



Objectivity/DB Quick Start

Release 5.0

Objectivity/DB Quick Start

Release 5.0, July 29, 1999

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 1998 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/C++, Objectivity/C++ Data Definition Language, Objectivity/DDL, Objectivity/C++ Standard Template Library, Objectivity/C++ STL, Objectivity for Java, Objectivity/Smalltalk, Objectivity/SQL++, Objectivity/SQL++ ODBC Driver, and Objectivity/ODBC 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	5
Organization	5
Conventions and Abbreviations	6
Getting Help	7
Chapter 1 Objectivity/DB Overview	9
About Objectivity/DB	9
Typical Project Life Cycle	10
Modeling	10
Prototyping	10
Optimizing	11
Language Choices	11
C++	11
Java	12
Advantages of the Objectivity/DB Approach	12
Chapter 2 Objectivity/DB Basics	13
Objectivity/DB Objects	13
Basic Object	14
Container	14
Database	14
Federated Database	14
Persistent Object Creation and Retrieval	15
Object Creation and Schema Definition	16
Storage Directives	18
Object Retrieval	18
Transaction Management	19

	Transaction Boundaries	19
	Transaction Types	20
Chapter 3	The Helloworld Application	21
	Basic Setup on All Platforms	21
	Setting Environment Variables	22
	Starting the Lock Server	22
	Creating the Federated Database	23
	C++ and Java Interoperability	24
	C++ Application Setup	24
	Java Setup	26
	Running the Application	27
	Debugging and Locks	27
Chapter 4	Exploring the Database	29
	Examining the Database	29
	oobrowse and ootoolmgr	29
	ooRunStatus	30
	Structuring for Optimal Performance	30
	Data structures	31
	Storage Strategies	32
	Class Strategies	32
	Associations	33
	Tips for Good Design	34
	Other Examples Available	34
Chapter 5	Testing Your Knowledge	35

About This Book

This book, *Objectivity/DB Quick Start*, presents a quick Objectivity/DB tutorial for programmers who may have little or no background in object-oriented database application programming. It demonstrates the basic concepts of making objects persistent in both Java and C++. The tutorial will familiarize the user with the concepts of object persistence and its idioms. A self-quiz follows the tutorial to help evaluate the reader's understanding.

Objectivity/DB provides tools and programming interfaces to help perform administration tasks. The tasks and tools are similar on all platforms; minor differences in usage, behavior, and naming that exist between different platforms are noted. The tools are described in the Objectivity/DB Administration book; programming interfaces are described in the Objectivity/C++, Objectivity for Java, and Objectivity/Smalltalk documentation.

Organization

- Chapter 1 introduces object-oriented programming and features, techniques, and advantages of Objectivity/DB.
- Chapter 2 describes the structure and methods of Objectivity/DB.
- Chapter 3 presents the `helloworld` sample application.
- Chapter 4 discusses how to further explore and analyze the database.
- Chapter 5 is a quiz to test the reader's understanding of the concepts presented in the Tutorial.

Conventions and Abbreviations

Navigation

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

Typographical Conventions

<code>oobackup</code>	Command, literal parameter, code sample, filename, pathname, output on your screen, or Objectivity-defined identifier
<code>installDir</code>	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
<code>lock server</code>	New term, book title, or emphasized word

Abbreviations

<i>(administration)</i>	Feature intended for database administration tasks
<i>(FTO)</i>	Objectivity/DB Fault Tolerant Option feature
<i>(DRO)</i>	Objectivity/DB Data Replication Option feature
<i>(ODMG)</i>	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 parameters. 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. Call Objectivity Customer Support to get 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 AM and 6 PM 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@objy.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 and 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/DB Overview

This chapter introduces the basic concepts of object-oriented programming and some of the techniques, special features, and unique advantages of Objectivity/DB. It briefly describes:

- Objectivity/DB
- A typical project approach
- Language choices and flexibility
- Some advantages of Objectivity/DB

For more information about specific platforms and environments, please refer to the customer support options listed in “About This Book”, and visit the Objectivity, Inc. InfoCenter web site.

About Objectivity/DB

In contrast to relational databases, where a large parser is used to access and manipulate data, Objectivity/DB uses more direct techniques to access persistent objects.

Objectivity/DB uses object hierarchies, map and hash constructs, and Standard Template Library classes to store and retrieve data. These constructs, in conjunction with a robust language binding, provide a nearly infinite combination of persistent storage techniques. These techniques can be combined to solve the most demanding persistent storage problems in an efficient and flexible way.

In Objectivity/DB, objects are accessed and retrieved through the use of an Object Identifier (OID). This enables the persistent storage of objects outside of computer memory, and allows Objectivity/DB to locate and manage objects with flexibility and safety. In C++, Objectivity/DB supports ODMG-style references to ensure that rogue pointers cannot corrupt persistent data.

The Objectivity/DB client-side cache maintains coherence with persistent data while enabling almost in-memory access speeds.

In common with relational databases, Objectivity/DB supports the concept of transactions. Transactions allow multiple users access to objects while guaranteeing that “ACID” properties are not violated. These properties, *Atomicity*, *Consistency*, *Isolation*, and *Durability*, are defined as follows:

<i>Atomicity</i>	Each transaction is executed completely or not at all.
<i>Consistency</i>	Transactions are serializable, allowing the database to remain in a consistent state.
<i>Isolation</i>	Operations within one transaction are not affected by other transactions. This is accomplished through locking mechanisms.
<i>Durability</i>	Committed data persists despite machine failures.

Typical Project Life Cycle

Although Objectivity/DB is often used in a distributed environment, development is typically done on a single computer or server. Development can be done independently of language bindings, compilation, and other configuration management issues.

Modeling

An Objectivity/DB project typically involves modeling a complex problem to test performance, code complexity, and scalability. Object modeling with Objectivity/DB is identical to modeling data in non-database applications, with the advantage that there is a natural mapping of the data model to both the application and the database. This inherent mapping of data on disk to that used in the application significantly improves performance, because the data does not have to be constructed from simpler parts. Using object modeling, it is easier to predict behavior and to show areas of potential code complexity.

