ActiveX Interface for ObjectStore

Release 3.0, March 1998

ObjectStore, Object Design, the Object Design logo, LEADERSHIP BY DESIGN, and Object Exchange are registered trademarks of Object Design, Inc. ObjectForms and Object Manager are trademarks of Object Design, Inc.

Microsoft is a registered trademark and Windows, Windows NT, OLE, and ActiveX are trademarks of Microsoft Corporation.

All other trademarks are the property of their respective owners.

Copyright © 1989 to 1998 Object Design, Inc. All rights reserved. Printed in the United States of America. Except as permitted under the Copyright Act of 1976, no part of this publication may be reproduced or distributed in any form or by any means, or stored in a database or retrieval system, without the prior written permission of the publisher.

COMMERCIAL ITEM — The Programs are Commercial Computer Software, as defined in the Federal Acquisition Regulations and Department of Defense FAR Supplement, and are delivered to the United States Government with only those rights set forth in Object Design’s software license agreement.

Data contained herein are proprietary to Object Design, Inc., or its licensors, and may not be used, disclosed, reproduced, modified, performed or displayed without the prior written approval of Object Design, Inc.

This document contains proprietary Object Design information and is licensed for use pursuant to a Software License Services Agreement between Object Design, Inc., and Customer.

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

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

Preface .................................................. vii

Chapter 1  Overview of the ActiveX Interface for ObjectStore

Installing the ActiveX Interface for ObjectStore ........ 4
  System Requirements ................................ 4
  Windows 95 Patch Requirements ....................... 4
  Installation ......................................... 4
  Uninstalling the ActiveX Interface for ObjectStore .... 5

Examples ................................................. 6
  Books Example ....................................... 6
  Portfolio Example .................................... 6
  Travel Example ...................................... 7

A Tour of the Books Example ............................ 8
  The Visual C++ Project .............................. 8
  The Visual Basic Project ............................ 10
  The Books Database ................................ 11

Chapter 2  Building OSAX Object Servers ................. 13

Building an Object Server .............................. 14
  Basic Operations Supported by an Object Server ...... 16
  Basic Types Supported by an Object Server ........... 17
  Object Server Configuration .......................... 18

Type Description File .................................. 19
  Library Specification ............................... 19
<table>
<thead>
<tr>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object Specification........................................................................................................... 21</td>
</tr>
<tr>
<td>Accessing OSAX Objects......................................................................................................... 29</td>
</tr>
<tr>
<td>Creating Objects..................................................................................................................... 31</td>
</tr>
<tr>
<td>Expose a Method on One Class That Creates Instances of Another Class.................................. 31</td>
</tr>
<tr>
<td>Expose a C++ Static Member Function That Creates Instances of Its Class................................ 31</td>
</tr>
<tr>
<td>Expose a C++ Constructor That Creates Instances of Its Class........................................... 32</td>
</tr>
<tr>
<td>Deleting Temporary C++ Objects............................................................................................... 33</td>
</tr>
<tr>
<td>Accessing and Creating Character Strings................................................................................ 34</td>
</tr>
<tr>
<td>Creating Persistent Character Strings..................................................................................... 35</td>
</tr>
<tr>
<td>Accessing Collections............................................................................................................... 37</td>
</tr>
<tr>
<td>Defining a Collection............................................................................................................... 37</td>
</tr>
<tr>
<td>Creating a Collection of Objects............................................................................................. 38</td>
</tr>
<tr>
<td>Iterating over Collections....................................................................................................... 39</td>
</tr>
<tr>
<td>Querying Collections................................................................................................................ 39</td>
</tr>
<tr>
<td>Deleting a Collection............................................................................................................... 40</td>
</tr>
<tr>
<td>Creating and Using Database Roots........................................................................................... 41</td>
</tr>
<tr>
<td>Object Server Exceptions......................................................................................................... 42</td>
</tr>
<tr>
<td>Signaling Exceptions from an Object Server............................................................................. 42</td>
</tr>
<tr>
<td>Handling Exceptions from an Object Server............................................................................. 42</td>
</tr>
<tr>
<td>The osgentyp Utility................................................................................................................ 44</td>
</tr>
<tr>
<td>Command-Line Syntax............................................................................................................... 44</td>
</tr>
<tr>
<td>Generating UUIDs for attributes............................................................................................... 45</td>
</tr>
</tbody>
</table>

### Chapter 3 Using OSAX with Microsoft ATL ................................................................. 47

<table>
<thead>
<tr>
<th>OSAX and ATL Concepts........................................................................................................... 49</th>
</tr>
</thead>
<tbody>
<tr>
<td>Terminology.................................. 49</td>
</tr>
<tr>
<td>OSAX Object Servers........................ 49</td>
</tr>
<tr>
<td>OSAX Classes.................................. 49</td>
</tr>
<tr>
<td>OSAX Methods.................................. 50</td>
</tr>
<tr>
<td>OSAX Instances................................. 50</td>
</tr>
<tr>
<td>OSAX Instance Classes......................... 51</td>
</tr>
<tr>
<td>Implementing an Object Server with ATL............................................................. 52</td>
</tr>
<tr>
<td>Step 1. Use the ATL COM AppWizard.............................................................. 52</td>
</tr>
</tbody>
</table>
Contents

Step 2. Make Project-Level Changes .......................................... 52
Step 3. Add ATL Classes .................................................. 53
Step 4. Add base Classes, Interfaces, and So On ...................... 54
Step 5. Add Methods ...................................................... 56
Creating OSAX Instances .................................................. 61
OSAX Instances for Existing C++ Objects ............................. 61
Object Deletion ........................................................... 63
API Reference ............................................................. 64
OSAX Methods ............................................................. 64
Exceptions ................................................................. 64
Template Classes ......................................................... 65
OSAX Instance Data Members ............................................. 65
Methods of the CInstance::CDataPtr Class ......................... 66
Using OSAX Instances to Implement Other Object Servers ........ 67
Collections ............................................................... 68
Macros for Class Definitions of OSAX Instances and Conversions 68
Global Functions ........................................................ 69
Overloadings of Operator new ........................................... 69

Chapter 4 Distributed OSAX Object Servers ......................... 71
Using Object Servers with Web Servers ............................... 73
Accessing Networked Object Servers Using DCOM .............. 74
DCOM Client Configuration .............................................. 74

Chapter 5 The Portfolio Sample Application ......................... 77
Portfolio Data Model ....................................................... 80
Implementation of the Portfolio Sample Application ............... 80
Visual Basic Implementation of the Application .................... 80
Implementation of the Object Server .................................. 82

Chapter 6 ActiveX Interface Reference ................................. 83
OSAX Interfaces at a Glance ............................................. 84
I O SAX Collection ....................................................... 86
Purpose ................................................................. 86
Preface

Purpose

ActiveX Interface for ObjectStore describes how to install and use the ActiveX Interface for ObjectStore (OSAX) Release 3.0 on a Windows NT or Windows 95 system.

Audience

This book is primarily intended for developers who will build OSAX object servers for accessing ObjectStore databases. It assumes that the developer is familiar with ObjectStore and Visual C++. This book (especially Chapter 1) is also intended for client-side users who want to access the database as ActiveX clients. They should be familiar with either Visual Basic or a scripting language that interacts with ActiveX.

Scope

This document provides all the necessary information for installing OSAX Release 3.0 and for building an OSAX object server.

How This Book Is Organized

The first chapter provides an overview of the product and its main features, and describes how to install OSAX. The later chapters describe how to build OSAX object servers.

