of a particular type is referenced by a short object reference of the corresponding type—that is, a basic object of an application-defined class *`appClass`* is referenced by instances of `ooShortRef(appClass)`.

Working With Short Object References

An application normally creates a short object reference implicitly as a data member or a short inline association of a persistent object. An application should not explicitly define subclasses of the `ooShortRef(ooObj)` classes; any necessary subclasses are generated automatically by the DDL processor if the application defines any subclasses of `ooObj`.

A new short object reference is normally *null*—that is, it contains the value 0 instead of an object identifier. The application can then set the short object reference to reference a particular basic object in any 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.

Unlike standard object references, short object references cannot be used as smart pointers. That is, you cannot use a short object reference to operate on the referenced basic object or to access its members. Rather, the member functions defined by the short object reference classes are limited to operations that apply to short object references themselves, such as comparing, testing, and printing.

You can assign a short object reference to a standard object reference or handle. The upper half of the object identifier is typically supplied by calling `set_container` on the standard object reference or handle.

Reference Summary

Creating	`ooShortRef(ooObj)`
Setting	`operator=`

Comparing	operator== operator!=
Getting Information	print sprint
Testing	isNull operator int

Reference Index

isNull	Tests whether this short object reference is null.
ooShortRef(ooObj)	Default constructor that constructs a null short object reference.
ooShortRef(ooObj)	Constructs a new short object reference that references the same basic object as the specified short object reference, standard object reference, handle, or pointer.
operator=	Assignment operator; sets this short object reference to the same basic object as the specified short object reference, standard object reference, handle, or pointer.
operator==	Equality operator; tests whether this short object reference has the same value as the specified item.
operator!=	Inequality operator; tests whether this short object reference has a different value from the specified item.
operator int	Conversion operator that tests whether this short object reference is null.
print	Prints the object identifier of the referenced basic object.
sprint	Returns a string containing the object identifier of the referenced basic object.

Constructors

ooShortRef(ooObj)

Default constructor that constructs a null short object reference.

```
ooShortRef(ooObj)();
```

ooShortRef(ooObj)

Constructs a new short object reference that references the same basic object as the specified short object reference, standard object reference, handle, or pointer.

1. `ooShortRef(ooObj)(`
   `const ooShortRef(ooObj) &`*`object`*`);`
2. `ooShortRef(ooObj)(`
   `const `*`ooRefHandle`*`(ooObj) &`*`object`*`);`
3. `ooShortRef(ooObj)(`
   `const ooObj *`*`objectP`*`);`

Parameters

`object`

Short object reference, standard object reference, or handle to an instance of *`appClass`*.

`objectP`

Pointer to an instance of *`appClass`*. The pointer may *not* be the result of `operator new` on *`appClass`*. Instead, the pointer must be the result of using `operator appClass*` on a handle *or* `ptr` on an object reference or handle earlier in the same transaction. If the specified pointer was originally extracted from a handle, that handle must still exist and reference the same object.

Discussion

If you use Variant 2 to construct a short object reference from a standard object reference or handle, the database and container portion of the object identifier are discarded.

Variant 3 allows you to resume persistence operations on a basic object after manipulating it through a pointer. The use of this variant should be rare, however, because pointers are not normally used to manipulate basic objects.

Operators

operator=

Assignment operator; sets this short object reference to the same basic object as the specified short object reference, standard object reference, handle, or pointer.

1. `ooShortRef(ooObj) &operator=(`
   `const ooShortRef(ooObj) &`*`object`*`);`
2. `ooShortRef(ooObj) &operator=(`
   `const `*`ooRefHandle`*`(ooObj) &`*`object`*`);`

```
3.   ooShortRef(ooObj) &operator=(
       const ooObj *objectP);
```

Parameters

object

Short object reference, standard object reference, or handle to an instance of *appClass*.

objectP

0, or a nonnull pointer to an instance of *appClass*. The pointer may *not* be the result of `operator new` on *appClass*. Instead, the pointer must be the result of using `operator appClass*` on a handle *or* `ptr` on an object reference or handle earlier in the same transaction. If the specified pointer was originally extracted from a handle, that handle must still exist and reference the same object.

Returns

This short object reference.

Discussion

If you use Variant 2 to assign a standard object reference or handle to a short object reference, the database and container portion of the object identifier are discarded.

Variant 3 allows you to set this short object reference to null. Otherwise, assignment-from-pointer enables you to resume persistence operations on a basic object after manipulating it through a pointer. The use of this variant should be rare, however, because pointers are not normally used to manipulate basic objects.

operator==

Equality operator; tests whether this short object reference has the same value as the specified item.

```
1.   ooBoolean operator==(
       const ooShortRef(ooObj) &compare) const;
2.   ooBoolean operator==(
       const ooRefHandle(ooObj) &compare) const;
3.   ooBoolean operator==(
       const ooObj *compare) const;
4.   ooBoolean operator==(int zero) const;
```

Parameters

compare

Short object reference, object reference, handle, or pointer to a basic object.

zero

Literal 0. This value allows you to use `operator==` as an alternative for `isNull`.

Returns

Variants 1, 2, and 3 return `oocTrue` if this short object reference references the same object as *compare*; otherwise `oocFalse`.
Variant 4 returns `oocTrue` if this short object reference is null; otherwise `oocFalse`.

Discussion

Variants 1, 2, and 3 test whether this object reference or handle references the same basic object as *compare*. If you specify a standard object reference or handle, this operator considers only the lower half of the object identifier, ignoring the database and container information. The comparison is most meaningful when the object referenced by *compare* resides in the same container as the object referenced by this short object reference.

Variant 4 tests whether this object reference or handle is null.

See also

`isNull` (as an alternative to variant 4)

operator!=

Inequality operator; tests whether this short object reference has a different value from the specified item.

```
1.  ooBoolean operator!=(
      const ooShortRef(ooObj) &compare) const;
2.  ooBoolean operator!=(
      const ooRefHandle(ooObj) &compare) const;
3.  ooBoolean operator!=(
      const ooObj* compare) const;
4.  ooBoolean operator!=(int zero) const;
```

Parameters

compare

Short object reference, object reference, handle, or pointer to a basic object.

zero

Literal 0.

Returns

Variants 1, 2, and 3 return `oocTrue` if this short object reference does not reference the same object as *compare*; otherwise `oocFalse`.
Variant 4 returns `oocTrue` if this short object reference is not null; otherwise `oocFalse`.

Discussion Variants 1, 2, and 3 test whether this object reference or handle references a different object than *compare*. If you specify a standard object reference or handle, this operator considers only the lower half of the object identifier, ignoring the database and container information. The comparison is most meaningful when the object referenced by *compare* resides in the same container as the object referenced by this short object reference.

Variant 4 tests whether this object reference or handle is nonnull.

operator int

Conversion operator that tests whether this short object reference is null.

```
operator int() const;
```

Returns 0 if this short object reference is null; otherwise, returns a nonzero integer.

Discussion This conversion operator enables you to use a short object reference as the conditional expression in an `if` or `while` statement to test whether the short object reference is null.

Example
```
ooShortRef(ooObj) objectR;
... // Set objectR to some object
if (objectR) {
... // Do something interesting if initialization was successful
}
```

Member Functions

isNull

Tests whether this short object reference is null.

```
ooBoolean isNull() const;
```

Returns `oocTrue` if this short object reference is null; otherwise `oocFalse`.

Discussion You can use this member function as an alternative to comparing this short object reference to 0 with `operator==`.

See also `operator==`

print

Prints the object identifier of the referenced basic object.

```
void print(FILE *outputFile = stdout) const;
```

Parameters

outputFile

Pointer to the file in which to print the object identifier. The default is standard output.

Discussion

The object identifier is printed in #*P*-*S* format, which identifies the logical page number (*P*) and logical slot number (*S*) of the basic object—for example, #3-12.

sprint

Returns a string containing the object identifier of the referenced basic object.

```
char *sprint(char *buffer = 0) const;
```

Parameters

buffer

String in which to return the object identifier. If you omit this parameter, `sprint` statically allocates a new string.

Returns

String representing the object identifier.

Discussion

The object identifier is printed in #*P*-*S* format, which identifies the logical page number (*P*) and logical slot number (*S*) of the basic object—for example, #3-12.

Each successive invocation of this member function without the parameter overwrites the statically allocated string. You should make a local copy of the returned string if you intend to use it later in the application.

ooString(N) Class

Inheritance: **ooString(N)**

The non-persistence-capable class ooString(*N*) represents an *optimized string* based on a VArray of characters and a fixed character array of length *N*.

See:

- "Reference Summary" on page 647 for an overview of member functions
- "Reference Index" on page 648 for a list of member functions

NOTE The name ooString(*N*) is a macro that expands to a template class whose parameter is *N*.

About Optimized Strings

An optimized string is a character string of any length that can be stored in a persistent object. Although an optimized string is able to store any number of characters, it is optimized for strings that are less than a particular length. You can use optimized strings anywhere in an application; however, their primary purpose is to serve as string attributes of persistence-capable classes (in place of C++ `char *` strings, which cannot be stored persistently). You can convert transparently between an optimized string and a `const char *` string, enabling optimized strings to be passed to functions as parameters of type `const char *` and vice versa.

Choosing Optimized Strings

Objectivity/C++ provides two kinds of strings that can be stored persistently—variable-size strings and optimized strings. If you know that the strings to be stored are generally less than *N* characters long, you should choose

optimized strings of class `ooString(N)`. Otherwise, you should use variable-size strings if you cannot predict the lengths of the strings to be stored or if you know these lengths will vary widely.

Structure of Optimized Strings

An optimized string of class `ooString(N)` contains a VArray of characters and a fixed 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 array, and the vector portion of the VArray is not allocated. Otherwise, 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 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 less than 8 bytes, 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 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.

You should choose a value for *N* (the length of the fixed character array in the class) so that a high percentage (for example, 90%) of the strings in the class have a length less than *N*. It is preferable that *N* be an even number. *N* must take into account the terminating null needed by C++ strings.

An optimized string allocates the fixed 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

Like instances of any other non-persistence-capable class, optimized strings are not independently persistent. However, when an optimized string is an attribute of a persistent object, it is saved in the federated database when the persistent object is saved.

You use public constructors to create optimized strings with 0 or more characters. You can initialize an empty optimized string by assigning a C++ string to it, and you can use the `resize` member function to grow or truncate the optimized string dynamically. Resizing an optimized string to 0 removes all its characters.

You get each character by its position in the optimized string. Characters are numbered starting with 0; the position number is the character's *index* or *subscript*. Operations on an optimized string verify that any specified indexes are valid based on the string's current size. The `length` of an optimized string is the number of characters up to the first null character.

Because of the way an optimized string is represented, you cannot get the first character by dereferencing the string; that is, the expression `*myVString` does not get 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.

Reference Summary

Creating	`ooString(N)`
Assigning	`operator=`
Type Conversion	`operator const char *` `operator ooVString`

Modifying	`operator=`
	`operator+=`
	`resize`

Finding Characters	`head` `operator[ ]`
Getting Information	`length`
Testing	`operator==` `operator!=`

Reference Index

Constructors and Destructors

ooString(*N*)

Default constructor that constructs a new optimized string whose size is 0.

```
ooString(N)();
```

ooString(*N*)

Constructs a new optimized string containing a copy of the characters in the specified string.

```
1.  ooString(N)(const char *p);
2.  ooString(N)(const ooString(N) &s);
3.  ooString(N)(const ooVString &s);
```

Parameters

p

Existing C++ string from which to construct the new optimized string.

s

Existing optimized string or variable-size string from which to construct the new optimized string.