Prototyping

Objectivity/DB projects often use prototypes to successfully emulate critical code sections and object access patterns. A prototype usually starts with an existing model or design. This can then be formulated to test critical code functionality.

The critical prototype is used throughout the development process to test performance and coding techniques in an isolated fashion.

Optimizing

When the project is assembled and working, other techniques can be used to maintain the flexibility and usefulness of the application:

- Schema migration is used to migrate an existing object model in order to keep current with changing external requirements.
- Object versioning and other built-in functionality provide ongoing versatility in design, enabling you to create systems that vary widely in scope and depth of function.
- Load balancing is achieved by distributing databases throughout the network or onto several clustered machines.
- Page size and cache size can be adjusted so that excess paging and network or memory traffic are optimized.

Language Choices

Objectivity/DB allows you to choose the language which is appropriate for your project.

C++

Using Objectivity/DB for a project being developed in C++ is extremely straightforward. There are no mapping layers, and the classes which are defined in the application are used directly to store instances of those classes on the database. Because the language and schema of the database are so closely tied, the development of your application is tied to the development of the database.

The Objectivity C++ language binding uses a preprocessor which recognizes a Data Definition Language (DDL). DDL is basically C++ with a few syntactic extensions for things like relationships. A DDL file looks very much like a C++ header file. Since all persistence-capable C++ classes must inherit from the ooObj class, ooObj must also be present in the .ddl files. The DDL preprocessor generates not only the database schema but also header (.h) and code (.cpp/.C) files which make the schema available to the C++ application.

Objectivity/DB allows you to easily change your schema as the needs of your applications and users change. This feature, known as schema evolution, allows you to change classes, their contents, and their inheritance. Objectivity/DB schema are ideal for data models of high complexity.

Java

Objectivity for Java allows applications being developed in the Java language to store objects with very little extra coding. The schema of the database is managed from within the Java application and supports the full flexibility of Java; new classes can be created and modified during the execution of an application. Objectivity for Java is an ODMG 2.0-compliant binding, with extensions which allow the full power of Objectivity/DB to be brought to bear. Objects are stored in the database either using inheritance, similar to C++, or through the implementation of an “interface”.

Advantages of the Objectivity/DB Approach

Objectivity/DB is a proven and powerful technology enabling complex applications to be flexible and scalable. It provides tools and interfaces for producing applications that can keep up with the most demanding environments. Databases and users can be distributed over many machines because of the “one logical view from any client” architecture.

The Objectivity/DB distributed architecture enables the same code to be used on platforms large and small. This enables a small customer to buy one computer to start with and later add more computers as requirements increase. A large customer can expand to a different location, add new computers with Objectivity/DB replication, and not have to change a single line of source code.

No matter which language you use, or even if you mix them, the integration of the database with your application means easier development, more robust deployment, and the cost-savings of not requiring a cadre of trained DBAs.

Objectivity/DB Basics

This chapter presents the basic components you will be working with in Objectivity/DB. It discusses some of the techniques for working with objects in C++ or Java, and some guidelines and methods for understanding multi-user transactions.

After you have read this chapter, the `helloworld` sample application will guide you through the basic steps in Objectivity/DB creation, retrieval, and storage of objects. When you have run `helloworld`, the Exploring the Database chapter helps you take a closer look at the concepts and features of the application.

This chapter describes:

- Objectivity/DB objects
- Persistent object creation and retrieval
- Transaction management

Objectivity/DB Objects

This section introduces the basic objects of Objectivity/DB.

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

Applications do not work with Objectivity/DB objects directly; instead they work with memory representations of objects, which must be retrieved from and written back to a federated database. To ensure that data maintained by Objectivity/DB objects remain consistent while being used by competing applications, Objectivity/DB uses a system of permissions, called *read locks* and *write locks*, to control access to the objects. Locks are administered by a lock server process, which can run on any machine in the network. Before an operation can

be performed on an Objectivity/DB object, an application must obtain access rights to the object from the lock server.

Basic Object

A *basic object* is the fundamental unit stored by Objectivity/DB. An object whose class is defined by your application is normally represented as a basic object. Each basic object is stored within a container. Objects are typically accessed using “smart pointers” called *handles*.

Object Identifiers

To identify a persistent basic object, Objectivity/DB uses an *object identifier* (OID). An OID identifies the database and container to which a basic object belongs, as well as other storage information. OIDs allow Objectivity/DB to locate and manage basic objects with more flexibility and safety than direct memory access.

Container

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

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

Each container is stored within a database.

Database

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

Federated Database

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

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

Files

Objectivity/DB stores all database information in normal system files. There are no specialized drivers required. In addition to database files, you will also find a boot file and journal files.

Every federated database has a *boot file*. Database applications and tools refer to a federated database using its boot filename, sometimes called the federated database's *system name*. A boot file is created automatically when you create a new federated database and contains entries specifying the various federated database attributes.

Whenever a transaction starts, it records update information in one or more *journal files*, which are used to return the federated database to its previously committed state if the transaction is aborted or terminated abnormally. Journal files enable Objectivity/DB to roll back changes made by incomplete transactions. Journal files are written in a federated database's *journal directory*.

Persistent Object Creation and Retrieval

This section discusses creation, storage, and retrieval of persistent objects in Objectivity/DB.

Persistent objects are accessed through the use of object references inside of a transaction. All persistent objects must inherit from or partner with a persistence-capable object base class, or, in Java, a persistent interface. By doing this, Objectivity/DB-supplied methods and persistent constructs are made available to the program.

When making an object persistent in Objectivity/DB, the code:

- Includes any necessary declarations.
- Initializes Objectivity/DB.
- Defines a handle class (C++) or reference (Java) for a persistent object.
- Starts a transaction.
- Opens or creates a database and container.
- Creates an object using `new` with a handle, or reference, to another existing object, container, or database to specify the storage location.
- Commits the transaction.