Notation Conventions

This document uses the following conventions:

<table>
<thead>
<tr>
<th>Convention</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bold</td>
<td>Bold typeface indicates user input or code.</td>
</tr>
<tr>
<td>Italic sans serif</td>
<td>Italic sans serif typeface indicates a variable for which you must supply a value. This most often appears in a syntax line or table.</td>
</tr>
</tbody>
</table>
### Internet Sources of More Information

#### World Wide Web
Object Design’s support organization provides a number of information resources. These are available to you through a Web browser such as Internet Explorer or Netscape. You can obtain information by accessing the Object Design home page with the URL [http://www.objectdesign.com](http://www.objectdesign.com). Select **Technical Support**. Select **Support Communications** for detailed instructions about different methods of obtaining information from support.

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

#### Automatic email notification
In addition to the previous methods of obtaining Object Design’s latest patch updates (available on the **ftp** server as well as the Object Design Support home page), you can now automatically be notified of updates. To subscribe, send email to **patch-info-request@odi.com** with the keyword **SUBSCRIBE patch-info <your siteid>** in the body of your email. This will subscribe you to Object

---

**Convention** | **Meaning**
---|---
*Italic serif* | In text, italic serif typeface indicates the first use of an important term.
[] | Brackets enclose optional arguments.
{a | b | c} | Braces enclose two or more items. You can specify only one of the enclosed items. Vertical bars represent OR separators. For example, you can specify *a* or *b* or *c*.
... | Three consecutive periods indicate that you can repeat the immediately previous item. In examples, they also indicate omissions.
Design’s patch information server daemon that automatically provides site access information and notification of other changes to the online support services. Your site ID is listed on any shipment from Object Design, or you can contact your Object Design Sales Administrator for the site ID information.

**Training**

If you are in North America, for information about Object Design’s educational offerings, or to order additional documents, call 781.674.5000, Monday through Friday from 8:30 AM to 5:30 PM Eastern Time. You can reach the Education Hotline at 781.674.5047.

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

**Your Comments**

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

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

You can also fax your comments to 781.674.5440.

*ActiveX Interface for ObjectStore*
Preface
Chapter 1
Overview of the ActiveX Interface for ObjectStore

The ActiveX Interface for ObjectStore (OSAX) exposes ObjectStore databases and their contents to Visual Basic and other ActiveX Automation controllers and scripting hosts. Database objects are implemented in ObjectStore using C++ and described in the ObjectStore schema using a superset of the C++ object model. The ActiveX Interface for ObjectStore provides a mapping between the ObjectStore schema and the object model underlying OLE. It also provides a mechanism for communicating object references and values between an ObjectStore database and an Automation controller.

OSAX features

The ActiveX Interface for ObjectStore includes the following features:

• Ability to access and traverse objects in the database using the normal syntax of the host language for accessing object properties and methods.

• Transparent access to the public data members of any ObjectStore database. You can integrate custom behavior using generated ActiveX type libraries.

• Access to the ObjectStore collections facility. You can use the collections facility transparently in place of Visual Basic collections to provide persistence, higher performance, larger scale, and associative query language.
- The full transaction and recovery capabilities of ObjectStore. ActiveX object references remain valid across transactions for easy GUI integration.

- All ActiveX configuration options, including InProc, Local, and DCOM communication, and “early” binding for direct function-call performance in InProc configuration.

ActiveX Automation is an object-oriented dynamically typed RPC mechanism that allows programmable automation controllers (typically programming languages or development tools) to access object servers by invoking properties and methods on the exposed interfaces.

For detailed information about the OSAX ActiveX interfaces, see Chapter 6, ActiveX Interface Reference, on page 83.

OSAX and the ActiveX architecture

There are three key ingredients in the ActiveX architecture that allow it to represent the contents of ObjectStore databases with good performance and fidelity:

- The IUnknown and IDispatch COM interfaces establish the well-formed model of object identity needed by ObjectStore object references. They also establish the protocol for retrieving properties and invoking methods of the referenced objects.

- You use the dynamically typed transport mechanism for primitive data types (such as integers, strings, references, and arrays) to deliver analogous C++ member types by value.

- You can easily extend the domain of object types understood by both controller and server, using OLE type libraries generated from an ObjectStore schema.

The following diagram shows the role the object server fills between an Automation controller and an ObjectStore database.
The ActiveX interface supports several different configurations for the OSAX object server and several means of making type information available to OLE. The ActiveX interface enables you to expose objects in an ObjectStore database, making them accessible to an ActiveX client (such as Visual Basic Script) with little or no direct programming of COM interfaces.
Installing the ActiveX Interface for ObjectStore

The ActiveX Interface for ObjectStore is distributed as a self-extracting archive file that contains an automated setup program.

System Requirements

Minimum system requirements for the ActiveX Interface for ObjectStore are

- One of the following:
  - Microsoft Windows NT 4.0 with Service Pack #3
  - Microsoft Windows 95 with Service Pack #1 and DCOM extensions
  - Windows 95B and DCOM extensions
- Microsoft Visual C++ 5.0 with Service Pack #3
- ObjectStore 5.1 for Windows NT or Windows 95

Windows 95 Patch Requirements

The original distribution of Windows 95 contained OLE libraries with a number of problems. You must install the following patches to Windows 95 in order to use this product successfully:

- Windows 95 Service Pack #1
- DCOM for Windows 95

Note: DCOM is required whether or not you intend to use distributed configurations. It contains updated OLE libraries that are required for local use as well.

Installation

Install the ActiveX Interface for ObjectStore by using setup. After setup performs the initial installation, it registers the supplied OSAX object servers in the system registry. It updates the system environment to refer to the new libraries and executables. It also adds the ObjectStore ActiveX AppWizard to the Microsoft Developer Studio Tools menu.
Uninstalling the ActiveX Interface for ObjectStore

Select the **Uninstall** icon in the ActiveX Interface for ObjectStore’s program group to remove the product. You can also use **Add/Remove Programs** from the Control Panel to remove the product.
Examples

The ActiveX Interface for ObjectStore includes example programs and databases. Each of the examples contains a Visual C++ 5.0 project with the source for the object server. You can build the project and register the object server for use. The examples will help you understand how to use the product and how to set up Visual C++ projects for it.

Books Example

The Books example is a simple database of book titles and authors. It includes a prebuilt database, an object server built using the ActiveX Interface for ObjectStore, and a Visual Basic executable that uses that object server to display the contents of the database.

Build this object server in the Books - Win32 release project configuration. This object server can then be used directly by the Visual Basic 5 application in the vbclient subdirectory. The Visual Basic 5 application contains an option to create a new database.

The Books example is a good starting point to learn how to build and use an OSAX object server; see A Tour of the Books Example on page 8.

Portfolio Example

The Portfolio example illustrates a simple ActiveX Interface for ObjectStore application implemented in C++ and Visual Basic using an OSAX object server. The application provides a user interface to do the following:

- Selecting among a number of defined financial securities and portfolios
- Computing the valuation of a selected instrument over a period of time
- Displaying the result using a standard graphing component

The source code for this example is common to both the ObjectStore and PSE products. It illustrates how an object server can target either storage system. The design workspace contains separate projects to build object servers for either product.
Build this object server in the OSAXPORT - Win32 Release project configuration. The object server is intended for use with a database of stock data. This database is supplied in prebuilt form in osaxport.db.

The Visual Basic 5 version of the Portfolio application provides an Open Database dialog. You can use this dialog to locate the prebuilt Portfolio database (osaxport.db) and open it with your newly built control.