Discussion

If *p* is null or if the length of *s* is 0, an uninitialized string of size 0 is created.

Operators

operator[ ]

Subscript operator; gets the specified character of this optimized string.

```
char &operator[](const uint32 index) const;
```

Parameters

index

Index of the character to get. Specify 0 to get the first character.

Returns

index'th character of this optimized string.

Discussion

An error is reported if the index is not within the allocated size of the optimized string (including the terminating null character).

operator+=

Append-to operator; concatenates this optimized string with the specified C++ string.

```
ooString(N) &operator+=(const char *p);
```

Parameters

p

C++ string whose characters are to be concatenated.

Returns This optimized string.

Discussion The concatenation operator adds the characters pointed to by *p* to the end of this optimized string.

You can use this operator to concatenate another optimized string to this one, because `operator const char *` automatically converts the string being concatenated to `const char *`.

operator=

Assignment operator; assigns a copy of the specified C++ string to this optimized string.

```
ooString(N) &operator=(const char *p);
```

Parameters *p*

C++ string whose characters are to be assigned.

Returns This optimized string.

Discussion The assignment operation resizes this optimized string to be the same size as *p*, and then copies the characters of *p* into this optimized string. Any characters already in this optimized string are overwritten. If *p* is null, this optimized string is in effect deleted and replaced with an empty string.

You can use this operator to assign another optimized string to this one, because `operator const char *` automatically converts the string being assigned to `const char *`.

operator==

Equality operator; tests whether this optimized string matches the specified string.

```
1.  ooBoolean operator==(const char *p) const;
2.  ooBoolean operator==(const ooString(N) &s) const;
3.  ooBoolean operator==(const ooVString &s) const;
```

Returns `oocTrue` if every character of this string matches the corresponding character of the other string; otherwise, `oocFalse`.

Discussion You can compare this optimized string to a C++ string (variant 1), to another optimized string (variant 2) or to a variable-size string (variant 3).

operator!=

Inequality operator; tests whether this optimized string is different from the specified string.

```
1.  ooBoolean operator!=(const char *p) const;
2.  ooBoolean operator!=(const ooString(N) &s) const;
3.  ooBoolean operator!=(const ooVString &s) const;
```

Returns `oocTrue` if any character of this string differs from the corresponding character of the other string; otherwise, `oocFalse`.

Discussion You can compare this optimized string to a C++ string (variant 1), to another optimized string (variant 2), or to a variable-size string (variant 3).

operator const char *

Conversion operator that accesses this optimized string as an object of type `const char *`.

```
operator const char *() const;
```

operator ooVString

Conversion operator that accesses this optimized string as an object of type `ooVString`.

```
operator const ooVString() const;
```

Discussion This operator copies the characters into the resulting variable-size string instead of sharing the same storage.

Member Functions

head

Gets a pointer to the first character of this optimized string.

```
char *head() const;
```

Returns Pointer to the first character of this optimized string.

length

Gets the number of characters in this optimized string.

```
uint32 length() const;
```

Returns Integer number of characters in this string preceding the first null terminating character as computed by `strlen`. If the string contains no characters, returns 0.

Discussion The actual number of bytes allocated is at least *`length()`* `+ 1` because an extra byte is reserved for the null terminating character. (If the string contains an embedded null character, the bytes beyond that null character are not included in the returned length.)

resize

Extends or truncates this optimized string to the specified number of characters.

```
ooStatus resize(const uint32 newLength);
```

Parameters *`newLength`*

Number of characters this optimized string is to have. Specify 0 to remove all the characters. If *`newLength`* is less than *`N`*, only the fixed-array portion of the optimized string is used, freeing any storage allocated to the optimized string's vector.

Returns `oocSuccess` if successful; otherwise `oocError`.

Discussion The actual number of bytes allocated is *`newLength`* `+ 1` because an extra byte is reserved for the terminating null character, 0, which is automatically inserted.

ooTrans Class

Inheritance: **`ooTrans`**

The non-persistence-capable class `ooTrans` represents a *transaction object*, which you can use to start and terminate a series of transactions against an Objectivity / DB federated database.

See:

- “Reference Summary” on page 654 for an overview of member functions
- “Reference Index” on page 654 for a list of member functions

(*ODMG*) The `ooTrans` class is equivalent to the ODMG standard class `d_Transaction`. You can use the name `d_Transaction` interchangeably with `ooTrans`.

About Transaction Objects

An application uses one or more transaction objects to start and stop its transactions. A single-threaded application normally creates a single transaction object; a multithreaded application normally creates one transaction object in each Objectivity context that is to execute transactions. A particular transaction object can be used to start and stop any number of 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.

Every transaction has an integer identifier that uniquely identifies it to the lock server. A transaction’s identifier is assigned when the federated database is

opened in that transaction. Administration tools such as `oolockmon` and `oolistwait` refer to a transaction using its identifier. Recovery functions use parameters of this type to identify a transaction of interest.

The way you start a transaction determines its concurrent access policy—that is, whether it is a *standard transaction* or a *multiple readers, one writer* (MROW) transaction. The concurrent access policy determines whether a requested read lock is considered compatible with an existing update lock on a container. However, a separate operation (namely, opening the federated database) determines whether the transaction is a *read transaction* or an *update transaction*; see *ooRefHandle*(ooFDObj)::open.

Reference Summary

Creating a Transaction Object	ooTrans
Controlling Transactions	abort commit commitAndHold start
Getting Information	getID
Testing the Transaction Object	isActive
Object Conversion	upgrade
ODMG Interface	begin checkpoint

Reference Index

abort	Terminates the currently active transaction on this transaction object, and aborts (does not apply) changes to the federated database.
begin	(ODMG) Starts a new transaction on this transaction object.
checkpoint	(ODMG) Checkpoints the currently active transaction on this transaction object.
commit	Terminates the currently active transaction on this transaction object, and commits all changes to the database.

commitAndHold	Checkpoints the currently active transaction on this transaction object.
getID	Gets the identifier of the currently active transaction on this transaction object.
isActive	Tests whether a transaction has been started on this transaction object.
ooTrans	Default constructor that constructs a new transaction object.
start	Starts a new transaction on this transaction object.
upgrade	Identifies the application containing this transaction object as a special-purpose upgrade application for converting objects after schema evolution.

Constructors

ooTrans

Default constructor that constructs a new transaction object.

```
ooTrans();
```

Member Functions

abort

Terminates the currently active transaction on this transaction object, and aborts (does not apply) changes to the federated database.

```
ooStatus abort(ooHandleMode mode = oocHandleToNull);
```

Parameters *mode*

Determines what happens to the transaction's open handles after the transaction is aborted:

- Omit this parameter (or specify `oocHandleToNull`) to set all open handles to null.
- Specify `oocHandleToOID` to close all open handles, which invalidates their pointers, but preserves the object identifiers they contain. You should test each closed handle for validity before reusing it in another transaction, however, to ensure that the referenced object has not been

deleted or moved by another process between your process's transactions.

Returns `oocSuccess` if successful; otherwise `oocError`.

begin

(*ODMG*) Starts a new transaction on this transaction object.

```
void begin();
```

Discussion This member function calls `start` with default parameter values.

checkpoint

(*ODMG*) Checkpoints the currently active transaction on this transaction object.

```
void checkpoint();
```

Discussion Checkpointing a transaction terminates it, commits all changes to the database, and implicitly starts a new transaction.

This member function calls `commitAndHold` with the default parameter value.

commit

Terminates the currently active transaction on this transaction object, and commits all changes to the database.

```
ooStatus commit();
```

Returns `oocSuccess` if successful; otherwise `oocError`.

Discussion If the federated database is open, this member function closes it.

commitAndHold

Checkpoints the currently active transaction on this transaction object.

```
ooStatus commitAndHold(
     ooDowngradeMode mode = oocNoDowngrade);
```

Parameters *mode*

Mode in which update locks are to be treated:

- Omit this parameter (or specify `oocNoDowngrade`) to preserve all locks held by the transaction as is.

- Specify `oocDowngradeAll` to change all locks to read locks (MROW read if the transaction is an MROW transaction, and normal read locks otherwise).

Returns `oocSuccess` if successful; otherwise `oocError`.

Discussion Checkpointing a transaction terminates it, commits all changes to the database, and implicitly starts a new transaction. By default, all locks acquired during the transaction are preserved as is.

getID

Gets the identifier of the currently active transaction on this transaction object.

```
ooTransID getID() const;
```

Returns Identifier of the current transaction, if the identifier has been assigned; otherwise 0.

Discussion Every transaction has an integer identifier that uniquely identifies it to the lock server. A transaction's identifier is assigned when the federated database is opened in that transaction. Consequently, you should call `getID` after you call `open` on a handle to the federated database. If `getID` is called before a the transaction's identifier is assigned or after the transaction ends, the value 0 is returned.

Administration tools such as `oolockmon` and `oolistwait` refer to a transaction using its identifier. Recovery functions use parameters of this type to identify a transaction of interest.

isActive

Tests whether a transaction has been started on this transaction object.

```
ooBoolean isActive();
```

Returns `oocTrue` if this transaction object is active; otherwise `oocFalse`.

start

Starts a new transaction on this transaction object.

```
ooStatus start(
     ooMode openMode = oocNoMROW,
     const int32 waitOption = oocTransNoWait,
     ooIndexMode indexMode = oocInsensitive);
```

Parameters *openMode*

Concurrent access policy for the newly started transaction. Objectivity / DB uses this policy to determine whether the locks requested or held by this transaction are compatible with those of other transactions:

- Specify `oocNoMROW` (the default) to enable the standard concurrent access policy. This policy allows multiple transactions to lock the same container for read, but prevents concurrent read and update locks on a container.
- Specify `oocMROW` to enable the MROW concurrent access policy. This policy allows multiple transactions to lock the same container for read while one transaction locks it for update.

waitoption

Lock-waiting behavior for the newly started transaction:

- Specify `oocTransNoWait` (the default) to use the default lock-waiting option currently in effect for the Objectivity context (see the `ooSetLockWait` global function).
- Specify `oocNoWait` or 0 to turn off lock waiting for the transaction.
- Specify `oocWait` to cause the transaction to wait indefinitely for locks.
- Specify a timeout period of *n* seconds for the transaction, where *n* is an integer in the range 1 <= *n* <= 14400. If *n* is less than 0 or greater than 14400, it is treated as `oocWait`.

Lock waiting does not apply to MROW read transactions. Therefore, any *waitOption* value you specify is ignored when you set the *openMode* parameter to `oocMROW`.

indexMode

Policy for updating indexes when objects of an indexed class are created or key-field values are modified:

- Specify `oocInsensitive` (the default) to update all applicable indexes automatically when the transaction commits.
- Specify `oocSensitive` to update all applicable indexes automatically when the next predicate scan is performed in the transaction or, if no scans are performed, when the transaction commits. This allows you to change indexed objects and then scan them in the same transaction using any applicable index. (Note, however, that the transaction must commit before the updates are available to other transactions.)
- Specify `oocExplicitUpdate` to suppress automatic index updates; the application must update indexes explicitly by calling the `ooUpdateIndexes` global function after every relevant change.

Returns `oocSuccess` if successful; otherwise `oocError`.