Transaction details are covered in the “Transaction Management” section and are mentioned here for completeness.

Object Creation and Schema Definition

When creating a persistent object, its class is defined in Objectivity/DB. Defining schema in Objectivity/C++ is more complex than in Objectivity for Java. Objectivity/C++ uses a definition language while Objectivity for Java uses dynamic schema loading to load schema information into the federated database. By following certain guidelines, it is possible to inter-operate between the two languages, so C++ can manipulate objects created in Java and Java can manipulate objects created in C++, although there are restrictions.

Objectivity/C++

When creating a new object in C++, the `new` operator is used as for heap-based objects, except that an additional parameter must be supplied, called a “storage directive”, and the result of `new` must be placed in an `ooHandle`, rather than in a pointer. The storage directive is simply a handle or a reference to another pre-existing persistent object like a database, container, or another basic object. Its use is explained in the section on “Storage Directives”. The `ooHandle` is a smart pointer which can be used like a C++ pointer, except it is used to access persistent objects. The `ooHandle` object itself contains information about the object identifier, or its location in the Objectivity/DB cache, depending on how it was used most recently. As with pointers in C++, `ooHandle` can be re-used to point to other objects. Since all persistent objects can be retrieved from the database at a later time, there is no such thing as a “dangling reference” problem in Objectivity/DB. An `ooHandle` can be re-assigned to point to another new object without fear of any memory leaks. Typically, after a persistent object is created it is added to another data structure, such as a lookup table or a collection object, so you can quickly navigate to the object later.

ooRef versus ooHandle: In Objectivity/C++, an `ooRef` is a light-weight `ooHandle` used to reference Objectivity/DB persistent objects. While an `ooHandle` contains information about an object’s state, such as where it is located in the cache, an `ooRef` is only aware of the OID of the object it points to. In general, `ooRef` and `ooHandle` are interchangeable except in the following circumstances:

- Only an `ooRef` can be used as a data member of another persistent object. It makes no sense to store an `ooHandle` inside a persistent object.
- Only an `ooHandle` can be used to store the return value of `new` when creating a persistent object.

in general, an `ooHandle` should be used in application code when multiple operations need to be performed on the same object. De-referencing an `ooHandle` is faster than de-referencing an `ooRef`, because an `ooHandle` knows the general location of an object in the cache. You can think of an `ooHandle` as optimized for speed, while an `ooRef` is optimized for size.

In Objectivity/C++, a schema is defined in the Database Definition Language and stored in a file with the extension `.ddl`. The language is basically C++ with a few language extensions to support Objectivity/DB-specific features. Typically, already existing C++ class definitions are taken from the header files and copied into one or more `.ddl` files, while ensuring that each class inherits directly or indirectly from the Objectivity/DB basic object class known as `ooObj`. Like C++ header files, one or many classes can be defined in a single `.ddl` file, and multiple `.ddl` files can be used for the schema. There are certain rules about what can be a data member of a persistence-capable class which are explained in more detail in the Objectivity/C++ Data Definition Language book.

When the `schema.ddl` files are run through the preprocessor `ooddlx`, the schema is loaded into the federated database. The preprocessor also generates some Objectivity/C++ files, including a `schema.h` file, a `schema_ref.h` file, and a `schema_ddl.cpp` file. The first header file contains class definitions which are included from the user's own source code modules. The other two files define some smart pointer classes which are used to manipulate persistent objects from the application. The last file must be added to the list of source code modules for a complete build. Finally, the application must link to some Objectivity/DB libraries.

Objectivity for Java

The major difference between persistent objects in Java and C++ is that the `new` operator is not overloaded in Java. After the object is created with `new`, it is made persistent by using one of the three techniques below:

- Using the `cluster()` method after creating it with `new`.
- Giving the object an ODMG-style named root.
- Adding the object to a persistent collection, or a relationship of another persistent object.

In Java, there is no such thing as an `ooHandle` or an `ooRef`. The reference in the Java language works in a way which is similar to the Objectivity/C++ `ooRef`, and can be used to access persistent objects the same way as for transient objects. The `ooId` class is used to obtain the OID of a persistent object.

In Objectivity for Java, persistence-capable classes need to inherit from the persistence-capable class `ooObj` or use the persistent interface, `Persistent`. As

long as the `CLASSPATH` environment variable is set properly, and the proper classes are imported at the top of each `.java` file, then it is possible to build an Objectivity for Java application with any standard JDK-compatible development environment.

Storage Directives

Storage directives and clustering strategies play an important role in database design, and have a high impact on performance. Objects stored close to one another are retrieved together. Storage strategies allow scalability and superior performance when matched intelligently with the problem domain.

All objects in the federated database have a unique OID. An OID is assigned to an object when it is created, and is used by Objectivity/DB to locate and access objects directly. Two other units of storage in Objectivity/DB, `ooContObj` and `ooDBObj`, are identified using an OID. They derive from `ooObj`. A reference to an `ooObj` could refer to anything identified by the OID. Therefore, it could also refer to an `ooContObj` or an `ooDBObj`. A cluster hint can be a reference to any of these kinds of objects. Because a basic object has a page and slot location, a basic object reference is more specific than an `ooDBObj` reference when used as a storage directive. In other words, when you supply a basic object as a storage directive, Objectivity/DB attempts to place the new object as close to the original object as possible. If only a container reference is supplied, Objectivity/DB can place the new object anywhere in the container. If a database reference is supplied, Objectivity/DB will place the new object in the default container of the specified database.

In Objectivity for Java, the `cluster` method is used to make a new object persistent.

For more information about good design and clustering strategies for Objectivity/DB applications, contact your systems engineer.

Object Retrieval

Objectivity/DB object retrieval is performed in all languages using an iterator. All iterators have a scan method to narrow the scope of returned objects. Iterators hide the complexity of the “one logical view” of objects anywhere in the network. Transaction details are explained in the “Transaction Management” section.