For more information about the Portfolio example, see Chapter 5, The Portfolio Sample Application, on page 77. This example is particularly pertinent to applications that work with collections.

**Travel Example**

The Travel sample application demonstrates the use of the ActiveX Interface for ObjectStore with Internet Explorer. It uses the Visual Basic Script engine to orchestrate the client-side presentation of multimedia content (various vacation experiences) stored in the database. For more information about the Travel sample application, see the Travel ReadMe.txt file.

The source code for this example is common to both ObjectStore and the ObjectStore PSE for C++ products. It illustrates how an object server can target either storage system. The design workspace contains separate projects to build object servers for either product.

Build this object server in the OSAXTRAVEL - Win32 Release project configuration. The object server is intended for use with a database of catalog data. This database is supplied in prebuilt form in osaxtravel.db.

The Travel Web page obtains its database from the parent directory in which the control is located.
A Tour of the Books Example

This section introduces you to the elements of an OSAX project and to the process of creating a project, using the Books example that comes with OSAX. The Books example uses ActiveX with Visual Basic to access objects in a prebuilt ObjectStore database of book titles and authors. After building the OSAX object server for this example, you can access the database from Visual Basic and display the following form:

![Books Database Creator and Viewer](image)

The \c:\odi\osax\examples\books\directory contains the files for the Books example. See ReadMe.txt for an explanation of each file.

The following sections describe the different elements of the Books example.

The Visual C++ Project

The OSAX object server for the Books example is built using a Visual C++ project. This project contains all the C++ source code for the project, as well as the type description file and database schema definition. The Books example project files are in the directory \c:\odi\osax\examples\books\objectserver. You can
examine, modify, and build that project using Microsoft Visual C++ 5.0.

The Object Server

The OSAX.Books object server is an ActiveX object that provides access to the contents of Books, an ObjectStore database of book titles and author names. The object server consists of a dynamic link library (DLL) that links application-specific methods with the database libraries and implements the interfaces required for ActiveX Automation, scripting, and other capabilities. The object server DLL is associated with an ActiveX type library that defines the classes, interfaces, properties, and methods provided by the object server.

The object server is built from C++ code and libraries generated by OSAX based on information provided in the type description file, books.ost.

Type description file

The type description file (books.ost) is a source file that implements the OSAX object server. It includes descriptions of the ActiveX interfaces that are to be built into the object server and used for accessing objects in the database. For detailed information about type description files, see Chapter 2, Building OSAX Object Servers, on page 13.

Note: OSAX also supports the Microsoft Active Template Library (ATL) as a way to create COM interfaces for persistent objects stored in an ObjectStore database. Using ATL instead of type description files requires experience with ATL and C++, but it provides more flexibility and control when customizing COM interfaces.

The ObjectStore ActiveX AppWizard

The type description file books.ost was initially generated by the ObjectStore ActiveX AppWizard that is added to the Microsoft Developer Studio Tools menu when you install OSAX. The AppWizard can be used to create an OSAX object server. Since OSAX requires custom build rules, you can use the AppWizard to generate template files and create a Microsoft Developer Studio project. After generating these files, you can modify them for your specific project, as was done with the type description file for the Books example.

To start the AppWizard, start Microsoft Developer Studio, and then select Tools->ObjectStore ActiveX AppWizard. The...
A Tour of the Books Example

AppWizard screens are described in the AppWizard’s on-line help.

The Visual Basic Project

The file books.vbp contains the Visual Basic 5.0 project that enables you to build a Visual Basic application to access the database. The Visual Basic project for the Books example does (among other things) the following:

• Defines the design for the form used to display objects from the database.
• References the OSAX.Books object server, which appears in the Visual Basic dialog reached from the Visual Basic 5.0 Project->References menu.
• Provides the code for the Visual Basic subroutines and functions that access the object server.

Global properties

The OSAX.Books object server provides two global properties to the Visual Basic project:

• **ObjectStore**, an OLE object with entry points for accessing ObjectStore databases (**BeginTrans**, **CommitTrans**, **Rollback**, **OpenDatabase**, and **CloseDatabase**). The **ObjectStore** object is common to all OSAX object servers.

• **CBookElt**, an OLE object representing a class of objects stored in the database.

A subroutine from the Visual Basic application

The following subroutine from the Visual Basic application uses both of the global properties, **ObjectStore** and **CBookElt**:

```
Sub DisplayBookList()
    ObjectStore.BeginTrans
    Dim BookElt As IBookElt
    Dim Book as IBook
    Dim Author, Title As String
    Set BookElt = osDatabase.Value("Books", CBookElt)
    Do
        Set Book = BookElt.Book
        Author = Book.Author.Name
        Title = Book.Name
        List1.AddItem Author & " - " & Title
        Set BookElt = BookElt.Next
    Loop While Not BookElt Is Nothing
    ObjectStore.CommitTrans
End Sub
```
Chapter 1: Overview of the ActiveX Interface for ObjectStore

The first statement in the subroutine:

**ObjectStore.BeginTrans**

uses **ObjectStore** to start a transaction in the database. The last statement:

**ObjectStore.CommitTrans**

uses it to commit the transaction, freeing resources and allowing other processes to modify objects that were read from the database.

The following **Set** statement

**Set BookElt = osDatabase.Value(“Books”, CBookElt)**

passes **CBookElt** as an argument that specifies the class of **BookElt**, which is the named object or root object to be retrieved from the database. The **CBookElt** argument is used for type-checking purposes. Once the root object has been retrieved and type-checked, all further access to ObjectStore objects is automatically type-checked according to the declared Visual Basic types.

### The Books Database

The Books database contains three types of objects, each of which is defined in **books.ost** as a C++ class: **Author**, **Book**, and **BookElt**. The following table lists both the C++ class definitions for each of these classes and the corresponding Visual Basic properties and syntax used to access them:

<table>
<thead>
<tr>
<th>Visual Basic Code</th>
<th>Code in books.ost</th>
<th>C++ Class Definition</th>
</tr>
</thead>
</table>
| Dim Author As IAuthor  
Dim AuthorName as String  
Set Author = osDB(“SampleAuthor”).Value  
AuthorName = Author.Name | class Author  
{  
[propget]  
char* Name()data name;  
...  
}; | class Author  
{  
public:  
char *name;  
}; |
| Dim Book As IBook  
Dim BookName as String  
Dim BookAuthor As IAuthor  
Set Book = osDB(“SampleBook”).Value  
BookName = Book.Name  
Set BookAuthor = Book.Author | class Book  
{  
[propget]  
char* Name()data name;  
[propget]  
Author* Author()data author;  
...  
}; | class Book  
{  
public:  
char* name;  
Author* author;  
}; |
The expression `osDB("SampleAuthor").Value` looks up the database root named `SampleAuthor` in the specified database and returns its value as an object that can be observed using its defined property names.

**Note**: Constants for the loop differ in C++ and Visual Basic. The C++ `NULL` pointer value is translated as `Null` in OLE and as `Nothing` in Visual Basic.

<table>
<thead>
<tr>
<th>Visual Basic Code</th>
<th>Code in <code>books.ost</code></th>
<th>C++ Class Definition</th>
</tr>
</thead>
</table>
| Dim Books As IBookElt  
Dim Book As IBook  
Set Books = osDB("SampleBooks").Value  
Do  
  Set Book = Books.Book  
  Set Books = Books.Next  
Loop While Not Books Is Nothing | class BookElt  
{  
  [propget]  
  Book* Book()data book;  
  [propget]  
  BookElt* Next()data next;  
  ...  
}; | class BookElt  
{  
  public:  
  Book book;  
  BookElt* next = NULL;  
}; |
Chapter 2
Building OSAX Object Servers

An OSAX object server is an ActiveX object that provides access to ObjectStore databases. It links application-specific methods with the database libraries and implements the COM interfaces required for ActiveX Automation, Scripting, and other capabilities. Once you build an object server, you can distribute it to target computers and register it in the ActiveX system registry, making it accessible to ActiveX controllers.