Discussion

After you call this member function, the first Objectivity / DB operation of the transaction must be to open the federated database by calling the open member function on a federated-database handle.

upgrade

Identifies the application containing this transaction object as a special-purpose *upgrade application* for converting objects after schema evolution.

```
ooStatus upgrade();
```

Discussion

You must call this member function before starting the first (and only) transaction in an upgrade application. This transaction must be an update transaction (that is, the transaction must open the federated database for update) and it must also call the *ooRefHandle*(ooFDObj)::upgradeObjects member function to initiate the upgrade process.

Object conversion is the process of making existing persistent objects consistent with class definition changes introduced by schema evolution. Certain schema evolution operations affect how instances of a class should be laid out in storage. After you perform such operations, existing objects of the changed classes are rendered out-of-date until they are converted to their new representations.

In general, affected objects are converted automatically when they are accessed after schema evolution. However, some schema changes require that you convert objects explicitly using an upgrade application to ensure referential integrity.

See also

Chapter 19, “Object Conversion,” in the Objectivity / C++ programmer’s guide

ooTreeAdmin Class

Inheritance: **ooObj->ooAdmin->ooTreeAdmin**

Handle Class: `ooHandle(ooTreeAdmin)`

Object-Reference Class: `ooRef(ooTreeAdmin)`

The persistence-capable class `ooTreeAdmin` represents *tree administrators*.

See:

- "Reference Summary" on page 662 for an overview of member functions
- "Reference Index" on page 663 for a list of member functions

To use this class, your application must include the `ooCollections.h` header file. For UNIX linking information, see *Installation and Platform Notes for UNIX*.

About Tree Administrators

Each ordered collection has a tree administrator that manages the containers used by the collection's internal objects, namely the nodes of the collection's B-tree and the array of references for each node. An ordered collection's tree administrator is created when the collection itself is created.

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. The default value for this property depends on the federated database's storage page size; it is calculated as:

  pageSize / 47

Because B-tree nodes are small objects, many of them can fit on a single storage page. Because nodes are not updated frequently, many can be clustered in the same container without causing locking problems.

- 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.

For additional information, see "Tree Administrator" on page 250 in the Objectivity/C++ programmer's guide.

Working With a Tree Administrator

Like other persistent objects, tree administrators are normally manipulated through handles or object references.

You call an ordered collection's `admin` member function to obtain an object reference to the collection's tree administrator; you must then cast the returned object reference to type `ooRef(ooTreeAdmin)` before you access the tree administrator's data members.

Reference Summary

Getting Information	`maxNodesPerContainer`
	`maxVArraysPerContainer`
	`nodeContainer`
	`vArrayContainer`

Setting Information	`setMaxNodesPerContainer` `setMaxVArraysPerContainer`

Reference Index

Member Functions

maxNodesPerContainer

Gets the maximum number of B-tree nodes per container for this tree administrator.

```
ooInt32 maxNodesPerContainer();
```

Returns The maximum number of B-tree nodes that can be stored in a single container.

See also setMaxNodesPerContainer

maxVArraysPerContainer

Gets the maximum number of arrays per container for this tree administrator.

```
ooInt32 maxVArraysPerContainer();
```

Returns The maximum number of arrays that can be stored in a single container.

See also setMaxVArraysPerContainer

nodeContainer

Gets this tree administrator's current node container.

```
ooRef(ooContObj) nodeContainer();
```

Returns Object reference to this tree administrator's current node container.

setMaxNodesPerContainer

Sets the maximum number of B-tree nodes per container for this tree administrator.

```
void setMaxNodesPerContainer(ooInt32 max);
```

Parameters *max*

The maximum number of B-tree nodes that can be stored in a single container.

Discussion 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.

See also `maxNodesPerContainer`

setMaxVArraysPerContainer

Sets the maximum number of arrays per container for this tree administrator.

```
void setMaxVArraysPerContainer(ooInt32 max);
```

Parameters *max*

The maximum number of arrays that can be stored in a single container.

Discussion 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.

Changing the maximum arrays per container affects only the collection's current array container and any array containers created in the future.

See also `setMaxVArraysPerContainer`

vArrayContainer

Gets this tree administrator's current array container.

```
ooRef(ooContObj) vArrayContainer();
```

Returns Object reference to this tree administrator's current array container.

ooTreeList Class

Inheritance: **ooObj->ooCollection->ooBTree->ooTreeList**

Handle Class: `ooHandle(ooTreeList)`

Object-Reference Class: `ooRef(ooTreeList)`

The persistence-capable class `ooTreeList` represents *lists* of persistent objects.

See:

- "Reference Summary" on page 668 for an overview of member functions
- "Reference Index" on page 669 for a list of member functions

To use this class, your application must include the `ooCollections.h` header file. For UNIX linking information, see *Installation and Platform Notes for UNIX*.

About Lists

A list is an ordered collection; unlike a set, a list can contain duplicate elements and null elements. An element of a list can be located by position, given as a zero-based index. Like all collections implemented with B-trees, lists are *scalable* collections, that is, they can increase in size with minimal performance degradation.

The `ooTreeList` class overrides the inherited `add` and `addAll` member functions to add elements to the end of the list. Member functions defined in this class add elements to the list, or insert elements into it, at an indicated position. The elements of a list are kept in the order in which they were added or inserted. Note, however, that an element's position may change as elements are inserted in front of it in the list. For example, the element that was at index 2 will be at index 4 after two elements are added to the front of the list.

For additional information, see Chapter 11, “Persistent Collections,” in the Objectivity/C++ programmer’s guide.

Working With a List

As is the case for any basic object, you specify whether a list is to be transient or persistent when you create it; lists *must be persistent*. You create a list with a call to the `new` operator; the clustering directive in that call specifies where in the federated database to store the new list.

Like other persistent objects, lists are normally manipulated through handles or object references. You can store and find a list in the database just as you would any other persistent object.

Related Classes

Two additional classes represent persistent collections of persistent objects:

- `ooHashSet` represents an *unordered* collection of persistent objects *with no duplicate elements*. It uses an extendible hashing mechanism.
- `ooTreeSet` represents a sorted collection of persistent objects *with no duplicate elements; elements are sorted by the corresponding comparator* (or by increasing OID if the sorted set uses the default comparator). Like this class, `ooTreeSet` is implemented using a B-tree data structure.

Reference Summary

In the following table:

- Operators indicated as *(inherited)* are overloaded in this class with no change in behavior; they are documented with the `ooObj` class (page 431), along with the other inherited member functions not listed here.
- Member function indicated as *(inherited)* are inherited from the `ooBTree` class (page 283) or the `ooCollection` class (page 173) and are documented with the defining class.

Creating and Deleting	ooTreeList operator new *(inherited)* operator delete *(inherited)*
Adding, Removing, and Changing Elements	add addAll addFirst addLast clear *(inherited)* remove removeAll *(inherited)* removeAllDeleted removeRange retainAll *(inherited)* set
Getting Elements	first get iterator last *(inherited)*
Getting Indexes	indexOf *(inherited)* lastIndexOf *(inherited)*
Getting Information	depth *(inherited)* size *(inherited)*
Finding Auxiliary Objects	admin comparator
Testing	contains containsAll *(inherited)* isEmpty *(inherited)*
Maintaining the B-Tree	compact *(inherited)*
Viewing in an MROW Transaction	refresh *(inherited)*

Reference Index

add	Adds the specified object to this list.
addAll	Adds all elements (or keys) in the specified collection to this list.

addFirst	Adds the specified object to the beginning of this list.
addLast	Adds the specified object to the end of this list.
admin	Finds the tree administrator for this list.
comparator	Overrides the inherited member function; disallows finding the comparator for lists.
contains	Tests whether this list contains the specified object.
first	Finds the first element in this list.
get	Finds the specified element of this list.
iterator	Initializes a scalable-collection iterator to find the elements of this list.
ooTreeList	Constructs a new list.
remove	Removes the first occurrence of the specified object from this list.
removeAllDeleted	Removes from this list all persistent objects that have been deleted from the federated database.
removeRange	Removes from this list all elements with indexes in the specified range.
set	Replaces the element at the specified index with the specified object.

Constructors

ooTreeList

Constructs a new list.

```
ooTreeList(
     int maxNodeSize = (oomGetPageSize() - 92) / 8,
     ooHandle(ooContObj) contAdminH = 0,
     ooHandle(ooContObj) contVarrayH = 0);
```

Parameters *maxNodeSize*

The node size for the new list's B-tree.

contAdminH

Handle to the container in which to store the tree administrator for the new lists.

contVarrayH

Handle to the initial array container for the new lists.

Discussion

The constructor creates an empty list. The optional parameters allow the caller to:

- Override the default node size.
- Prevent creation of a container for the list's tree administrator.
- Prevent creation of the list's initial array container.

Member Functions

add

Adds the specified object to this list.

```
1.   virtual ooBoolean add(
       const ooHandle(ooObj) &objH);
2.   ooStatus add(
       const ooInt32 index,
       const ooHandle(ooObj) &objH);
```

Parameters

objH

Handle to the object to be added.

index

The index where the new element is to be inserted.

Returns

(Variant 1) `oocTrue` if an element was added; otherwise, `oocFalse`.

(Variant 2) `oocSuccess` if successful; otherwise, `oocError`.

Discussion

Variant 1 overrides the inherited member function to add the specified object at the end of this list.

Variant 2 inserts the specified object into this list at the specified index, increasing the size of the list by one and effectively incrementing the index of all subsequent elements. For example, if *index* is 2, this member function inserts the new element before the third existing element. The new element now has index 2 and what used to be the third element is now the fourth element (and has index 3).

See also
addAll
addFirst
addLast
remove
set

addAll

Adds all elements (or keys) in the specified collection to this list.

```
1.  ooStatus addAll(
      int index,
      const ooHandle(ooCollection) &collectionH);
2.  ooBoolean addAll(
      const ooHandle(ooCollection) &collectionH);
```

Parameters

index

The index where the first of the new elements is to be inserted.

collectionH

Handle to the collection whose elements are to be added to this list. If the elements of the collection are key-value pairs, its keys are added to this list.

Note: *collectionH* may done be a handle to this list. That is, this method cannot be used to add another copy of this list's elements to this list.

Returns

(Variant 1) `oocSuccess` if successful; otherwise, `oocError`.

(Variant 2) `oocTrue` if any elements were added; otherwise, `oocFalse`.

Discussion

Variant 1 inserts elements (or keys) of the specified collection into this list at the indicated index. It fails if *index* is not a valid index for this list.

Variant 2 overrides the inherited method to add the new elements to the end of this list.

See also
add
addFirst
addLast

addFirst

Adds the specified object to the beginning of this list.

```
ooStatus addFirst(const ooHandle(ooObj) &objH);
```

Parameters *objH*

Handle to the object to be added.

Returns `oocSuccess` if successful; otherwise, `oocError`.

Discussion The new element becomes the first element of the list, effectively incrementing the index of all existing elements.

See also `add`
`addAll`
`addLast`

addLast

Adds the specified object to the end of this list.

```
ooStatus addLast(const ooHandle(ooObj) &objH);
```

Parameters *objH*

Handle to the object to be added.

Returns `oocSuccess` if successful; otherwise, `oocError`.

See also `add`
`addAll`
`addFirst`

admin

Finds the tree administrator for this list.

```
virtual ooRef(ooAdmin) admin() const;
```

Returns Object reference to the tree administrator for this list.