The general process for retrieving objects is to initialize an iterator (which can contain expressions to limit values), iterate through the returned objects, and commit the transaction.

The power of object-oriented databases becomes apparent when the use of object access patterns matches the problem domain. In general, performance-based object access with such constructs as hash tables (`ooMap`) and trees (indexes) can add great value to a problem domain. With good object design or discovery

techniques, the balance between language constructs and object access patterns adds performance focus to object processing and the problem domain.

Transaction Management

Transactions allow more than one application to access and manipulate persistent objects at the same time. The transaction can be considered a unit of work. A unit of work can be as simple as referencing an object, or it can be a complex interaction with several different levels of work, including threading.

Transaction Boundaries

Determining transaction boundaries can be confusing and elusive to the traditional programmer. In mature problem domains, transactions are often well defined. For other applications, Universal Modeling Language (UML), event modeling, or other object modeling tools can be used to determine rough transaction boundaries. Design is useful for getting a good picture of the problem, although there is nothing like coding to find out where the transaction boundaries really are.

For programmers who are new to object databases, Objectivity/DB provides some suggested coding guidelines to make the transaction discovery process easier. These guidelines are intended to help with quick prototyping of critical code sections or small test cases, and are summarized as follows:

- Define public static procedures/functions for all of the obvious units of work, such as creating, updating, or looking up chunks of data. These can be called from within class methods as place holders and can be used easily in a wide variety of other places. Typically, each static procedure/function might represent a transaction. It might include all the necessary things to start, commit, or abort the transaction. In addition to the tutorial, there are several complex code examples and transaction class (`d_session`) examples in the InfoCenter web site.
- Avoid the use of complex inheritance and other constructs until the transaction model is understood. With so many performance and coding options available in Objectivity/DB, using complex constructs may limit tuning and coding options. Keep your options open using good modeling techniques or the coding technique mentioned above. If an interface layer is needed, it can be assembled from a well-defined and running prototype.
- Use feedback and timing tools like `ooRunStatus()`, `ooBrowse`, `ootoolmgr`, and classes like the `d_session` to obtain statistics on a particular code section or critical prototype. This will allow code changes to be controlled and assessed in a knowledgeable and performance-driven way.

For the actual guidelines, code examples, and more information, please refer to the Objectivity InfoCenter web site.

Transaction Types

Transactions in Objectivity/DB involve setting the storage object open mode. The federated database, databases, and containers all have open modes that can be set to not open (`oocNoOpen`), read (`oocRead`), and update (`oocUpdate`).

A transaction (`ooTrans` in C++) and a session (`Session` in Java) have open modes as well. Transaction modes are set in the transaction start methods. Refer to the example code or the documentation for exact language syntax and default open modes.

Many Readers One Writer (MROW) is designed for high concurrency applications. It allows many readers and one writer to share a consistent view of the objects and object cache. There are many different types of locking and transaction scenarios. More information on MROW and the different modes can be found in the documentation.

Transaction status can be checked with the `oolockmon` command. Any cleanup of locks can be checked with the `oocleanup` command. Please refer to the administration section of this guide or the documentation for more information.

The C++ `d_session` class combines both storage open modes and transaction modes into a single class. The `Session` class in Java also combines storage modes and transaction modes. The basic pattern of this class is considered a singleton or facade pattern and is used to keep code clean by reusing object references. The `d_session` class is included in this tutorial for reference. Freely view the `d_session` C++ source code and documentation for its varied storage and transaction strategies.

The Helloworld Application

This chapter explains how to create and run the `helloworld` sample application. The `helloworld` application demonstrates the basic steps it takes to make objects persistent in Objectivity/DB. It is used to store and retrieve greetings and demonstrates how to create a persistent object, store it on disk, and retrieve it. The `helloworld` application uses a persistent class `pGreeting`. This class definition contains a simple string as well as class methods and supporting code.

The sample application contains both C++ and Java examples. The Java example contains both a Visual Café directory and a UNIX directory. The C++ example contains a directory for Microsoft Windows and UNIX. It also explains some code that you may find different from traditional C++. There are several code samples in separate subdirectories. For more information, see the directory listing.

Review all pertinent documentation. This will help with locating detailed information on a subject of interest. The Java documentation is located in the `.doc` subdirectory of the Objectivity/DB installation. The C++ documentation is available in print.

After you have run the application, the next chapter, *Exploring the Database*, provides an in-depth look at Objectivity/DB structure, features, and techniques introduced in `helloworld`.

Basic Setup on All Platforms

The basic setup provides the necessary working environment for Objectivity/DB. The steps to create the federated database are the same for all platforms. Once you have set up the federated database, follow the application setup steps for your platform.

This section describes:

- Setting environment variables.
- Starting the lock server.

- Creating a federated database using the `oonewfd` tool.
- C++ and Java Interoperability
- C++ application setup
- Java setup

Setting Environment Variables

Several of the command line tools use the `OO_FD_BOOT` environment variable to obtain the boot file name when one is not supplied from the command line, including `oonewfd`.

`OO_FD_BOOT` specifies the default Objectivity/DB boot file location for tools. With this environment variable set to the absolute pathname to the boot file, most tools do not require a boot file to be specified.

The `path` environment variable will need to contain an Objectivity/DB binaries location such as `install_dir/platform/bin`. This will enable the command line tools to be accessed without the full pathname to their location.

To set up the browser (known as `ooToolmgr` on UNIX), two environment variables are needed. Both of these environment variables are needed to point into the Objectivity/DB `install_dir/etc` directory. This directory contains the bit maps and other resources necessary for the GUI browser to operate. The variables are:

```

XFILESEARCHPATH=$(install_dir)/$(arch)/etc/app-defaults/%N
XBMLANGPATH=$(install_dir)/$(arch)/etc/bitmaps/%N/%B

```

The `install_dir` and the `arch` (architecture) are platform specific. Typically, on a Sun Unix platform running Solaris, the install directory and the architecture would look similar to `/objy51/solaris/etc`, where `/objy51` is the install directory and `solaris` is architecture. Please refer to the UNIX *Installation and Platform Notes* for more information.