The following sections provide detailed information for building an OSAX object server, including information about the type description (.ost) file that is used to implement the object server and the osgentyp utility that processes this file.
Building an Object Server

The following procedure outlines the steps for building a typical OSAX object server. As an alternative to this approach, you can use the Microsoft Active Template Library (ATL) to create COM objects; see Chapter 3, Using OSAX with Microsoft ATL, on page 47. The approach described here is simpler, but does not provide the flexibility of using ATL directly.

To build an object server, perform the following steps:

1. Use the ObjectStore ActiveX AppWizard to create a skeleton project.
2. Define the interfaces to your C++ objects in a type description (.ost) file.
3. Provide implementations of your C++ objects either directly in the project or through an independently created DLL.
4. Build the project.

The skeleton project produced in Step 1 includes a .def file. This file defines the exported entry points for the DLL that implements the object server. These points are the usual DLL registration entry points. All object server DLLs must export these entry points.

In Step 2, you write a type description (.ost) file, which defines the COM interfaces and object implementations in terms of C++ objects. The ObjectStore ActiveX AppWizard uses a custom build rule to run the osgentyp utility on the .ost file. For more information about the .ost file, see Type Description File on page 19.

The osgentyp utility generates the following files:

- A standard .idl file for the interfaces
- .rgs files for COM class registration
- A .rc resource file to include the .rgs files and the type library files in the DLL
- A C++ file (*_imp.cpp) for the COM object implementations

For more information about the osgentyp utility, see The osgentyp Utility on page 44.
Chapter 2: Building OSAX Object Servers

The .idl file is processed by Microsoft’s idl compiler, midl. The application wizard adds a custom build rule to the project to process the .idl file. The midl compiler generates the following files:

- A type library binary (.tlb) file
- A header file for the interface definitions
- Additional files that can optionally be used for marshaling

The ObjectStore ActiveX AppWizard invokes the midl compiler with the option for type library marshaling, so you can ignore the marshaling files.

Step 3 is the same as for any ObjectStore application. The application wizard creates an empty schema file and a custom build rule to run the osg utility on the schema file. If you are using an independent DLL for your C++ objects, your server does not have a schema and you should remove the schema file from your projects. You will need to include the header file for the DLL’s API in stdafx.h.

The *_imp.cpp file produced by the osgentyp utility contains the implementations of all the objects in the server. The C++ code uses the ATL interface to OSAX to implement the objects. If you decide later to use ATL directly, you can use the *_imp.cpp file produced by the osgentyp utility as a starting point. For information about using ATL, see Chapter 3, Using OSAX with Microsoft ATL, on page 47.

After all the C++ and resource files have been compiled, they are linked together along with the registration information and the type library. The final step is to register the server. The osgentyp utility and the application wizard produce a self-registering DLL that can be given to the program regsvr32. The project registers the DLL as the final step of the build process.
# Basic Operations Supported by an Object Server

An OSAX object server supports the following operations on databases and transactions:

<table>
<thead>
<tr>
<th>Object Type</th>
<th>Visual Basic Syntax</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>IOSAXObjectStore</td>
<td>ObjectStore.OpenDatabase <em>pathname</em></td>
<td>Open the specified database.</td>
</tr>
<tr>
<td></td>
<td>ObjectStore.BeginTrans</td>
<td>Begin a dynamic transaction.</td>
</tr>
<tr>
<td></td>
<td>ObjectStore.CommitTrans</td>
<td>Commit the current transaction.</td>
</tr>
<tr>
<td></td>
<td>ObjectStore.Rollback</td>
<td>Abort the current transaction.</td>
</tr>
<tr>
<td>IOSAXDatabase</td>
<td>osDB.Value(rootname, type)</td>
<td>Return the value in the named root, which must be of the indicated type.</td>
</tr>
<tr>
<td></td>
<td>osDB.Open</td>
<td>Open the database.</td>
</tr>
<tr>
<td></td>
<td>osDB.Close</td>
<td>Close the database.</td>
</tr>
</tbody>
</table>
Basic Types Supported by an Object Server

An OSAX object server provides a number of Automation Interfaces to represent ObjectStore types such as databases, collections, and cursors. The contents of databases—essentially, C++ pointers to C++ objects containing C++ data types—are converted to one of the following:

- Directly to the analogous ActiveX data types if possible (for simple literal types like numbers)
- To application-specific types provided by an OSAX type library, if available
- To a generic interface called IDispatch

The conversions performed on the various C++ types are listed in the following table:

<table>
<thead>
<tr>
<th>C++ Member Type</th>
<th>ActiveX Data Type</th>
<th>Visual Basic Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>char, int, long</td>
<td>I4</td>
<td>Long</td>
</tr>
<tr>
<td>float, double</td>
<td>R4 or R8</td>
<td>Float, Double</td>
</tr>
<tr>
<td>char*, signed char*,</td>
<td>BSTR (Literal values)</td>
<td>String</td>
</tr>
<tr>
<td>unsigned char*, char[]</td>
<td>IOSAXString* (Object values)</td>
<td>IOSAXString</td>
</tr>
<tr>
<td>void*</td>
<td>IDispatch*</td>
<td>Object</td>
</tr>
<tr>
<td>Application-specific</td>
<td>Generated interface from</td>
<td>Generated object type, or Object</td>
</tr>
<tr>
<td>pointers</td>
<td>typelib, if available; else IDispatch</td>
<td></td>
</tr>
<tr>
<td>NULL (void* == 0)</td>
<td>Null</td>
<td>Nothing</td>
</tr>
<tr>
<td>struct, class (embedded)</td>
<td>Generated interface from typelib, if available; else IDispatch*</td>
<td>Generated object type, or Object</td>
</tr>
<tr>
<td>os_array*</td>
<td>IOSAXCollection*</td>
<td>IOSAXCollection or Object</td>
</tr>
<tr>
<td>os_bag*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>os_collection* and</td>
<td></td>
<td></td>
</tr>
<tr>
<td>subclasses</td>
<td></td>
<td></td>
</tr>
<tr>
<td>os_dictionary*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>os_set*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>os_database*</td>
<td>IOSAXDatabase*</td>
<td>IOSAXDatabase</td>
</tr>
<tr>
<td>os_segment*</td>
<td>IOSAXSegment*</td>
<td>IOSAXSegment</td>
</tr>
</tbody>
</table>

Note: C++ character arrays require special treatment. See Accessing and Creating Character Strings on page 34.
Object Server Configuration

An OSAX object server is configured to run in the same process as the client. If you want out-of-process or remote servers, you can write a simple out-of-process server to manage the in-process OSAX servers, or you can use the ATL interface to OSAX to create an OSAX server as an executable or as an NT service.

OSAX object servers use a threading model of **Both**, which means that they are suitable for single-threaded and multi-threaded clients. Object servers expose each C++ class with a single dual automation interface, making them accessible to scripting languages. The ATL interface to OSAX may be used to expose multiple custom interfaces.
Type Description File

You define an OSAX object server by writing a type description file for processing by the `osgentyp` utility (see The `osgentyp` Utility on page 44). The type description file is a small source file written in a combination of C++ and idl. The name of the file has the `.ost` extension.