Discussion You typically call this member function when you want to change the way that this list's internal objects (B-tree nodes and the arrays they reference) are assigned to containers. Before you do so, you must cast the returned object reference to `ooRef(ooTreeAdmin)`.

comparator

Overrides the inherited member function; disallows finding the comparator for lists.

```
virtual ooRef(ooCompare) comparator() const;
```

Returns A null object reference.

Discussion Lists do not use comparators, so this member function always returns null.

contains

Tests whether this list contains the specified object.

```
virtual ooBoolean contains(
     const ooHandle(ooObj) &objH) const;
```

Parameters *objH*

Handle to the element to be tested for containment in this list.

Returns `oocTrue` if this list contains an element equal to the specified object; otherwise, `oocFalse`.

first

Finds the first element in this list.

```
virtual ooRef(ooObj) first() const;
```

Returns Object reference to the first element of this list.

See also `get`

get

Finds the specified element of this list.

```
1.   ooRef(ooObj) get(const ooInt32 index) const;
2.   virtual ooRef(ooObj) get(const void *lookupVal) const;
```

Parameters *index*

The zero-based index of the desired element.

lookupVal

Pointer to data that identifies the desired element.

Returns (Variant 1) Finds the element at the specified index; it returns an object reference to the element whose index is *index*.

(Variant 2) Overrides the inherited member function to disallow looking up a list element by data that identifies the element. It always returns a null object reference.

Discussion Variant 2 requires the collection to use an application-defined comparator that can identify an element based on class-specific data. Because a list has no comparator, variant 2 is not relevant for lists.

See also `first`

iterator

Initializes a scalable-collection iterator to find the elements of this list.

```
virtual ooCollectionIterator *iterator() const;
```

Returns A pointer to a scalable-collection iterator for finding the elements of this list; the caller is responsible for deleting the iterator when it is no longer needed.

Discussion The returned iterator finds the elements as ordered in this list.

You must delete the iterator when you have finished using it.

remove

Removes the first occurrence of the specified object from this list.

```
virtual ooBoolean remove(
     const ooHandle(ooObj) &objH);
```

Parameters *objH*

Handle to the object to be removed.

Returns `oocTrue` if an element was removed; otherwise, `oocFalse`.

See also `add`
`removeAllDeleted`
`removeRange`

removeAllDeleted

Removes from this list all persistent objects that have been deleted from the federated database.

```
virtual void removeAllDeleted();
```

Discussion You can calling this member function to restore this list's referential integrity.

removeRange

Removes from this list all elements with indexes in the specified range.

```
ooStatus removeRange(
     int fromIndex,
     int toIndex);
```

Parameters

fromIndex

The index of the first element to be removed.

toIndex

The index of the last element to be removed.

Returns

`oocSuccess` if successful; otherwise, `oocError`.

See also

`remove`

set

Replaces the element at the specified index with the specified object.

```
ooStatus set(
     const int index,
     const ooHandle(ooObj) &objH);
```

Parameters

index

The index of the element to be replaced.

objH

Handle to the object that is to replace the existing element at the specified index.

Returns

`oocSuccess` if successful; otherwise, `oocError`.

See also

`add`

ooTreeMap Class

Inheritance: **ooObj->ooCollection->ooBTree->ooTreeSet->ooTreeMap**

Handle Class: `ooHandle(ooTreeMap)`

Object-Reference Class: `ooRef(ooTreeMap)`

The persistence-capable class `ooTreeMap` represents *sorted object maps*.

See:

- "Reference Summary" on page 678 for an overview of member functions
- "Reference Index" on page 679 for a list of member functions

To use this class, your application must include the `ooCollections.h` header file. For UNIX linking information, see *Installation and Platform Notes for UNIX*.

About Sorted Object Maps

An object map is a collection of key-value pairs; each key and each value is a persistent object. No two elements of the object map may have the same key. As the name implies, each element of an object map is a mapping from its key object to its value object.

Like all collections implemented with B-trees, sorted object maps are *scalable* collections, that is, they can increase in size with minimal performance degradation.

The elements of a sorted object map are sorted by their keys according to the ordering implemented by the object map's corresponding comparator. If a sorted object map has a default comparator, its elements are sorted by the object identifiers (OIDs) of their keys.

For additional information, see Chapter 11, “Persistent Collections,” in the Objectivity/C++ programmer’s guide.

Working With a Sorted Object Map

As is the case for any basic object, you specify whether a sorted object map is to be transient or persistent when you create it; sorted object maps *must be persistent*. You create a sorted object map with a call to the `new` operator; the clustering directive in that call specifies where in the federated database to store the new sorted object map.

Like other persistent objects, sorted object maps are normally manipulated through handles or object references. You can store and find a sorted object map in the database just as you would any other persistent object.

Related Classes

Two additional classes represent persistent collections of key-value pairs:

- `ooHashMap` represents an *unordered* object map. It uses an extendible hashing mechanism.
- `ooMap` represents an *unordered* name map, that is, a collection of key-value pairs in which *the key is a string* and the value is an object reference to a persistent object. It uses a traditional (non-extendible) hashing mechanism.

Reference Summary

In the following table:

- Operators indicated as *(inherited)* are overloaded in this class with no change in behavior; they are documented with the `ooObj` class (page 431), along with the other inherited member functions not listed here.
- Member function indicated as *(inherited)* are inherited from the `ooTreeSet` class (page 283), the `ooBTree` class (page 283), or the `ooCollection` class (page 173) and are documented with the defining class.

Creating and Deleting	ooTreeMap operator new *(inherited)* operator delete *(inherited)*
Adding, Removing, and Changing Elements	add addAll clear *(inherited)* put remove removeAll *(inherited)* removeAllDeleted retainAll *(inherited)*
Getting Elements	first *(inherited)* get keyIterator *(inherited)* valueIterator last *(inherited)*
Getting Indexes	indexOf *(inherited)* lastIndexOf *(inherited)*
Getting Information	depth *(inherited)* size *(inherited)*
Finding Auxiliary Objects	admin *(inherited)* comparator *(inherited)*
Testing	containsKey containsValue containsAll *(inherited)* isEmpty *(inherited)*
Maintaining the B-Tree	compact
Viewing in an MROW Transaction	refresh *(inherited)*

Reference Index

add	Adds the specified object to this sorted object map.
addAll	Adds all elements in the specified object map to this sorted object map.
compact	Minimizes the number of nodes in this sorted object map's B-tree.

containsKey	Tests whether this sorted object map contains an element with the specified key.
containsValue	Tests whether this sorted object map contains an element with the specified value.
get	Finds the value paired with the specified key in this sorted object map.
ooTreeMap	Constructs a new sorted object map.
put	Maps the specified key to the specified value in this sorted object map.
remove	Removes the element, if any, with the specified key from this sorted object map.
removeAllDeleted	Removes from this sorted object map all elements in which either the key or the value has been deleted from the federated database.
valueIterator	Initializes a scalable collection iterator to find all values in this sorted object map.

Constructors

ooTreeMap

Constructs a new sorted object map.

```
1.   ooTreeMap(
       int maxNodeSize = (oomGetPageSize() - 92) / 8,
       ooHandle(ooContObj) contAdminH = 0,
       ooHandle(ooContObj) contVarrayH = 0);
2.   ooTreeMap(
       ooHandle(ooCompare) &compH,
       int maxNodeSize = (oomGetPageSize() - 92) / 8,
       ooHandle(ooContObj) contAdminH = 0,
       ooHandle(ooContObj) contVarrayH = 0);
```

Parameters

maxNodeSize

The node size for the new sorted object map's B-tree.

contAdminH

Handle to the container in which to store the tree administrator for the new sorted object map.

contVarrayH

Handle to the initial array container for the new sorted object map.

compH

Handle to the comparator for the new sorted object map; must be an instance of an application-specific derived class of `ooCompare`.

Discussion

Variant 1 creates an empty sorted object map with a default comparator; Variant 2 creates an empty sorted object map with the specified comparator. The optional parameters to both constructors allow the caller to:

- Override the default node size.
- Prevent creation of a container for the sorted object map's tree administrator.
- Prevent creation of the sorted object map's initial array container.

Member Functions

add

Adds the specified object to this sorted object map.

```
virtual ooBoolean add(
     const ooHandle(ooObj) &objH);
```

Parameters

objH

Handle to the object to be added.

Returns

`oocTrue` if an element was added; otherwise, `oocFalse`.

Discussion

This member function adds a new key-value pair to this sorted object map with the specified object as its key and a null value. If this sorted object map currently has an element whose key is the specified object, the value of the existing element is replaced by null.

See also

`addAll`
`remove`

addAll

Adds all elements in the specified object map to this sorted object map.

```
virtual ooBoolean addAll(
     const ooHandle(ooCollection) &collectionH);
```

Parameters *collectionH*

Handle to the sorted or unordered object map whose elements are to be added to this sorted object map.

Returns `oocTrue` if any elements were added; otherwise, `oocFalse`.

Discussion If the specified collection is an object map, its elements are added to this sorted object map. If this sorted object map currently has an element with the same key as an element of the specified collection, the existing element is replaced by the element of the specified collection.

If the specified collection is a collection of persistent objects, each of its elements is added as a key to this sorted object map; a null value is paired with each key. If this sorted object map currently has an element whose key is an element of the specified collection, the value of the existing element is replaced by null.

See also `add`
`get`

compact

Minimizes the number of nodes in this sorted object map's B-tree.

```
virtual void compact();
```

Discussion After you have added all elements that you expect this sorted object map to have, you can call this member function to minimize the number of nodes in its B-tree. Doing so saves space and improves read performance. Indexes of elements within the sorted object map remain unchanged.

If you call this member function before all elements have been added, insert (add) performance will not necessarily improve. After the B-tree has been compacted, adding an element will very likely cause one or more nodes to be added to the B-tree.

containsKey

Tests whether this sorted object map contains an element with the specified key.

```
1.  ooBoolean containsKey(
      const ooHandle(ooObj) &keyH);
2.  ooBoolean containsKey(
      const void *lookupVal);
```

Parameters *keyH*

Handle to the key to be tested for containment in this sorted object map.

lookupVal

Pointer to data that identifies the key to be tested for containment in this sorted object map.

Returns `oocTrue` if this sorted object map contains an element with the specified key; otherwise, `oocFalse`.

Discussion You can call this member function to check whether this sorted object map maps the specified key to some value.

Variant 2 tests whether any key is "equal" to the specified lookup data, as determined by the comparator for this sorted object map. It is useful if this sorted object map has an application-defined comparator that can identify a key based on class-specific data.

See also `containsValue`

containsValue

Tests whether this sorted object map contains an element with the specified value.

```
ooBoolean containsValue(
     const ooHandle(ooObj) &valueH) const;
```

Parameters *valueH*

Handle to the value to be tested for containment in this sorted object map.

Returns `oocTrue` if this sorted object map contains an element whose value is the specified object; otherwise, `oocFalse`.

Discussion You can call this member function to check whether this sorted object map maps at least one key to the specified value.

See also `containsKey`

get

Finds the value paired with the specified key in this sorted object map.

```
1.   ooRef(ooObj) get(const ooHandle(ooObj) &keyH) const;
2.   ooRef(ooObj) get(const void *lookupVal) const;
```

Parameters *keyH*

Handle to the key to be looked up.

lookupVal

Pointer to data that identifies the desired key.