There are tuning environment variables that can be set to view runtime statistics. Please refer to the tuning section of the manuals or `OO_RUN_STATUS` environment variable for more information.

Starting the Lock Server

The lock server must be running and various environment variables must be set —such as `path` and `ld_library_path`— before the federated database is created.

1. To make sure the lock server is running, select `oocheckls computer_name` from a console window.

2. On Windows:

Verify the status and start the lock server by double-clicking the control panel icon, or type `oolockserver` from an operating system window.

On UNIX:

Type `oolockserver <enter>`

Use `oolockserver -help` for a full listing of recovery options and various other distributed parameters. Check documentation for recovery options.

WARNING The lock server creates journal files for update transactions. Once a federated database has been created, the lock server must be started by a user with permission to write to the specific journal directory, specified when the federated database was created. Follow the installation guide and any release notes for your specific platform.

Creating the Federated Database

To create a federated database, you invoke `oonewfd` with options specifying:

- The federated database identifier
- The federated database file host and file path
- The journal directory host and path
- The lock server host
- The page size
- The boot filename

You must specify the federated database filepath, the lock server host, and a boot file name. The other attributes have default values.

```
oonewfd
  [-fdfilehost fdFileHost] -fdfilepath fdFilePath
  -lockserverhost lockServerHost
  [[-jnldirhost jnlDirHost] -jnldirpath jnlDirPath]
  [-fdnumber fdId]
  [-pagesize pageSize]
  [-bootonly]
  [-standalone]
  [-notitle]
  [-quiet]
  [-help]
  [bootFilePath]
```

Options

`-fdfilehost fdFileHost`

Host where the federated database file is to be located. The default host is the current host.

`-lockserverhost lockServerHost`

Host where the lock server servicing the new federated database is located.

For all other options, refer to the Objectivity/DB Administration book.

C++ and Java Interoperability

The `helloworld` application contains both C++ and Java examples, located in separate directories, which create a persistent `pGreeting` object. The federated database from one of the examples can be reused by the other samples and extensions. As long as the sample application opens the proper boot file, it can access either the C++ or Java based federated database.

Objectivity/DB applications can access remote databases on multiple platforms. For more information, please refer to the Objectivity Technical Support web site and the Objectivity/DB Administration book.

NOTE **C++:** The Windows example comes complete with a workspace. The UNIX example comes complete with a makefile. These will help speed the compilation of the products. All UNIX example makefiles use a regular `make` command. Any other `make`, or similar, commands may require modification to the supplied makefile. The UNIX `.cfg` file must be modified.

Java: The Java example includes a Visual Café workspace and all necessary class files. A UNIX makefile is supplied and will need to be adjusted with local references to Objectivity/DB and supporting Java files. This may be as simple as adjusting the class path and local hostnames.

C++ Application Setup

Each of the platforms requires some modification of the development environment or makefile for the examples to work.

If makefile or C++ problems do occur, make a backup and try to simplify the makefile or problem example to isolate the cause.

Windows

Caution: Make sure the lock server is running. Use `oocheckls computer_name` from a DOS window.

Make sure that you have the following environment variable set:

```
OO_VC_VERSION=6.0
```

Double-click on `ExampleProject.dsw` from your explorer, which should run Microsoft Visual C++ 6.0.

The project has a number of custom build steps already configured for you, so a simple `build` command (F7) should create the federated database, compile the schema, and build the application. For more information on how to set up your own environment like this, download the `VCWRKSPC.ZIP` file from the info center, which has a sample workspace as well as step-by-step instructions on how to do this yourself.

UNIX

The UNIX setup involves modifying a configuration file (`unix.cfg` or `local.cfg`) to match local hostnames and locations of libraries and include files. Be sure to use correct hostnames and full pathnames to directories and file locations.

Makefiles

Objectivity/DB uses multithreaded libraries. This may require adjusting makefiles to use special flags and compiler options. Please verify the use of any special flags with platform-specific compiler documentation.

As part of the general setup, makefiles need additional information on where Objectivity/DB is installed and the language bindings being used. When extending or modifying the makefile, the following helpful hints may save time and effort:

- Use full pathnames to boot files and directories. Avoid the use of the dot (`./`) syntax. Avoid Uniform Naming Convention (UNC) names to start. If UNC names are to be used, please refer to the FAQ and documentation concerning UNC names.
- Use only one installation of Objectivity/DB per machine. Several different versions of Objectivity/DB can cause problems with runtime as well as development environments.
- Verify that the pathname to the Objectivity/DB include directory is correct. If using STL, include the `local.cfg` file and appropriate link and compiler options. Please refer to the Objectivity/C++ STL book for more information.

This is only a general description of what is necessary. Please refer to the example makefile for more details.

Java Setup

Setting the Environment

Before you build any applications, it is important to first make sure that the Java class path is properly set. The JDK uses the environment variable `CLASSPATH` in both NT and UNIX to determine which directories contain the required runtime libraries. Find the file `oojava.jar` on your system and make note of its path. Then double check to make sure it is in your `classpath`.

On Windows NT:

```
set CLASSPATH=d:\objy51\lib\oojava.jar;
```

You may want to set the environment variable from the system control panel rather than the DOS prompt.

On UNIX:

```
export CLASSPATH=/usr/object/solaris4/java/lib/oojava.jar:
```

Creating the Federated Database

The next step involves creating the federated database. From the command prompt or the UNIX shell, enter the following:

On Windows NT:

```
oonewfd -fdfilepath federation.fdb -lockserverhost  
        %COMPUTERNAME% -fdnumber 141 bootfile
```

On UNIX:

```
oonewfd -fdfilepath federation.fdb -lockserverhost 'hostname'  
        -fdnumber 141 bootfile
```

Building the Java Source Code

Next, compile the Java source:

```
javac *.java
```

Running the Application

In UNIX C++:

From a command line prompt, type `./hello`.