The type description file defines the COM interfaces and their implementations, and contains the following information:

- The name of the type library
- The names and CLSIDs (class IDs) of COM objects exposed with class factories
- The C++ classes exposed by the object server
- The COM interfaces to the C++ objects exposed by the object server
- The connection between the COM interfaces and C++ members and functions

The type description file begins with a description of the type library, followed by descriptions of the exposed objects. The exposed objects include a combination of the following:

- **Top-level objects** that are not tied to C++ classes and objects
- **Instances** that reference C++ objects
- **Instance containers** that can hold instances
- **Instance classes** that are related to C++ classes

Top-level objects and instance classes have no persistent state; instances are bound to C++ objects. For detailed information about these objects, see OSAX and ATL Concepts on page 49.

The following sections describe the different specifications to include in the type description file.

**Library Specification**

The type description file must contain a library specification with the following syntax:

```
[attributes] library name { library-members }
```

---

Chapter 2: Building OSAX Object Servers
Type Description File

**attributes**
Is a comma-separated list of the following:

**helpstring**(string)
Associates string with the generated type library. string specifies textual information about the OSAX object server and appears in browsers. This attribute is optional.

**isomorphic**(boolean)
Specifies the default value for library members. This attribute is optional. If it is not specified, the default is **true**.

**lcid**(lcid)
Specifies the locale ID—a language code for the type library. lcid is a 32-bit value identifying the language and a sort identifier. The first 12 bits of lcid are reserved by Microsoft. The next four bits contain the sort ID. The lower word identifies the language supported. This attribute is optional. If it is not specified, the locale ID defaults to **0x0000**.

**objectstore**
Specifies that the object server uses ObjectStore for persistence. This attribute is required; it causes the ObjectStore header files to be included.

**uuid**(uuid)
Associates uuid with this library. This attribute is optional. If it is not specified, osgentyp generates one. You can have osgentyp rewrite the .ost file with generated uuids for the missing ones by invoking it with the /u option (see The osgentyp Utility on page 44); or you can supply a GUID generated using the Create GUID tool in the Microsoft Developer’s Studio.

**version**(major[.minor])
Specifies the software version number, as assigned by the user. This attribute is required. If minor is not specified, the default is 0.

**name**
Is the name of type library (TLB) to be built.

**library-members**
Specifies the library and class information to be generated, consisting of object specifications (see Object Specification on page 21).
Example

The following is the library specification from \texttt{books.ost} in the Books example:

\begin{verbatim}
[  helpstring("ObjectStore OSAX books example 3.0"),
   lcid(0x0000),
   uuid(326D9EC0-4012-11D1-B9C3-0800091AAA11)
    version (3.0),
   objectstore
]
library booksObjectServer
{
   // Objects and classes to be defined here...
}
\end{verbatim}

These elements define a type library with the name \texttt{booksObjectServer}. The \texttt{helpstring}, \texttt{lcid}, \texttt{uuid}, and \texttt{version} attributes conform to standard Interface Definition Language (IDL). The \texttt{objectstore} attribute instructs \texttt{osgentyp} to link the object server with the OSAX libraries and storage system.

**Object Specification**

The type description file must contain object specifications with the following syntax:

\begin{verbatim}
   [attributes] class typename {method-specifications };
   [attributes] object coclass {method-specifications };
\end{verbatim}

**class typename**

Defines an OSAX class and instance class for the C++ \texttt{typename}. The instance class is defined as \texttt{typenameClass}, and the instance interface as \texttt{typename}. Instances are references to C++ objects of the specified type. You also tell \texttt{osgentyp} that all occurrences of \texttt{typename} should be exposed to the OSAX client using the instance interface.

**object coclass**

Defines \texttt{coclass} as a COM object and interface that is a top-level object. The top-level object includes methods and properties for accessing interfaces to ObjectStore and instances and instance classes.

**attributes**

Is a comma-separated list of the following:
appobject
Specifications that the object or instance class should be exposed as an appobject. Appobjects are automatically created when Visual Basic starts. Their methods appear at global scope. This attribute is optional and is meaningful only when specified with the object keyword.

appinstance
Specifications that the instance container should be exposed as an appobject. Appobjects are automatically created when Visual Basic starts. Their methods appear at global scope. This attribute is optional and is meaningful only when specified with the object keyword.

classinterface(name)
Specifies name for the class interface. This attribute is optional; if not specified, the default name for the class X is IXClass.

classuuid(uuid)
Associates uuid with the class interface being defined. This attribute is optional; if not specified, osgentyp generates a UUID. You can have osgentyp rewrite the .ost file with generated UUIDs for the missing ones; see the description of the /u command-line option in The osgentyp Utility on page 44.

conversion
Specifies that the object is implemented by another object server. You must specify the interface and classinterface attributes in the .ost file. The object server that implements the object must provide instance containers.

exception(hresult)
Catches and translates C++ exceptions (derived from the class exception) into ActiveX exceptions. The exception attribute takes one argument. This argument is the name of the hresult to be returned if a C++ exception is thrown. You can specify the exception attribute for the library, a class, or a method. This attribute is optional; if specified, it affects all contained methods.

factoryuuid(uuid)
Associates uuid with the class factory being defined. This attribute is optional and is meaningful only when defining a class factory; see the progid attribute. If this attribute is not
Chapter 2: Building OSAX Object Servers

specified, **osgentyp** generates one. You can have **osgentyp** rewrite the .ost file with generated UUIDs for the missing ones; see the description of the /u command-line option in The osgentyp Utility on page 44. For information about setting the UUID for the instance, see the **instancefactoryuuid** attribute.

**helpstring(string)**
Associates string with the generated class for use in browsers. This attribute is optional.

**instanceprogid**(DepName,IndepName)
Specifies that an instance container should be defined—that is, there will be a class factory for the instance. This attribute is meaningful only when defining an instance container. See the **progid** attribute for the meaning of DepName and IndepName.

**instancefactoryuuid**(uuid)
Associates uuid with the class factory of the instance container. This attribute is optional and is meaningful only when the **instanceprogid** attribute has also been specified. If this attribute is not specified, **osgentyp** generates one. You can have **osgentyp** rewrite the .ost file with generated UUIDs for the missing ones; see the description of the /u command-line option in The osgentyp Utility on page 44.

**interface(name)**
Overrides the default name for the interface for either an instance or server object. This attribute is optional.

**isomorphic(boolean)**
Overrides the default value set in the Library Specification on page 19. If boolean is true, the C++ object identity implies object identity for instance objects.

**progid**(DepName,IndepName)
Specifies that, in the class case, the instance container (or, in the object case, the top-level object) should have a class factory registered in the registry and associated with an application name. Two names are specified. **DepName** includes a version number. **IndepName** means “get the latest version.” The Visual Basic function **CreateObject**, as well as the **As New** modifier to a variable declaration, obtain objects through the class factory mechanism. See the **instanceprogid** attribute for information about creating a class factory for the instance.
uuid(uuid)
In the class case, associates uuid with the instance interface of this class. In the object case, associates uuid with the interface. This attribute is optional. If it is not specified, osgentyp generates one. You can have osgentyp rewrite the .ost file with generated UUIDs for the missing ones; see the description of the /u command-line option in The osgentyp Utility on page 44.

version(major[.minor])
Specifies the version number for the interface. For more information, see the version attribute for the Library Specification on page 19.

method-specifications
Specifies the method or property of the object specification, using the following syntax:

(attributes) return-type MethodName (argument-list) implementation;
attributes
Is a comma-separated list of any or none of the following:

class
Indicates that this method should be associated with the instance class interface rather than the instance interface.

exception(hresult)
Catches and translates C++ exceptions (derived from the class exception) into ActiveX exceptions. The exception attribute takes one argument, hresult, which is the value to be returned if a C++ exception is thrown. You can specify the exception attribute for the library, a class, or a method. When specified, it affects all contained methods.

helpstring(string)
Associates string with the generated class for use in browsers.

id(value)
Sets the method ID to value. Some OLE interfaces define method IDs explicitly. See the OLE documentation for further information.

propget
Specifies that the method is a property accessor. As far as the object server is concerned, there is no difference between properties and methods. However, in Visual Basic, a property
behaves like a data member. Properties that can be put are also allowed to appear on the left-hand side of an assignment.