Returns

Object reference to the value in the element with the specified key, or a null object reference if this sorted object map contains no mapping for that key.

Discussion

A return value of null does not necessarily indicate that no element has the specified key. It is possible that this sorted object map explicitly maps the key to null. You can use the `containsKey` member function to distinguish these two cases.

Variant 2 finds the element whose key is "equal" to the specified lookup data, as determined by the comparator for this sorted object map. It is useful if this sorted object map has an application-defined comparator that can identify a key based on class-specific data.

See also

`put`
`addAll`

put

Maps the specified key to the specified value in this sorted object map.

```
ooStatus put(
     const ooHandle(ooObj) &keyH,
     const ooHandle(ooObj) &valueH);
```

Parameters

keyH

Handle to the key.

valueH

Handle to the value.

Returns

`oocSuccess` if successful; otherwise, `oocError`.

Discussion

If this sorted object map already contains an element with the specified key, this member function replaces the value in that element. Otherwise, this member function adds a new element with the specified key and value.

See also

`get`
`addAll`

remove

Removes the element, if any, with the specified key from this sorted object map.

```
1.   virtual ooBoolean remove(
       const ooHandle(ooObj) &keyH);
2.   ooBoolean remove(
       const void *lookupVal);
```

Parameters

keyH

Handle to the key of the element to be removed.

lookupVal

Pointer to data that identifies the key of the element to be removed.

Returns

`oocTrue` if an element was removed; otherwise, `oocFalse`.

Variant 2 removes the element whose key is "equal" to the specified lookup data, as determined by the comparator for this sorted object map. It is useful if this sorted object map has an application-defined comparator that can identify a key based on class-specific data.

See also

`add`

removeAllDeleted

Removes from this sorted object map all elements in which either the key or the value has been deleted from the federated database.

```
virtual void removeAllDeleted();
```

Discussion

You can call this member function to restore this sorted object map's referential integrity.

See also

`remove`

valueIterator

Initializes a scalable collection iterator to find all values in this sorted object map.

```
ooCollectionIterator *valueIterator() const;
```

Returns

A pointer to a scalable collection iterator for finding all the persistent objects used as values in elements of this sorted object map. The iterator finds the values in an unspecified order.

Discussion

The returned iterator finds the values in the order in which the elements are sorted; that is, it finds the values in the sorted order of their keys.

You must delete the iterator when you have finished using it.

ooTreeSet Class

Inheritance: **ooObj->ooCollection->ooBTree->ooTreeSet**

Handle Class: `ooHandle(ooTreeSet)`

Object-Reference Class: `ooRef(ooTreeSet)`

The persistence-capable class `ooTreeSet` represents *sorted sets* of persistent objects with no duplicate elements.

See:

- "Reference Summary" on page 688 for an overview of member functions
- "Reference Index" on page 689 for a list of member functions

To use this class, your application must include the `ooCollections.h` header file. For UNIX linking information, see *Installation and Platform Notes for UNIX*.

About Sorted Sets

The elements of a sorted set are sorted according to the ordering implemented by the set's corresponding comparator. If a sorted set has a default comparator, its elements are sorted by their object identifiers (OIDs).

Like all collections implemented with B-trees, sorted sets are *scalable* collections, that is, they can increase in size with minimal performance degradation.

For additional information, see Chapter 11, "Persistent Collections," in the Objectivity/C++ programmer's guide.

Working With a Sorted Set

As is the case for any basic object, you specify whether a sorted set is to be transient or persistent when you create it; sorted sets *must be persistent*. You create a sorted set with a call to the `new` operator; the clustering directive in that call specifies where in the federated database to store the new sorted set.

Like other persistent objects, sorted sets are normally manipulated through handles or object references. You can store and find a sorted set in the database just as you would any other persistent object.

Related Classes

Two additional classes represent persistent collections of persistent objects:

- `ooHashSet` represents an *unordered* collection of persistent objects with no duplicate elements. It uses an extendible hashing mechanism.
- `ooTreeList` represents a collection of persistent objects that are *maintained in the order specified when they are added* to the collection. A list *can contain duplicate elements*. Like this class, `ooTreeList` is implemented using a B-tree data structure.

Reference Summary

In the following table:

- Operators indicated as *(inherited)* are overloaded in this class with no change in behavior; they are documented with the `ooObj` class (page 431), along with the other inherited member functions not listed here.
- Member function indicated as *(inherited)* are inherited from the `ooBTree` class (page 283) or the `ooCollection` class (page 173) and are documented with the defining class.

Creating and Deleting	ooTreeSet operator new *(inherited)* operator delete *(inherited)*
Adding and Removing Elements	add addAll *(inherited)* clear *(inherited)* remove removeAll *(inherited)* removeAllDeleted *(inherited)* retainAll *(inherited)*
Getting Elements	first *(inherited)* get iterator *(inherited)* last *(inherited)*
Getting Indexes	indexOf *(inherited)* lastIndexOf *(inherited)*
Getting Information	depth *(inherited)* size *(inherited)*
Finding Auxiliary Objects	admin comparator
Testing	contains containsAll *(inherited)* isEmpty *(inherited)*
Maintaining the B-Tree	compact *(inherited)*
Viewing in an MROW Transaction	refresh *(inherited)*

Reference Index

ooTreeSet	Constructs a new sorted set.
remove	Removes the specified object from this sorted set.

Constructors

ooTreeSet

Constructs a new sorted set.

```
1.   ooTreeSet(
       int maxNodeSize = (oomGetPageSize() - 92) / 8,
       ooHandle(ooContObj) contAdminH = 0,
       ooHandle(ooContObj) contVarrayH = 0);
2.   ooTreeSet(
       ooHandle(ooCompare) &compH,
       int maxNodeSize = (oomGetPageSize() - 92) / 8,
       ooHandle(ooContObj) contAdminH = 0,
       ooHandle(ooContObj) contVarrayH = 0);
```

Parameters

maxNodeSize

The node size for the new sorted set's B-tree.

contAdminH

Handle to the container in which to store the tree administrator for the new sorted set.

contVarrayH

Handle to the initial array container for the new sorted set.

compH

Handle to the comparator for the new sorted set; must be an instance of an application-specific derived class of ooCompare.

Discussion

Variant 1 creates an empty sorted set with a default comparator; Variant 2 creates an empty sorted set with the specified comparator. The optional parameters to both constructors allow the caller to:

- Override the default node size.
- Prevent creation of a container for the sorted set's tree administrator.
- Prevent creation of the sorted set's initial array container.

Member Functions

add

Adds the specified object to this sorted set.

```
virtual ooBoolean add(
     const ooHandle(ooObj) &objH);
```

Parameters

objH

Handle to the object to be added.

Returns

`oocTrue` if an element was added; otherwise, `oocFalse`.

Discussion

This member function returns false if the specified object is already an element of this sorted set.

See also

`remove`

admin

Gets the tree administrator for this sorted collection.

```
virtual ooRef(ooAdmin) admin() const;
```

Returns

Object reference to the tree administrator for this sorted collection.

Discussion

You typically call this member function when you want to change the way that this sorted collection's internal objects (B-tree nodes and the arrays they reference) are assigned to containers. Before you do so, you must cast the returned object reference to `ooRef(ooTreeAdmin)`.

comparator

Finds the comparator for this sorted collection.

```
virtual ooRef(ooCompare) comparator() const;
```

Returns

Object reference to the comparator for this sorted collection, or null if this sorted collection has a default comparator.

contains

Tests whether this sorted set contains the specified object.

```
1.  virtual ooBoolean contains(
      const ooHandle(ooObj) &objH) const;
2.  ooBoolean contains(
      const void *lookupVal) const;
```

Parameters

objH

Handle to the element to be tested for containment in this sorted set.

lookupVal

Pointer to data that identifies the object to be tested for containment in this sorted set.

Returns

`oocTrue` if this sorted set contains an element equal to the specified object; otherwise, `oocFalse`.

Discussion

Variant 2 tests whether any element is "equal" to the specified lookup data, as determined by the comparator for this sorted set. It is useful if this sorted set has an application-defined comparator that can identify an element based on class-specific data.

get

Gets the specified element of this sorted set.

```
1.  ooRef(ooObj) get(const ooInt32 index) const;
2.  virtual ooRef(ooObj) get(const void *lookupVal) const;
```

Parameters

index

The zero-based index of the desired element.

lookupVal

Pointer to data that identifies the desired element.

Returns

Variant 1 finds the element at the specified index; it returns an object reference to that element, or a null object reference if *index* is out of bounds.

Variant 2 finds the element that is "equal" to the specified lookup data, as determined by the comparator for this sorted set; it returns an object reference to that element, or a null object reference if this sorted set does not contain such an element.

Discussion Variant 2 is useful if this sorted set has an application-defined comparator that can identify an element based on class-specific data.

remove

Removes the specified object from this sorted set.

```
1.   virtual ooBoolean remove(
       const ooHandle(ooObj) &objH);
2.   virtual ooBoolean remove(
       const void *lookupVal);
```

Parameters *objH*

Handle to the object to be removed.

lookupVal

Pointer to data that identifies the element to be removed.

Returns `oocTrue` if an element was removed; otherwise, `oocFalse`.

Discussion Variant 2 is useful if this sorted set has an application-defined comparator that can identify an element based on class-specific data.

See also `add`

ooTVArrayT<*element_type*> Class

Inheritance: **ooTVArrayT<*element_type*>**

The non-persistence-capable template class `ooTVArrayT<element_type>` represents a *temporary variable-size array* (*VArray*) whose elements are of type `element_type`.

See:

- "Reference Summary" on page 697 for an overview of member functions
- "Reference Index" on page 697 for a list of member functions

For backward compatibility, you can use the macro-style name `ooTVArray(element_type)` instead of the name `ooTVArrayT<element_type>`.

About Temporary VArrays

A temporary VArray is a variable-size array that can only be transient. Unlike *standard VArrays* (instances of `ooVArrayT<element_type>`), temporary VArrays cannot be incorporated in a persistent object, either through inheritance or embedding as a data member; only standard VArrays can be made persistent.

Elements of a Temporary VArray

Elements of temporary VArrays can be of any type, including non-persistence-capable class types. For example, `ooTVArrayT<Point>` is a template class representing temporary VArrays whose elements are instances of the non-persistence-capable class `Point`.

NOTE Temporary VArrays cannot contain other VArrays, either directly or indirectly.

Unlike standard VArrays, elements of temporary VArrays can be handles, iterators, or, more generally, elements that contain memory pointers to other elements.

Like the elements of fixed C++ arrays, the *element_type* of a temporary VArray must have a default constructor (a constructor that can take no arguments).

Structure and Behavior

A temporary VArray is a compound object consisting of a reference to a vector of elements. The reference portion of the temporary 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.

You use public constructors to create temporary VArrays with 0 or more elements. An empty temporary VArray has no vector allocated for it until you add elements using the `resize` or `extend` member function. You use these member functions to grow or truncate the vector dynamically. Resizing a temporary VArray to 0 elements deallocates the vector.

You access each element by its position in the array. Elements are numbered starting with 0; the position number is the element's *index* or *subscript*. VArray operations verify that any specified indexes are valid based on the VArray's current size; the `elem` access function bypasses this subscript range checking.

Because of the way temporary VArrays are represented, you cannot access the first element of an array by dereferencing it; that is, the expression **myArray* does not access the first element of *myArray*. You must use member functions such as `operator[ ]`, `elem`, or `head` to access the first element of a temporary VArray.