In MS VC++:

Select **Project>Run**

In UNIX Java:

From a command line prompt, type `java helloworld`.

In Windows Java:

Select **Project>Run**

Debugging and Locks

When debugging or troubleshooting a persistent Objectivity/DB application in a debugger, the debugger may trap Objectivity/DB clean-up signal handlers and leave read and update locks open. Follow the process below to clean these locks. This will also work to restore a federated database if there are locking problems during testing.

- Run the `oocleanup -local bootfile`.
- To verify that a federated database lock status is in a clean state, or to shut down the lock server process, use the steps below. The status on these tools should have a clean state with no outstanding locks. If there are outstanding locks or journal files please refer to the administration guide or call support for help.
 - a. Run `oocleanup bootfile`.
 - b. Run `oolockmon bootfile`.
 - c. Verify there are no outstanding `.JNL` files.

The most common cause of locking errors is a mismatched commit or abort with a transaction start. Lock wait settings and other errors can be identified using the `oolockmon` tool.

NOTE On all tools, use the `-help` option to see what other options are available. For example, the `oolockmon` tool has a `-detail` option that is very useful. Please refer to the Objectivity/DB Administration book for more information.

Automatic recovery is another option for Objectivity/DB applications. Automatic recovery can be initiated by the lock server or the client application. Refer to the Objectivity/DB Administration book for more information.

The Fault Tolerant Option (*FTO*) and Data Replication Option (*DRO*) are two optional administration features which provide additional flexibility in this area. They are beyond the scope of this tutorial but are discussed in detail in the Objectivity/FTO and Objectivity/DRO book.

Exploring the Database

Now that you have run `helloworld`, this chapter provides a closer look at the details of the application. Through discussion and exercises using tools and status-gathering functions, you will become more familiar with persistence concepts and Objectivity/DB.

This chapter explores:

- Examining the database: `oobrowse` and `ootoolmgr`
- Structuring for optimal access and performance
- Tips for good design

Examining the Database

Several tools can be used to view where and how objects are stored in Objectivity/DB.

To analyze the `helloworld` program:

1. Look at the federated database in detail using the object browser `oobrowse` (Windows) or `ootoolmgr` (UNIX).
2. Explore the concepts of transactions, cache, and handles using `ooRunStatus()`.

oobrowse and ootoolmgr

The object browser `oobrowse (ootoolmgr)` is a powerful tool for looking at the way objects are stored in Objectivity/DB. It displays the storage hierarchy, databases, containers, and objects, as well as any other special relationships such as associations. The object identifier, or OID, is displayed along with the objects and is a great source of storage information. The OID is encapsulated in the handle and is used internally to store and retrieve an object.

The `oobrowse(ootoolmgr)` command gives a more static layout of the objects than `ooRunStatus`. Along with application timing information, `ooRunStatus()` provides a good performance-based runtime picture of how objects interact with other objects and the object cache. This, combined with the flexibility of Objectivity/DB architecture, can lead to the adjustment of cache size and other performance factors that can sometimes at least double the access speed.

ooRunStatus

The `ooRunStatus()` function is run after the first federated database open call, then compared to the other `ooRunStatus` executed after the commit call.

The procedure for using `ooRunStatus` is:

1. Start the application and set any cache or other parameters.
2. After the first federated database open command, place an `ooRunStatus` call.
3. Compare all additional `ooRunStatus` calls to the preceding call to obtain a difference in the various parameters.

An additional benefit can be obtained by stopping the application in the debugger and running `oolockmon` and `oocleanup` to see how locking and journaling affect the application. Be sure to use `oocleanup -local` to cleanup any transactions if the application does not run to completion. Please refer to the Objectivity/DB Administration book for additional information on `oolockmon` or `oocleanup`, or use the `-help` flag to view the various parameters for these commands.

When examining `ooRunStatus` output, look for time-consuming things like number of disk reads, number of virtual arrays resized, or the number of handles used. For example, an application which creates a new handle inside a `for` loop instead of reusing it will have a large number of handles open and created. The number of buffer reads indicates how well the object cache is working. Without a buffer read, the disk must be accessed. With objects clustered efficiently, the number of buffer reads will be high.

There are many tuning and storage possibilities in Objectivity/DB. Refer to the Objectivity Technical Support web site for more information.

Structuring for Optimal Performance

In a relational database, the only way of accessing data is through queries, which may or may not be optimized by indexes. Because these are expected features of a database, Objectivity/DB supports these and many more data access methods. With the ability to connect objects together in many ways (graphs, trees, and patterns), Objectivity/DB permits the use of directed or tuned access to single or multiple objects. Object graphs and other object-oriented “patterns” create a

powerful and flexible means of accessing and manipulating data. This is why Objectivity/DB has a proven track record of providing world-class solutions to problems that other databases have not been able to solve.

Data structures

Data structures are basically pieces of data connected together in a particular way. Depending on what kind of data structure or algorithm you use, certain operations, such as searching, sorting, inserting, and deleting, are faster or slower. Examples of data structures include sorted arrays, lists, trees, and hash tables. In Objectivity/DB, it is possible to use such structures to provide quick access to persistent data as well. Lists and B-trees from STL or hash tables and arrays from Objectivity/DB are already provided, saving you the trouble of having to re-implement them. A data structure is usually not general-purpose enough to solve every need. With Objectivity/DB, it is possible to customize or optimize the operations that are performed most often.

For example, an application may use a large map as an *access construct* to efficiently locate individual elements by an exact key-match. The map data structure can be stored separately from the actual data, and the map can store references to your data objects rather than the objects themselves. By properly encapsulating the lookup tables and the physical storage routines, it may even be possible to replace one kind of access construct with another, providing better performance on the operations you perform most often, without breaking the rest of your code.