**propput**
Specifies that the method is a property setter. See the OLE documentation for additional information.

**propputref**
Specifies that the method is a property setter. See the OLE documentation for additional information.

**restricted**
Specifies that the method is restricted. See the OLE documentation for additional information.

**return-type**
Is the C++ type of the value returned by the method. The `osgentyp` utility generates code that translates the C++ type to an ActiveX type if you have defined a class in the `.ost` file for the type; see The `osgentyp` Utility on page 44.

**MethodName**
Is the name of the method as it appears to the OLE client.

**argument-list**
Is a C++-style argument list of types and variables. The ActiveX interface uses ActiveX types for the C++ types that have classes defined for them in the `.ost` file. The `osgentyp` utility generates code that reverse-translates the ActiveX types to their C++ equivalents; see The `osgentyp` Utility on page 44. The variables in the argument list are available as C++ types for use in the method implementation.

**implementation**
Tells `osgentyp` how to implement the OLE method in C++. If no implementation is specified, `osgentyp` puts the method in the interface and class definition and expects you to provide the code. The following implementations are available. Note that `argument-list` (where specified) is optional.

**data name** `[[array-arguments]]`
If the method’s **return-type** is not void, this returns the value of the named member. If the method’s **return-type** is void, this sets the named member.
If the `class` attribute was specified, `name` is that of a static member. For the `object` case, it is a global variable.

When more arguments than are needed are specified for the method, they are passed as `array-arguments`. The final argument to a set method is always the new value.

Specify the order of `array-arguments` used for array access by following the `name` with the argument names. Separate the argument names with commas and enclose the list in brackets.

```plaintext
method name [(argument-list)]
```

The `method` implementation calls the specified C++ method on the associated C++ object. If the `class` attribute is specified, this calls the static member function.

The arguments in `argument-list` are passed to the C++ `method`. You control their order by following the name with `argument-list` in parentheses, as they would appear in a C++ method call.

```plaintext
function name [(argument-list)]
```

This method implementation calls the specified C++ function with the specified arguments. You can use the name `this` to refer to the associated C++ object. Methods can specify a namespace or class in the function name.

The arguments in `argument-list` are passed to the C++ `function`. You control their order by following the name with `argument-list` in parentheses, as they would appear in a C++ function call.

```plaintext
class name
```

Use this implementation to return the `class` object associated with one of your instance objects. The `return-type` should be a pointer to the appropriate interface type.

```plaintext
new [where] name [(argument-list)]
```

Use this method to allocate a C++ object specified by `name`. `where` is the database to allocate, and `argument-list` (if present) is the comma-separated list of constructor arguments.

**Example**

The following object specifications are from the Books example. The elements in the first example define an ActiveX object named `OSAXBooksServer` with a class factory implemented by the object server:
Chapter 2: Building OSAX Object Servers

helpstring("ObjectStore OSAX books example"),
uuid(326D9EC1-4012-11D1-B9C3-0800091AAA11),
version(3.0),
appobject,
progid("OSAX.Books.3","OSAX.Books"),
factoryuuid(326D9EC5-4012-11D1-B9C3-0800091AAA11), interface(IOSAXBooks)
]

object OSAXBooksServer

The class factory enables ActiveX clients and controllers to refer to the OSAXBooksServer object by name (for example, using CreateObject("OSAX.Books")). This name is specified by the progid attribute. The attribute takes both versioned and unversioned names. The appobject attribute specifies that this object is an appobject for controllers that support appobjects.

The elements in the next example define the top-level properties implemented by the object server:

{  
    [propget]
    IOSAXObjectStore* ObjectStore() class OSAXObjectStore;
    [propget] IBookEltClass* CBookElt() class BookElt;
    [propget] IBookClass* CBook() class Book;
    [propget] IAuthorClass* CAuthor() class Author;
    [propget] IOSAXStringConstructor* CString() class char;
    [propget]
    BSTR ObjectServerDirectory() function get_server_directory;
    }

The first element is the property named ObjectStore. The ObjectStore property supports the OLE interface IOSAXObjectStore, implemented by the C++ class named OSAXObjectStore. This interface and this class are part of the OSAX product. The second property, CBookElt, supports an interface representing the class BookElt. This property is used to type-check the first object retrieved from the database. The third property, CBook, is defined to return a class, so that its constructors and other static methods can be called. The CString property is defined to represent the constructor class for character strings. The ObjectServerDirectory property defines a method implemented by a global C++ function.

The following elements define the object server interface for the C++ class named Book:

[  
    uuid(326D9EC6-4012-11D1-B9C3-0800091AAA11),
    ]
The C++ objects of that book (stored in the database) are exposed in ActiveX as object references with two properties, **Name** and **Author**. **Name** uses an OLE interface that **osgentyp** automatically generates for the C++ type `char*` to represent the C++ data member named `name`. **Author** uses an OLE interface that **osgentyp** automatically generates for the C++ type `Author*` to represent the C++ data member named `author`. The **Book** class (exposed by the top-level property **CBook**) defines a constructor exposed as a class method named **Create**.

The interfaces for the C++ classes named **Author** and **BookElt** are defined similarly.
Accessing OSAX Objects

Every OSAX object has an associated context. The OSAX context holds the transaction state associated with an object. When a method of an OSAX object is invoked, OSAX makes the object’s context current. Any new OSAX objects that are created will use the current context as their context or, if there is no current context, create a new context.

Only one context can use ObjectStore at a time. With OSAX, using ObjectStore means running a method, or being in a transaction. In general, all of your objects will have the same context. The easiest way to ensure this is to have a master object in your object server. Your client creates the master object and then uses the master object to create all other objects. For example, in the Active Server Pages (ASP) application environment you can set a session variable to your top-level object so that all pages in the session run in the same context.

When you create an instance container, it will have its own context. However, when you initialize the instance container with an instance, the instance container’s context is changed to the context of the instance. Thus, you can safely use instance containers to hold instances. For example, in ASP you can initialize a session variable to an instance container for a database, and then store a database instance in the container.

An instance class is normally obtained from a method on the top-level server object with class implementation, so that it picks up the correct context. If an instance class is created with a class factory, you can write a SetContext method for it to put it in the proper context, as follows:

```c
void SetContext(IUnknown* pUnk) function SetContextImpl(pUnk);
```

Instance classes are given class-wide methods (such as constructors) or methods that return the extent of the class. Instance classes are also used as type objects for some methods that need to know how to associate an OSAX object with an arbitrary C++ pointer.

Instances are only obtained from OSAX. Internally, every instance and instance class in a context has a unique kernel class that obtains instances for C++ pointers. There are two kinds of kernel
classes: those with object tables and those without. Kernel classes with object tables are used for instances that reflect C++ identity. This is the default. When the isomorphic attribute of the instance is set to false in the .ost file, the kernel class will not have an object table, so every request for an OSAX instance for a C++ pointer returns a new OSAX instance.