Effect of 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 standard and temporary VArrays, which affects the permitted element types:

- When a standard VArray is relocated, its elements are *bit-wise* copied, which preserves the element data exactly, but invalidates any element data that consists of memory pointers to other (now relocated) 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 as appropriate.

Reference Summary

Creating	ooTVArrayT<element_type>
Assigning	operator=
Modifying	operator= extend resize set update
Finding Elements	operator[ ] elem head
Getting Information	size

Reference Index

operator=	Assignment operator; assigns the specified temporary VArray to this temporary VArray, automatically adjusting the size of this temporary VArray.
operator[ ]	Subscript operator; accesses the specified element of this temporary VArray.
resize	Extends or truncates this temporary VArray to the specified number of elements.
set	Sets the specified element of this VArray to be the indicated value.
size	Gets the current number of elements in this temporary VArray.
update	Returns `oocSuccess` because a temporary VArray can always be updated.

Constructors and Destructors

ooTVArrayT<*element_type*>

Default constructor that constructs a new temporary VArray.

```
ooTVArrayT<element_type> ();
```

Discussion Constructs an uninitialized temporary VArray whose size is 0. No vector is allocated until you add elements using the `resize` or `extend` member function.

ooTVArrayT<*element_type*>

Constructs a new temporary VArray of the specified size.

```
ooTVArrayT<element_type> (uint32 initSize);
```

Parameters *initSize*

Initial number of elements to allocate. If you specify 0, no vector is allocated until you add elements using the `resize` or `extend` member function.

ooTVArrayT<*element_type*>

Copy constructor that constructs a new copy of the specified temporary VArray.

```
ooTVArrayT<element_type> (ooTVArrayT<element_type> &array);
```

Parameters *array*

Temporary VArray of the same type as this temporary VArray.

Discussion Constructs a temporary VArray whose size is equal to the size of *array*, then performs an element-by-element copy from *array* into the newly created VArray. The constructor operation populates the new temporary VArray by using the *element_type* default constructor to create new, empty elements, and then using *element_type* assignment to assign each element of *array* to a corresponding new element.

Operators

operator=

Assignment operator; assigns the specified temporary VArray to this temporary VArray, automatically adjusting the size of this temporary VArray.

```
ooTVArrayT<element_type> &operator=(
    ooTVArrayT<element_type> &array);
```

Parameters *array*

Temporary VArray of the same type as this temporary VArray.

Returns This temporary VArray.

Discussion The assignment operation resizes this temporary VArray to be the same size as *array*, and then performs an element-by-element copy from *array* into this temporary VArray. The assignment operation populates this temporary VArray by using the *element_type* default constructor to create new, empty elements, and then using *element_type* assignment to assign each element of *array* to a corresponding new element.

operator[ ]

Subscript operator; accesses the specified element of this temporary VArray.

```
element_type &operator[] (uint32 index);
```

Parameters *index*

Index of the element to access. Specify 0 to access the first element.

Returns *index*'th element of this temporary VArray.

Discussion You can use the subscript operator to either get or change the specified element of this temporary VArray (that is, you can use the subscript operator on either the right or left side of an assignment operation).

Subscript boundaries are checked to ensure integrity. You can use the `elem` member function to bypass subscript range checking.

Member Functions

elem

Gets the specified element of this temporary VArray, without performing subscript boundary checking.

```
element_type &elem(uint32 index);
```

Parameters *index*

Index of the element to access. Specify 0 to access the first element.

Returns Reference to the *index*'th element of the temporary VArray.

Discussion You can use `elem` to either get or set the specified element of this temporary VArray (that is, you can use `elem` on either the right or left side of an assignment operation).

The `elem` member function bypasses subscript range checking. You should use `operator[ ]` if you want subscript boundaries to be checked.

extend

Allocates a new element at the end of this temporary VArray, and sets it to the specified value.

```
ooStatus extend(element_type &newValue);
```

Parameters *newValue*

Value to be assigned to the new element, passed by reference.

Returns `oocSuccess` if successful; otherwise `oocError`.

Discussion Extending a temporary 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.

head

Gets the first element of this temporary VArray.

```
element_type *head() const;
```

Returns Pointer to the first element of this temporary VArray. If this VArray contains no elements, a null pointer is returned.

resize

Extends or truncates this temporary VArray to the specified number of elements.

```
ooStatus resize(uint32 newSize);
```

Parameters *newSize*

Total number of elements that this temporary VArray is to have. Specify 0 to remove all the elements, freeing the storage allocated to the element vector.

Returns `oocSuccess` if successful; otherwise `oocError`.

Discussion 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 temporary VArray to the new size.

To keep the temporary VArray's elements contiguous, `resize` may relocate the vector portion of this VArray in virtual memory. If this happens, the elements of this VArray are copied *element-by-element* to the new location. Element-by-element copying 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 element data that consists of memory pointers to other (now relocated) elements, provided that the *element_type* default constructor and the *element_type* destructor manage the pointer linkage as appropriate. This kind of copying causes resizing a temporary VArray to be slower than resizing a standard VArray.

set

Sets the specified element of this VArray to be the indicated value.

```
ooStatus set(uint32 index, element_type &newValue);
```

Parameters *index*

Index of the element to access. Specify 0 to access the first element.

newValue

Value to be assigned to the specified element, passed by reference.

Returns `oocSuccess` if successful; otherwise `oocError`.

size

Gets the current number of elements in this temporary VArray.

```
uint32 size();
```

Returns Number of elements in this temporary VArray.

update

Returns `oocSuccess` because a temporary VArray can always be updated.

```
ooStatus update();
```

Returns `oocSuccess`

Discussion This member function is defined for consistency with other classes.

ooUtf8String Class

Inheritance: **ooVArrayT<ooChar> -> ooUtf8String**

The class `ooUtf8String` is a Java compatibility class that represents a *Unicode string*—a sequence of Unicode characters in UTF-8 encoding.

See:

- "Reference Summary" on page 704 for an overview of member functions
- "Reference Index" on page 704 for a list of member functions

To use the Java-compatibility classes, your application source must include the `javaBuiltins.h` header file.

About Unicode Strings

A Java string (of the Java class `java.lang.String`) is stored in an Objectivity / DB federated database as an embedded object of the class `ooUtf8String`. If your application interoperates with a Java application to access objects with a field that contains a string, you can define the corresponding data member of your C++ class to be of type `ooUtf8String`.

The class `ooUtf8String` represents a Unicode string as a variable-size array (VArray) whose elements are the component bytes of the string. This class simply allows 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. If your application renders a Unicode string, it is responsible for selecting the appropriate glyph for any non-ASCII character in the string.

Working With Unicode Strings

Because this class is derived from `ooVArrayT<ooChar>`, you work with a Unicode string just as you would work with a character VArray. Alternatively, you can work with an object of class `ooUtf8String` using member functions defined for the Objectivity / C++ variable-size string class `ooVString`. To do this, you cast the `ooUtf8String` object to a `const char *` and then pass the `const char *` to the `ooVString` constructor.

Related Classes

The persistence-capable class `oojString` represents a string element of a persistent array. It is a wrapper for a string of the `ooUtf8String` class.

A Java string array (of the Java type `String[]`) is stored in an Objectivity / DB federated database as an object reference to a persistent array object of the class `oojArrayOfObject`. Elements of the array are object references to instances of `oojString`.

Reference Summary

Creating	`ooUtf8String`
Assigning	`operator=`
Type Conversion	`operator const char *`

Reference Index

`ooUtf8String`	Constructs a new Unicode string.
`operator=`	Assignment operator; assigns the specified C++ string to this Unicode string.
`operator const char *`	Conversion operator that accesses this Unicode string as an object of type `const char *`.

Constructors

ooUtf8String

Constructs a new Unicode string.

```
1.  ooUtf8String();
2.  ooUtf8String(const char *p);
```

Parameters *p*

Existing C++ string from which to construct the new Unicode string.

Discussion Variant 1 is the default constructor. It constructs an empty Unicode string. Variant 2 constructs a new Unicode string containing a copy of the characters in the specified C++ string.

Operators

operator=

Assignment operator; assigns the specified C++ string to this Unicode string.

```
ooUtf8String &operator=(const char *p);
```

Parameters *p*

C++ string whose characters are to be assigned.

Returns This Unicode string.

Discussion The assignment operation resizes this Unicode string to be the same size as *p*, and then copies the characters of *p* into this Unicode string. Any characters already in this Unicode string are overwritten. If *p* is null, the effect is to delete the string.

operator const char *

Conversion operator that accesses this Unicode string as an object of type `const char *`.

```
operator const char *() const;
```

ooVArrayT<*element_type*> Class

Inheritance: **ooVArrayT<*element_type*>**

The non-persistence-capable template class ooVArrayT<*element_type*> represents a *standard variable-size array (VArray)* whose elements are of type *element_type*.

See:

- "Reference Summary" on page 710 for an overview of member functions
- "Reference Index" on page 711 for a list of member functions

(*ODMG*) The ooVArrayT<*element_type*> class is equivalent to the ODMG standard class d_Varray<*element_type*>, but without the base class d_Collection<*element_type*>, which is not implemented by Objectivity/C++.

For backward compatibility, you can use the macro-style name ooVArray(*element_type*) instead of the name ooVArrayT<*element_type*>.

About Standard VArrays

A standard VArray is a variable-size array that can be used transiently or saved persistently—for example, as an embedded data member of a persistent object. Because standard VArrays can be persistent, their element types are subject to restrictions. In contrast, *temporary VArrays* are only transient and can therefore be of any transient element type (see ooTVArrayT<element_type>).

Elements of a Standard VArray

Elements of standard VArrays can be of most data types permitted in a persistence-capable class definition. For example, given a non-persistence-capable class Point, you can use the template class ooVArrayT<Point> to define VArrays whose elements are instances of class

`Point`. However, if class `Point` is persistence-capable, you must use `ooVArrayT<ooRef(Point)>` to produce VArrays whose elements are object references to points.

NOTE Standard VArrays cannot contain other VArrays, either directly or indirectly.

Specifically, the *`element_type`* of a standard VArray can be any of the following:

- A primitive type
- An object-reference type for a persistence-capable class or structure
- An embedded-class type—specifically, a non-persistence-capable class or structure for which all of the following are true:
  - None of its base classes are virtual.
  - Each of its data members is of a primitive type, an object-reference type, or a valid embedded-class type.
  - None of its data members is of a prohibited element type.

The following element types are prohibited:

- VArrays
- Unions
- Bit fields
- Member pointers
- Persistence-capable structures and classes (however, object references to these are permitted)
- Handles and iterators for persistence-capable structures or classes

These restrictions apply recursively to standard VArray elements.

If you need to create VArrays of handles, iterators, or elements that contain memory pointers to other elements, you must use instances of the temporary VArray class `ooTVArrayT<`*`element_type`*`>`.

Like the elements of fixed C++ arrays, the *`element_type`* of a VArray must have a default constructor (a constructor that can take no arguments).

Structure and Behavior

A standard 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.

You use public constructors to create VArrays with 0 or more elements. An empty VArray has no vector allocated for it until you assign another VArray to it or add elements using the `resize` or `extend` member function. You use these member functions to grow or truncate the vector dynamically. 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*. VArray operations verify that any specified indexes are valid based on the VArray's current size; the `elem` access function bypasses this subscript range checking.