Using Objectivity/DB's storage hierarchy of databases and containers can also assist you in breaking up your data into smaller more manageable pieces.

Sometimes during design, it is difficult to determine when a map, iterator, or collection will be needed. Some guidelines for access constructs follow.

- Keep access constructs in a separate database or container from the objects they are indexing.
- Match meaning in the application to databases and/or containers.
- Design or discover access patterns and use appropriate access constructs.
- Use `ooRunStatus` for testing performance considerations.

Iterators and Tree Access

An iterator by itself is just a way to access objects sequentially. Since a C++ iterator is also an object reference, the user-defined methods can be called from the iterator.

The iterator with a predicate and a pre-created index uses a hybrid tree-based algorithm with roughly $O(\log N)$ access performance. This is good for large range-based queries.

Maps and Hash Tables

An Objectivity/DB `ooMap`, in contrast to an STL map, is a hash table. Access to the objects is achieved through the use of a name associated with an object identifier. The bucket-based algorithm will need some tuning on maps with more than 50 thousand elements (50K). You can either adjust the hash table default parameters, or use a map of maps. To adjust a hash table, the bucket size and the page size are important. A map-of-maps-strategy forces segmented searching and can be very speedy. The hash algorithm may be over-ridden for special demanding applications.

Large maps typically belong in a separate container. Applications that update maps continually may benefit from a map-of-maps-strategy where the maps are isolated in both contention and storage.

The STL map and multi-map have superior storage and flexibility attributes. The `d_allocator` allows flexible storage characteristics. However, STL maps are based on a B-tree and therefore have a lookup time which is more like $O(\log N)$. Java persistent collections are also very useful for custom storage and retrieval strategies.

Storage Strategies

When expanding an application, a storage hierarchy (encapsulated class) can be taken into consideration. Databases that have meaning in an application domain can be moved and located close to processing centers. A form of load balancing, distributing databases this way can not only improve performance, it can add a degree of flexibility during application testing and deployment. Collections of databases can also have meaning in an application. A federated database can hold up to 65535 (64k) databases.

The `d_session` class in Objectivity/C++ is an example storage class. It includes transaction encapsulation and reuse, a multi-threading interface and several containment strategies. Please refer to the documentation on the `d_session` class for more information. A similar class is also available for Java.

Class Strategies

There are certain restrictions on the kinds of member types you can use in a persistence-capable class. The restrictions and the reasoning behind them are described for C++ and Java.

Member Types in Objectivity/C++

The main restriction in C++ is that a data member cannot be a pointer. Pointers in C++ are frequently used to connect graphs of objects together. However, storing pointers persistently makes no sense in an ODBMS unless there is some kind of

pointer “swizzling” occurring, where the pointers get translated from a client-specific memory address to something persistent and back each time the data is requested by a different client. In Objectivity/DB, pointers are not swizzled, so they are stored as 32-bit numbers, which would not point to anything useful when the objects they contain are loaded on other clients.

Pointers are used to connect objects, as well as to manipulate variable-size dynamic arrays. In Objectivity/DB, you can use an `ooRef` or an association to connect two persistence-capable objects together, and an `ooVArray` or an `ooVString` to store variable-size arrays of things.

`ooRef` can only point to objects which derive from `ooObj`, because only things that derive from `ooObj` have an OID, which is how `ooRef` accesses an object.

C++ also supports embedded objects. Objectivity/DB permits the use of embedded objects as well, as long as their types are not persistence-capable. Therefore, to achieve object aggregation, you can use two mechanisms:

- `ooRef` or relationships to other `ooObj` derived types
- Embedded classes or structures for non-`ooObj` derived types

Objectivity/C++ supports static-sized arrays of `ooRefs` as well as non-`ooObj` types.

Member Types in Java

In Java, a persistent object can contain primitive types (`int`, `char`, `float`, `long`, etc.) and certain standard Java classes, such as `String`, `Date`, and `StringBuffer`. There is a list of all the supported types permitted. If it is not on this list, then the member must be of a persistence-capable type, such as `ooObj`-derived classes, or of a persistence-implementing type. If none of these conditions hold true, the member must be a transient type.

Objectivity for Java also supports relationships, which are basically the same thing as C++ associations, and arrays of persistence-capable objects, primitive types, and `Strings`.

Any data member which is marked `Transient` is not stored in Objectivity/DB. Transient members can be initialized automatically when they are brought into the local cache with the `activate` method.

Associations

Associations provide a level of consistency in object relationships. With associations, one-to-one, one-to-many, and many-to-many relationships between objects can be made easily. Locks and deletes can optionally propagate automatically across to related objects. While the to-many associations are quite suitable for low-cardinality (a couple of hundred objects per association), certain

operations can degrade in performance as the cardinality increases into the hundred-thousands. Therefore, custom-made associations can be made with persistent collections and other techniques. Refer to the documentation or the Objectivity Technical Support web site for more information.

Tips for Good Design

The following are some general tips which will assist you in developing fast, responsive applications.

- Avoid large VArrays of embedded objects. You may want to replace them with VArrays of ooRefs to persistence-capable objects.
- Avoid having lots of short transaction models. Use other constructs to consolidate actions into longer transactions. Pass objects to a specific worker thread with an open transaction, or use a longer transaction with commit and hold.
- Keep indexes and large maps in a separate container from the data they reference. Use scope names to name and later look up maps and other lookup objects.
- Watch the database and container sizing. Keeping container sizes between 30 and 62 percent of maximum is a good idea if possible. Optimizing the containers at about 30 percent of maximum will provide better performance.
- Do not throw exceptions in the error handler. This can cause serious errors. Objectivity/DB expects error handlers to return a value and pop their stack frames properly. If it is absolutely necessary that an exception be thrown in an error handler, contact Objectivity, Inc. support or your system engineer for help.

Other Examples Available

This `helloworld` example is just a basic example of persistence. The rental fleet example available on the Info Center web site is a full example of a distributed multi-threaded program.