You can use the following method to change the instance of an instance container:

```c
void SetValue(IUnknown* pUnk) function SetDataImpl(pUnk);
```
Creating Objects

An OSAX object server can provide methods to create new persistent or transient objects, in addition to accessing and modifying existing objects. Here are several ways to do this:

- Expose a method on one class that creates instances of another class.
- Expose a C++ static member function that creates instances of its class.
- Expose a C++ constructor that creates instances of its class.
- Expose another object. The new object will be allocated near where this object is allocated.

The following sections describe the different approaches to creating objects in more detail.

**Expose a Method on One Class That Creates Instances of Another Class**

In this approach, one C++ class (for example, `Bookshelf`) defines a member function that creates an instance of another C++ class. OSAX simply exposes this method on the corresponding types. Following is the C++ and `.ost` code for this approach:

```cpp
class Bookshelf {
public:
    os_List<Book*> books;
    Book *AddBook(char *title);
}
```

Given these definitions, a Visual Basic program could create a new book using syntax like the following:

```vbnet
Dim newBook As IBook
Set newBook = Bookshelf.AddBook(newTitle)
```

where the `Bookshelf` object is obtained at a higher level, perhaps as the value of a database root.

**Expose a C++ Static Member Function That Creates Instances of Its Class**

In this approach, a C++ class defines a static member function (for example, `Book::create()`) that creates an instance of that C++ class. The `[class]` attribute in the `.ost` file directs OSAX to expose that
Creating Objects

method on the class corresponding to the exposed type (that is, IBookClass). Following is the C++ and .ost code for this approach:

```
<table>
<thead>
<tr>
<th>C++ Definition</th>
<th>Definition in .ost File</th>
</tr>
</thead>
<tbody>
<tr>
<td>class Book {</td>
<td>class Book {</td>
</tr>
<tr>
<td>public:</td>
<td>[class]</td>
</tr>
<tr>
<td>static Book *create(char *title);</td>
<td>Book *Create(char *title) method</td>
</tr>
<tr>
<td>}</td>
<td>create(title);</td>
</tr>
</tbody>
</table>
```  

Every type exposed by OSAX also has an associated OSAX class. The latter two approaches expose methods on the class associated with an exposed type. The class is used for type checking and also for exposing static members or class methods such as constructors. For example, if you expose a C++ class named `Device`, it would normally be associated with two OLE interfaces, `IDevice` and `IDeviceClass`. Any instance members you wanted to expose would appear on `IDevice`. Any static members or constructors would appear on `IDeviceClass`.

Use the `[class]` attribute in the .ost file to expose class members. Use the `class` keyword to access the class. For example, the following .ost object specification syntax defines an object server with top-level properties to directly access the class objects for `Book` and `Author`:

```
object OSAXBookExample
{
    IOSAXObjectStore* ObjectStore() class OSAXObjectStore;
    [propget] IBookClass* CBook() class Book;
    [propget] IAuthorClass* CAuthor() class Author;
}
```

Given these definitions, a Visual Basic program could create and access a new book using syntax like the following:

```
Dim newBook As IBook
Set newBook = CBook.Create(newTitle)
MsgBox "The new book title is " & newBook.Name
```

**Expose a C++ Constructor That Creates Instances of Its Class**

In this approach, a C++ class defines a constructor (for example, `Book::Book()`) that creates an instance of that C++ class. The `[class]` attribute in the .ost file directs OSAX to expose that method on the class corresponding to the exposed type (that is, IBookClass). The `new` keyword makes the implementation of that method call the
C++ overloaded `new` operator for that type. Following is the C++ and `.ost` code for this approach:

```cpp
C++ Definition
class Author
{
    Author(char *n) { name = n; }
};

Definition in .ost File
class Author
{
    [class]
    Author* Create(IOSAXStorage* location, char *name) new(location)
    Author(name);
};
```

The `new` keyword used to expose constructors takes a single parameter that specifies where to allocate the new object. This parameter corresponds to the first parameter of the overloaded `new` operator of the ObjectStore C++ interface. It can be one of the following:

- (Usually) An object of type `IOSAXDatabase`, specifying a particular database in which to create the new object
- An object of type `IOSAXSegment`, specifying a particular segment within a database
- Nothing, which indicates a transient storage object

### Deleting Temporary C++ Objects

Ordinarily, persistent ObjectStore objects are not deleted when they are unreferenced by any ActiveX object. This behavior might not be appropriate for transient objects. To delete the C++ object when the reference count of the ActiveX object becomes 0, define a method like this in the application object:

```cpp
void SetCleanup(IUnknown* s) function DeleteDataOnFinalRelease(s, TRUE);
```

In the controller (for example, Visual Basic), you can use this method to cause an OSAX object to be deleted when its reference count goes to 0. To do this, call `SetCleanup()`, as in the following example:

```vbs
Dim X As MyObject
Set X = GetOneOfMyObjects
Call SetCleanup(X)
```

You can also define a method on your instance to mark it for deletion when on final release, as follows:

```cpp
void DeleteOnRelease() function DeleteDataOnFinalRelease(TRUE);
```
Accessing and Creating Character Strings

OSAX treats character strings specially to accommodate the different language semantics of C++ and ActiveX. In C++ and in ObjectStore, character strings are objects with unique identities. In ActiveX and Visual Basic, strings are considered literal values without identity. The OSAX object server effectively provides both behaviors. It exposes C++ `char*` strings as full-fledged object references of type `IOSAXString`. Instances of `IOSAXString` have identity and a default value property to coerce them to their literal ActiveX representations (Unicode BSTR).

Most Automation controllers, when given an `IDispatch` object in contexts requiring a string, will attempt to call the default value property to obtain the string value. This hybrid behavior is also suitable for exposing character pointers that are not actually strings.

In the `.ost` file, character strings are declared using the C++ `char*` data type, as in the following example.

```cpp
class Author {
    char *name;
    Book *book;
    Author (char* name);
    ~Author();
};

class Author {
    [propget] char* Name() data name;
    [propput] void Name(char* n) data name;
    [class]     Author* Create(IOSAXStorage *loc, char *n) new(loc)
                Author(n);
};
```

OSAX represents the character string values at run time using the interface `IOSAXString`. This interface has a default value property that converts the value to a literal Unicode BSTR. The following Visual Basic excerpt shows a character string property accessed both as a literal value and as an object.

```vbnet
Dim Author, Copy As IAuthor
MsgBox "The author's name is " & Author.Name
Set Copy.Name = Author.Name
```

The `MsgBox` statement requires a Visual Basic string for the concatenation of the message text. Therefore, the `Author.Name` property is automatically converted to a literal string. However, the `Author.Name` property is actually exposed as an object using the `IOSAXString` interface. This is so the property can be copied to
another object (for example, \texttt{Copy.Name}) without losing its identity. The result is that the persistent object representing \texttt{Copy} contains exactly the same \texttt{char*} character string as the persistent object representing \texttt{Author}.

**Creating Persistent Character Strings**

OSAX provides a mechanism for creating and initializing persistent and transient strings. OSAX uses either the ANSI or OEM code pages to translate from the Unicode representation used by ActiveX to the 8-bit native C++ representation of \texttt{char*}. The strings are exposed using the \texttt{IOSAXString} interface, and they are created using the \texttt{IOSAXStringConstructor} interface.