Because of the way VArrays are represented, you cannot access the first element of an array by dereferencing it; that is, the expression `*myArray` does not access the first element of `myArray`. You must use member functions such as `operator[ ]`, `elem`, or `head` to access the first element of a VArray.

Effect of 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 standard and temporary VArrays, which affects the permitted element types:

- When a standard VArray is relocated, its elements are *bit-wise* copied, which preserves the element data exactly, but invalidates any element data that consists of memory pointers to other (now relocated) 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 as appropriate.

Working With Persistent VArrays

A VArray is transient unless you incorporate it in a persistent object, typically by embedding the VArray as a data member of the persistent object or one of its base classes. A VArray can be stored persistently through multiple levels of inheritance and embedding (a member of a member of a base class, and so on).

EXAMPLE This example uses a VArray class as a data member type in a persistence-capable class (Polygon), resulting in a persistent VArray embedded in a persistent object.

```
//DDL file
struct Point{
   int32 xCoord;
   int32 yCoord;
};

class Polygon : public ooObj {
   …
   ooVArrayT<Point> vertices;  // Define VArray data member
   …
};
```

```
//Application code file
…
ooHandle(Polygon) polygonH;
polygonH = new (contH) Polygon();  // VArray embedded in Polygon
polygonH->vertices->resize(10);    // Set the Varray size
…
```

When you incorporate a 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.

Reference Summary

Creating	ooVArrayT<element_type>
Assigning	operator=

Modifying	operator= extend resize set update
Finding Elements	operator[ ] elem head
Getting Information	size
ODMG Interface	cardinality create_iterator insert_element is_empty remove_all replace_element_at retrieve_element_at upper_bound

Reference Index

ooVArrayT<element_type>	Copy constructor that constructs a new copy of the specified VArray.
operator=	Assignment operator; assigns the specified VArray to this VArray, automatically adjusting the size of this VArray.
operator[ ]	Subscript operator; accesses the specified element of this VArray.
remove_all	(ODMG) Removes all the elements from this VArray, changing its size to 0.
replace_element_at	(ODMG) Replaces the specified element of this VArray with the indicated value.
resize	Extends or truncates this VArray to the specified number of elements.
retrieve_element_at	(ODMG) Gets the specified element of this VArray.
set	Sets the specified element of this VArray to be the indicated value.
size	Gets the current number of elements in this VArray.
update	Explicitly opens this VArray for update.
upper_bound	(ODMG) Gets the current number of elements in this VArray; equivalent to the `size` member function.

Constructors and Destructors

ooVArrayT<*element_type*>

Default constructor that constructs a new VArray.

```
ooVArrayT<element_type>();
```

Discussion

Constructs an uninitialized VArray whose size is 0. No vector is allocated until you assign another VArray or add elements using the `resize` or `extend` member function.

ooVArrayT<*element_type*>

Constructs a new VArray of the specified size.

```
ooVArrayT<element_type>(uint32 initSize);
```

Parameters *initSize*

Initial number of elements to allocate. If you specify 0, no vector is allocated until you assign another VArray or add elements using the `resize` or `extend` member function.

ooVArrayT<*element_type*>

Copy constructor that constructs a new copy of the specified VArray.

```
ooVArrayT<element_type>(ooVArrayT<element_type> &array);
```

Parameters *array*

VArray of the same type as this VArray.

Discussion Constructs a VArray whose size is equal to the size of *array*, then performs an element-by-element copy from *array* into the newly created VArray. The constructor operation populates the new VArray by using the *element_type* default constructor to create new, empty elements, and then using *element_type* assignment to assign each element of *array* to a corresponding new element.

Operators

operator=

Assignment operator; assigns the specified VArray to this VArray, automatically adjusting the size of this VArray.

```
ooVArrayT<element_type> &operator=(
    ooVArrayT<element_type> &array);
```

Parameters *array*

VArray of the same type as this VArray. *array* may be either a persistent or transient instance of `ooVArrayT<element_type>`.

Returns This VArray.

Discussion The assignment operation resizes this VArray to be the same size as *array*, and then performs an element-by-element copy from *array* into this VArray. The assignment operation populates this VArray by using the *element_type* default constructor to create new, empty elements, and then using *element_type* assignment to assign each element of *array* to a corresponding new element.

If this VArray is persistent, it is implicitly opened for update, and the lock on the container is upgraded, if necessary. If *array* is persistent, it is implicitly opened for read.

operator[ ]

Subscript operator; accesses the specified element of this VArray.

```
element_type &operator[](uint32 index);
```

Parameters

index

Index of the element to access. Specify 0 to access the first element.

Returns

index'th element of this VArray.

Discussion

You can use the subscript operator to either get or change the specified element of this VArray (that is, you can use the subscript operator on either the right or left side of an assignment operation).

If this VArray is persistent, the subscript operator implicitly opens the VArray for read. However, you must call the `update` member function on the VArray before using the subscript operator to modify an element. (Alternatively, you can use the `set` member function to open the VArray for update and modify an element in a single operation.)

Subscript boundaries are checked to ensure integrity. You can use the `elem` member function to bypass subscript range checking.

See also

`elem`

Member Functions

cardinality

(*ODMG*) Gets the current number of elements in this VArray; equivalent to the `size` member function.

```
uint32 cardinality() const;
```

Returns

Number of elements in this VArray.

See also

`size`

create_iterator

(*ODMG*) Creates an iterator for finding the elements of this VArray.

```
d_Iterator<element_type> create_iterator() const;
```

Returns Iterator of the same type as the VArray elements.

Discussion The created iterator is initialized to point to the first element of this VArray.

elem

Accesses the specified element of this VArray, without performing subscript boundary checking.

```
element_type &elem(uint32 index);
```

Parameters *index*

Index of the element to access. Specify 0 to access the first element.

Returns *index*'th element of this VArray.

Discussion You can use `elem` to either get or set the specified element of this VArray (that is, you can use `elem` on either the right or left side of an assignment operation).

If this VArray is persistent, it is implicitly opened for read. You must call the `update` member function on the VArray before using `elem` to modify an element. (Alternatively, you can use the `set` member function to open the VArray for update and modify an element in a single operation.)

The `elem` member function bypasses subscript range checking. You should use `operator[ ]` if you want subscript boundaries to be checked.

See also `operator[ ]`

extend

Allocates a new element at the end of this VArray, and sets it to the specified value.

```
ooStatus extend(element_type &newValue);
```

Parameters *newValue*

Value to be assigned to the new element, passed by reference.

Returns `oocSuccess` if successful; otherwise `oocError`.

Discussion If this VArray is persistent, it is implicitly 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.

See also `resize`

head

Gets the first element of this VArray.

```
element_type *head();
```

Returns Pointer to the first element of this VArray. If the VArray contains no elements, a null pointer is returned.

Discussion If this VArray is persistent, it is implicitly opened for read.

insert_element

(*ODMG*) Allocates a new element at the end of this VArray, and sets it to the specified value; equivalent to the `extend` member function.

```
void insert_element(const element_type &element);
```

Parameters *element*

Value to be assigned to the new element, passed by reference.

Discussion If this VArray is persistent, it is implicitly opened for update, and the lock on the container is upgraded, if necessary. Extending a persistent VArray causes the entire VArray to be written to disk when the transaction commits.

Extending a VArray implicitly resizes it, which is an expensive operation. You should therefore use this operation as a convenient way to add a single element to a VArray. If you need to add a large number of elements in a single transaction, you should consider using `resize` to allocate all the elements in one operation.

See also `resize`

is_empty

(*ODMG*) Checks whether the size of this VArray is 0.

```
int is_empty() const;
```

Returns True (a nonzero integer) if this VArray has no elements; otherwise, false (the integer 0).

remove_all

(*ODMG*) Removes all the elements from this VArray, changing its size to 0.

```
void remove_all();
```

Discussion If this VArray is persistent, it is implicitly opened for update, and the lock on the container is upgraded, if necessary. This member function is equivalent to calling `resize` with a value of 0.

See also `resize`

replace_element_at

(*ODMG*) Replaces the specified element of this VArray with the indicated value.

```
1.  void replace_element_at(
      const element_type &newValue,
      uint32 index);
2.  void replace_element_at(
      const element_type &newValue,
      const Iterator<element_type> &iterator);
```

Parameters *newValue*

Value to be assigned to the specified element, passed by reference.

index

Index of the VArray element to replace. Specify 0 to access the first element.

iterator

Iterator indicating the element to replace.

Discussion If this VArray is persistent, it is implicitly opened for update, and the lock on the container is upgraded, if necessary.

resize

Extends or truncates this VArray to the specified number of elements.

```
ooStatus resize(uint32 newSize);
```

Parameters

newSize

Total number of elements that this VArray is to have. Specify 0 to remove all the elements, freeing the storage allocated to the element vector.

Returns

`oocSuccess` if successful; otherwise `oocError`.

Discussion

If this VArray is persistent, it is implicitly opened for update, and the lock on the container is upgraded, if necessary. Resizing a persistent VArray causes the entire VArray to be written to disk when the transaction commits.

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.

To keep the VArray elements contiguous, `resize` may relocate the vector portion of this VArray in virtual memory. If this happens, the elements of this VArray are *bit-wise* copied to the new location. Bit-wise copying preserves the element data exactly, which is efficient, but invalidates any element data that consists of memory pointers to other (now relocated) elements. You should use instances of the transient-only class `ooTVArrayT<`*element_type*`>` for temporary VArrays of elements that contain memory pointers to each other.

See also

`extend`
`insert_element`
`remove_all`

retrieve_element_at

(*ODMG*) Gets the specified element of this VArray.

```
const element_type &retrieve_element_at(uint32 index) const;
```

Parameters

index

Index of the VArray element to access. Specify 0 to access the first element.

Returns

index'th element of this VArray.

Discussion If this VArray is persistent, it is implicitly opened for read.

set

Sets the specified element of this VArray to be the indicated value.

```
ooStatus set(uint32 index, element_type &newValue);
```

Parameters *index*

Index of the VArray element to access. Specify 0 to access the first element.

newValue

Value to be assigned to the specified element, passed by reference.

Returns `oocSuccess` if successful; otherwise `oocError`.

Discussion If this VArray is persistent, it is implicitly opened for update, and the lock on the container is upgraded, if necessary.

size

Gets the current number of elements in this VArray.

```
uint32 size();
```

Returns Number of elements in this VArray.

See also `cardinality`
`upper_bound`

update

Explicitly opens this VArray for update.

```
ooStatus update();
```

Returns `oocSuccess` if successful; otherwise `oocError`.

Discussion This operation applies only to persistent VArrays; it has no effect on transient VArrays.

You must explicitly open a persistent VArray for update before you can modify any element using `elem` or `operator[ ]` on the left side of an assignment operation. (Alternatively, you can use the `set` member function to both open the VArray for update and modify an element in a single operation.)

Explicitly opening a VArray for update locks the enclosing container for update and causes the entire VArray to be written to disk when the transaction commits.

You should use `update` primarily for changing a large number of elements in a single transaction.

upper_bound

(*ODMG*) Gets the current number of elements in this VArray; equivalent to the `size` member function.

```
uint32 upper_bound() const;
```

Returns Number of elements in this VArray.

See also `size`

ooVString Class

Inheritance: **ooVArrayT<ooChar> -> ooVString**

The non-persistence-capable class `ooVString` represents a *variable-size string* based on a VArray of 8-bit characters.