The rental fleet program demonstrates several new concepts, maps, indexes, and different ways to handle transaction encapsulation.

To extend the rental fleet example, STL can be used to simplify the object model and streamline the relationships using sets and lists.

5

Testing Your Knowledge

Test your knowledge of Objectivity/DB by answering these 25 multiple choice questions. Answers are supplied at the conclusion of this section.

1. Objectivity/DB is based on what kind of architecture?
 - a. Central server
 - b. Distributed
 - c. Application server
 - d. All of the above

2. Documentation on Objectivity/DB products and services and sample code can be found:
 - a. On the web site www.objectivity.com
 - b. On the release disk
 - c. In the language manuals
 - d. All of the above

3. An Object Identifier is:
 - a. A class name
 - b. The `ooObj` class
 - c. A construct containing persistent location information
 - d. A 64bit Identifier contained inside a class handle
 - e. C and D

4. To add Objectivity/DB persistence to an existing application, the following must be true:
 - a. The application must be based on files
 - b. The classes in an application must relate to a persistent capable class
 - c. An application must be distributed
 - d. Objectivity/DB must be added to all classes

5. In an Objectivity/DB evaluation, the following statements are true:
 - a. The evaluator is supplied with a support alias
 - b. Access to web site resources is provided on request
 - c. The evaluator can use 1800-SOS-OBJY support
 - d. All of the above

6. A typical Objectivity/DB coding project involves:
 - a. Code and configuration management
 - b. Management involvement
 - c. Prototyping and critical section testing
 - d. Web site resources and information searching
 - e. A, C and D

7. In Objectivity/ C++, three files are generated from the pre-processor. What are these files?
 - a. Two header files and a C++ file
 - b. Binary libraries
 - c. Makefiles
 - d. A and C

8. A fourth file can be added to the three C++ generated files. This C++ file does what?
 - a. Handles dependency information for header files
 - b. Can optionally hold C++ method definitions
 - c. Is used for resolving links into the Objectivity/DB libraries
 - d. A and B

9. The Java language binding uses what mechanism to create persistent objects?
 - a. A JNI interface along with persistence-capable class relations and interfacing
 - b. An `oojava.jar` and a dynamic library
 - c. A special method calls `fetch()` and `mark_modified()` in class methods
 - d. All of the above

10. Java persistent classes must use special garbage-collectable containers for what reason?
 - a. So disk space can be reused
 - b. Because persistent classes can change on the fly
 - c. Because they contain collections
 - d. They do not need to be used

11. An Objectivity/DB object cache:
 - a. Is tunable
 - b. Can be set to an unlimited size
 - c. Holds objects for use at the page level
 - d. All of the above

12. Objectivity/DB has “one logical view of objects from any node on the network.” How does it do this?
 - a. It interfaces with the web
 - b. It has a flexible and distributed architecture
 - c. It is an object server
 - d. It swizzles pointers in memory

13. A federation is created using what tool?
 - a. Your application
 - b. A star fleet and men in funny sleek clothing with special symbols
 - c. The `oonewfd` tool
 - d. B and C

14. The lock server provides what essential service?
 - a. Single user object access
 - b. Transaction and multi-user support
 - c. Lock caching and OID manipulation
 - d. Federation validation

15. What must be present to execute an Objectivity/DB program successfully?
 - a. The federation and `ooidy` must be running
 - b. The schema must be loaded
 - c. The federation must exist and the lock server be running
 - d. B and C

16. When using Objectivity/DB tools, what is important?
 - a. The lock server must be running
 - b. All parameters must be filled in
 - c. The hostname and full pathname to the boot file must be specified
 - d. A and C

17. If an application exits the debugger abnormally, what needs to be done?
 - a. Reboot the computer
 - b. Exit all applications and log in again
 - c. Run `oocleanup -local <boot file>`
 - d. Stop the lock server

18. What process is used to verify that the lock server can be shut down?
 - a. Debug the lock server by attaching to the process
 - b. Check if `oolockmon` and `oocleanup` have outstanding locks
 - c. Verify that `oolockmon` and `oocleanup` have no outstanding locks and there are no journal files
 - d. Verify that the boot file exists

- 19.** An OID can reveal what kind of information?
 - a. What page an object is on
 - b. Which database the object resides in
 - c. What host the database is on
 - d. Where the boot file is
 - e. A and B

- 20.** A storage strategy is used to:
 - a. Store objects outside of the federation
 - b. Intelligently scale and integrate persistent objects
 - c. Avoid locking conflicts and unnecessary disk access
 - d. Manage cache size and handle reuse
 - e. B, C and D

- 21.** MROW is used for:
 - a. Managing many rows of objects
 - b. Exclusively locking the federation
 - c. Suspending a process
 - d. Handling many transactions in an elegant way

- 22.** Object access patterns, when applied intelligently, benefit the application in what way?
 - a. Slow down an application because they need to be parsed
 - b. Link intelligently to many diverse libraries
 - c. Give applications flexible alternatives to improve performance
 - d. Play audio and video effortlessly

- 23.** What two techniques can be used to solve transaction performance problems?
 - a. SQL and OQL
 - b. MROW and Iterators
 - c. Design, modeling and discovery
 - d. Hacking the lock server

- 24.** Associations are used for what logical construct?
- a.** Joining object hierarchies
 - b.** Maintaining integrity between objects
 - c.** Subclass references
 - d.** Runtime type identification
- 25.** What world-class project is Objectivity, Inc. known for?
- a.** Mapping stars and space
 - b.** Finding the secrets of the universe at CERN
 - c.** Intelligent baby monitoring
 - d.** Iridium
 - e.** All of the above

Answers:

1 b, 2 e, 3 e, 4 b, 5 d, 6 e, 7 a, 8 b, 9 d, 10 a, 11 d, 12 b, 13 c, 14 b, 15 c, 16 d, 17 c, 18 c, 19 e, 20 e, 21 d, 22 c, 23 c, 24 b, 25 e or d.