See:

- "Reference Summary" on page 723 for an overview of member functions
- "Reference Index" on page 724 for a list of member functions

(*ODMG*) The `ooVString` class is equivalent to the ODMG standard class `d_String`. You can use the name `d_String` interchangeably with `ooVString`.

About Variable-Size Strings

A variable-size string is a character string of any length that can be stored in a persistent object. Although you can use variable-size strings anywhere in an application, their primary purpose is to serve as string attributes of persistence-capable classes (in place of C++ `char *` strings, which cannot be stored persistently). You can convert transparently between a variable-size string and a `const char *` string, enabling variable-size strings to be passed to functions as parameters of type `const char *` and vice versa.

EXAMPLE This example shows a C++ class `Loan` that was adapted to create the persistence-capable class `Loan` in the DDL file `Loan.ddl`. In the DDL file, the `dueDate` data member has been changed to an `ooVString` from a pointer to an array of characters.

```
// Original C++ header file Loan.h
class Loan {
   ...
   char *dueDate;                  // Due date of loan
```

```
   ...
};

Loan::Loan (char *date) {
   ...
   dueDate = malloc(strlen(date) + 1);
   strcpy(dueDate, date);          // Allocate memory
   ...
}
```

```
// DDL file Loan.ddl
class Loan : public ooObj {
   ...
   ooVString dueDate;              // Due date of loan
   ...
};

Loan::Loan (char *date) {
   ...
   dueDate = date;
   ...
}
```

Choosing Variable-Size Strings

Objectivity/C++ provides two kinds of strings that can be stored persistently—variable-size strings and optimized strings. You should choose variable-size strings if you cannot predict the lengths of the strings to be stored or if you know these lengths will vary widely. In contrast, if you know that the strings to be stored are generally less than *N* characters long, you should use an optimized string of class `ooString(`*N*`)` for greater efficiency; see “ooString(N) Class” on page 645.

Structure of Variable-Size Strings

A variable-size string is a VArray of character elements. 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; 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.

Working With Variable-Size Strings

Like instances of any other non-persistence-capable class, variable-size strings are not independently persistent. However, when a variable-size string is an attribute of a persistent object, it is saved in the federated database when the persistent object is saved.

You use public 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 can use the `resize` member function to grow or truncate the vector dynamically. Resizing a variable-size string to 0 elements deallocates the vector.

You get each 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.

Because of the way a variable-size string is represented, you cannot get the first character by dereferencing the string; that is, the expression *`*myVString`* does not get 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.

Reference Summary

Creating	`ooVString`
Assigning	`operator=`
Type Conversion	`operator const char *`

Modifying	`operator=`
	`operator+=`
	`resize`

Getting Characters	`operator[ ]` `head`
Getting Information	`length`

Testing	::operator== ::operator!= ::operator< ::operator<= ::operator> ::operator>=
ODMG Interface	::operator== ::operator!= ::operator< ::operator<= ::operator> ::operator>=

Reference Index

::operator<=	(ODMG) Less-than-or-equal-to operator; tests whether one string is less than or equal to another.
::operator>	(ODMG) Greater-than operator; tests whether one string is greater than another.
::operator>=	(ODMG) Greater-than-or-equal-to operator; tests whether one string is greater than or equal to another.
operator const char *	Conversion operator that accesses this variable-size string as an object of type `const char *`.

Constructors and Destructors

ooVString

Default constructor that constructs a new variable-size string whose size is 0.

```
ooVString();
```

Discussion

No vector is allocated until you assign a value or add elements using the `resize` member function.

ooVString

Constructs a new variable-size string containing a copy of the characters in the specified string.

```
1.  ooVString(const char *p);
2.  ooVString(const ooVString &s);
```

Parameters

p

Existing C++ string from which to construct the new variable-size string.

s

Existing variable-size string from which to construct the new variable-size string.

Discussion

If *p* is null or if the length of *s* is 0, an uninitialized string of size 0 is created. No vector is allocated until you assign a new value or add elements using the `resize` member function.

Operators

operator[ ]

Subscript operator; gets the specified character of this variable-size string.

```
char &operator[](const uint32 index) const;
```

Parameters *index*

Index of the character to get. Specify 0 to get the first character.

Returns *index*'th character of this variable-size string.

Discussion An error is reported if the index is not within the allocated size of the vector (including the terminating null character).

operator+=

Append-to operator; concatenates this variable-size string with the specified C++ string.

```
ooVString &operator+=(const char *p);
```

Parameters *p*

C++ string whose characters are to be concatenated.

Returns This variable-size string.

Discussion The concatenation operator adds the characters pointed to by *p* to the end of this variable-size string.

You can use this operator to concatenate another variable-size string to this one, because `operator const char *` automatically converts the string being concatenated to `const char *`.

operator=

Assignment operator; assigns a copy of the specified C++ string to this variable-size string.

```
ooVString &operator=(const char *p);
```

Parameters *p*

C++ string whose characters are to be assigned.

Returns This variable-size string.

Discussion

The assignment operation resizes this variable-size string to be the same size as *p*, and then copies the characters of *p* into this variable-size string. Any characters already in this variable-size string are overwritten. If *p* is null, the vector is deallocated—in effect, deleting the string.

You can use this operator to assign another variable-size string to this one, because `operator const char *` automatically converts the string being assigned to `const char *`.

::operator== global function

(*ODMG*) Equality operator; tests whether the specified strings match.

```
1.  int ::operator==(
      const ooVString &left,
      const ooVString &right);
2.  int ::operator==(
      const ooVString &left,
      const char *right);
3.  int ::operator==(
      const char *left,
      const ooVString &right);
```

Returns

1, if every character of one string matches the corresponding character of the other; 0, if the two strings are lexicographically unequal.

Discussion

You can compare two variable-size strings (variant 1) or a C++ string and a variable-size string (variants 2 and 3).

::operator!= global function

(*ODMG*) Inequality operator; tests whether the specified strings are different.

```
1.  int ::operator!=(
      const ooVString &left,
      const ooVString &right);
2.  int ::operator!=(
      const ooVString &left,
      const char *right);
3.  int ::operator!=(
      const char *left,
      const ooVString &right);
```

Returns

1, if any character of one string differs from the corresponding character of the other; 0, if strings are lexicographically equal to each other.

Discussion You can compare two variable-size strings (variant 1) or a C++ string and a variable-size string (variants 2 and 3).

::operator< global function

(*ODMG*) Less-than operator; tests whether one string is less than another.

```
1.  int ::operator<(
      const ooVString &left,
      const ooVString &right);
2.  int ::operator<(
      const ooVString &left,
      const char *right);
3.  int ::operator<(
      const char *left,
      const ooVString &right);
```

Returns 1, if *left* is lexicographically less than *right*, or if *left* contains no characters or is a null pointer; otherwise, returns 0.

Discussion You can compare two variable-size strings (variant 1) or a C++ string and a variable-size string (variants 2 and 3).

::operator<= global function

(*ODMG*) Less-than-or-equal-to operator; tests whether one string is less than or equal to another.

```
1.  int ::operator<=(
      const ooVString &left,
      const ooVString &right);
2.  int ::operator<=(
      const ooVString &left,
      const char *right);
3.  int ::operator<=(
      const char *left,
      const ooVString &right);
```

Returns 1, if *left* is lexicographically less than or equal to *right*, or if *left* contains no characters or is a null pointer; otherwise, returns 0.

Discussion You can compare two variable-size strings (variant 1) or a C++ string and a variable-size string (variants 2 and 3).

::operator> global function

(*ODMG*) Greater-than operator; tests whether one string is greater than another.

```
1.   int ::operator>(
       const ooVString &left,
       const ooVString &right);
2.   int ::operator>(
       const ooVString &left,
       const char *right);
3.   int ::operator>(
       const char *left,
       const ooVString &right);
```

Returns 1, if *left* is lexicographically greater than *right*, or if *right* contains no characters or is a null pointer; otherwise, returns 0.

Discussion You can compare two variable-size strings (variant 1) or a C++ string and a variable-size string (variants 2 and 3).

::operator>= global function

(*ODMG*) Greater-than-or-equal-to operator; tests whether one string is greater than or equal to another.

```
1.   int ::operator>=(
       const ooVString &left,
       const ooVString &right);
2.   int ::operator>=(
       const ooVString &left,
       const char *right);
3.   int ::operator>=(
       const char *left,
       const ooVString &right);
```

Returns 1, if *left* is lexicographically greater than or equal to *right*, or if *right* contains no characters or is a null pointer; otherwise, returns 0.

Discussion You can compare two variable-size strings (variant 1) or a C++ string and a variable-size string (variants 2 and 3).

operator const char *

Conversion operator that accesses this variable-size string as an object of type `const char *`.

```
operator const char *() const;
```

Discussion This operator results in a null pointer if the vector is not allocated.

Member Functions

head

Gets a pointer to the first character of this variable-size string.

```
char *head() const;
```

Returns Pointer to the first character of this variable-size string. If the string contains no characters, returns a null pointer.

length

Gets the number of characters in this variable-size string.

```
uint32 length() const;
```

Returns Integer number of characters in this string preceding the first null terminating character as computed by `strlen`. If the string contains no characters, returns 0.

Discussion The actual number of bytes allocated is at least *`length()`* `+ 1` because an extra byte is reserved for the null terminating character. (If the string contains an embedded null character, the bytes beyond that null character are not included in the returned length.) You can use the inherited `size` member function to determine the actual allocated size.

resize

Extends or truncates this variable-size string to the specified number of characters.

```
ooStatus resize(const uint32 newLength);
```

Parameters *newLength*

Number of characters this variable-size string is to have. Specify 0 to remove all the characters, freeing the storage allocated to the variable-size string's vector.

Returns `oocSuccess` if successful; otherwise `oocError`.

Discussion The actual number of bytes allocated is *newLength* + 1 because an extra byte is automatically reserved for the null terminating character. However, this member function does not actually store the required null character; the caller is responsible for updating the contents of the string.

Example The following code fragment resizes a variable-size string and provide it with the null terminating character:

```
ooVString s;
s.resize(n);
s[n] = '\0';
```

Topic Index

This index lists topics that are discussed in this book. For a list of classes, see "Classes Index" on page 763. For a list of functions, including member functions, see "Functions and Macros Index" on page 773. For a list of non-class types and constants, see "Types and Constants Index" on page 787.

Symbols

Numerics

A

D

E

F

G

H

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

Classes Index

This index contains an alphabetical list of classes, with member functions listed under each class. For a list of topics that are discussed in this book, see “Topic Index” on page 733. For an alphabetical list of all functions, including member functions, see “Functions and Macros Index” on page 773. For a list of non-class types and constants, see “Types and Constants Index” on page 787.

A

D

O

Functions and Macros Index

This index contains an alphabetical list of all global functions, member functions, and global macros. For a list of topics that are discussed in this book, see “Topic Index” on page 733. For an alphabetical list of classes, with member functions listed under each class, see “Classes Index” on page 763. For a list of non-class types and constants, see “Types and Constants Index” on page 787

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

Y

Types and Constants Index

This index lists non-class types and constants. For a list of topics that are discussed in this book, see "Topic Index" on page 733. For an alphabetical list of classes, see "Classes Index" on page 763. For a list of functions, including member functions, see "Functions and Macros Index" on page 773.

R

T

U

W