An \texttt{IOSAXStringConstructor} interface represents the class object for \texttt{IOSAXString}. It is typically exposed at the top level of an object server, as in the following example. This example \texttt{.ost} object definition provides class objects named \texttt{CAuthor} and \texttt{CString} for the \texttt{Author} objects described in Accessing and Creating Character Strings on page 34.

```cpp
object AuthorExample
{
  [propget] IOSAXObjectStore* ObjectStore() class OSAXObjectStore;
  [propget] IAuthorClass* CAuthor() class Author;
  [propget] IOSAXStringConstructor* CString() class char;
};
```

The \texttt{CString} class used in this example implements three methods:

- \texttt{IOSAXStringInterface::Ansi} creates a new string in the specified location and initializes it using the supplied ActiveX Unicode string translated to 8-bit representation using the ANSI code page.
- \texttt{IOSAXStringInterface::OEM} creates a new string in the specified location and initializes it using the supplied ActiveX Unicode string translated to 8-bit representation using the OEM code page. If you do not know whether to use \texttt{OEM} or \texttt{Ansi}, use \texttt{Ansi}.
- \texttt{IOSAXStringInterface::Copy} creates a new string in the specified location and initializes it using the supplied \texttt{IOSAXString}, which is already in 8-bit form.
Accessing and Creating Character Strings

The following Visual Basic function uses these class objects to create a new persistent Author object. It includes a persistent character string to represent the author name.

Function CreateAuthor(Name As String) As IAuthor
    Dim osName As IOSAXString
    Set osName = CString.Ansi(Name, osDatabase)
    Set CreateAuthor = CAuthor.Create(osDatabase, osName)
End Function
Accessing Collections

To expose ObjectStore collections defined using the template classes `os_Set, os_Bag, os_List, and os_Dictionary` through OSAX, use the instantiated template class as a normal C++ type. You can use the template classes to specify the type of any argument or return value, as in the following example:

### C++ Definition

```cpp
class Bookshelf {
public:
    os_List<Book*> *books;
}
```

### Definition in .ost File

```cpp
class Bookshelf {
    [propget] os_List<Book*> *Books() data books;
}
```

At run time, all ObjectStore collections are represented using the `IOSAXCollection` interface. The `IOSAXCollection` interface provides methods for item lookup, insertion, and removal, as well as iteration and query.

For an example of an application that accesses collections, see Chapter 5, The Portfolio Sample Application, on page 77.

### Defining a Collection

To define and expose a collection class in the OSAX .ost type description file, use standard techniques. However, since template instantiation does not exist in ActiveX, you must provide nontemplated names for the class.

The following example shows how to expose a collection class in the OSAX .ost type description file, along with its element type. Note that the object declaration provides an accessor `CBook` for the element type, and `CBookList` for the collection type. These classes are defined later. The accessors enable a client to call static members and constructors.

```cpp
object Library {
    [propget] IOSAXObjectStore* ObjectStore() class OSAXObjectStore;
    [propget] IOSAXStringConstructor* CString() class char;
    IBookClass* CBook() class Book;
    IBookListClass* CBookList() class os_List<Book*>;
};
```
Accessing Collections

The following declaration for the Book class defines an ActiveX interface for it, in the standard manner. Note that the Create() class method is connected to the C++ constructor for the Book class.

class Book
{
    [propget]
    char* Author() { data author; }
    [propget]
    char* Title() { data title; }
    [class]
    Book* Create(IOSAXStorage* db, char* title, char* author) new (db) Book (title, author);
};

The following declaration for the instantiated collection class os_List<Book*> defines an ActiveX interface for it. OSAX provides the ActiveX interface from the interface for the os_Collection base class. The classinterface() and interface() attributes are used to specify appropriate names for use in the ActiveX domain. A Create class method is defined to provide a type-safe constructor based on the static C++ member os_List::create().

[ classinterface(IBookListClass),
  interface(IBookList) ]
class os_List<Book*>{
    [class]
    os_List<Book*> & Create(IOSAXStorage* db) method create (db);
};

Creating a Collection of Objects

The following Visual Basic excerpt shows how to use the interfaces defined in the hypothetical Library object server to create a collection of Book objects. The CreateLibrary routine uses the Create() class method to construct a new CBookList collection in the specified database. This routine also assigns the Create class method to a database root named "Books". Several Books are then created and inserted into the collection.

Sub CreateLibrary()
    Set Root = osDatabase.CreateRoot("Books")
    Set Root.Value = CBookList.Create(osDatabase)
    Set Root.Type = CBookList
    Set Library = Root.Value
The following `CreateBook` routine uses the `Create()` class method to construct a new `CBook` object in the specified database. The two arguments are copied from Visual Basic into the database. The standard ANSI code page is used for character conversion.

```vba
Function CreateBook(Title As String, Author As String) As IBook
    Dim osTitle, osAuthor As IOSAXString
    Set osTitle = CString.Ansi(Title, osDatabase)
    Set osAuthor = CString.Ansi(Author, osDatabase)
    Set CreateBook = CBook.Create(osDatabase, osTitle, osAuthor)
End Function
```

### Iterating over Collections

The `IOSAXCollection` interface provides an implementation of the standard COM iteration protocol (IEnumVARIANT), so the normal iteration syntax of the hosting environment can be used. In Visual Basic, the iteration syntax is the `For Each...Next` statement, as in the following example:

```vba
For Each Book in MyBookShelf.Books
    Print Book.Title
Next Book
```

### Querying Collections

The `IOSAXCollection` interface allows a collection to be queried. Use the ObjectStore collection query language documented in the *ObjectStore C++ API User Guide*.

The `IOSAXCollection::Evaluate()` function takes one argument, a string representing the query expression, as described in the *ObjectStore C++ API User Guide* for `os_Collection::query`. The string is of the form

```
"type:query-expression"
```

For example:

```vba
Set ResultSet = ASetInstance.MemberSet.Evaluate("A*:Num == 111")
```

Or, using Visual Basic syntax extensions

```vba
Set ResultSet = ASetInstance.MemberSet[A*:Num == 111]
```
Accessing Collections

The result is a transient collection of elements of the specified type (A*).

Deleting a Collection

A collection of elements (but not the elements) is deleted when the last reference to the OSAX object is released.
Chapter 2: Building OSAX Object Servers

Creating and Using Database Roots

Database roots are explicitly named and explicitly typed objects. They act as entry points to the contents of the database. Typically, a database has one database root. Sometimes, a database has several database roots. The database roots contain objects with properties or methods that lead to other objects in the database, such as a collection or list. Database roots are typed and their values are dynamically type-checked. This is so that all access to persistent objects in the database is type-safe.

You can create a database root two ways: use the ObjectStore C++ API in C++, or use an object server using the exposed `IOSAXDatabaseRoot` interface. The following excerpt from the Books example shows the creation of a database. It contains one root named `Books` intended to hold a list of books built of `CBookElt` objects. Initially, the value of the root is empty (Nothing in Visual Basic, NULL in C++).

```vbnet
Set osDatabase = ObjectStore.CreateDatabase(cFilename.filename)
ObjectStore.BeginTrans
    Dim Root As IOSAXDatabaseRoot
    ' Create the database root, representing an empty list of books
    Set Root = osDatabase.CreateRoot("Books")
    ' Set root type before initializing root value
    Set Root.Type = CBookElt
    Set Root.Value = Nothing
CreateBookList
ObjectStore.CommitTrans
```

Once a database root has been created, you can access and update its value using the `Value` property of the `IOSAXDatabase` interface. The following excerpt from the Books example shows the insertion of a new `Book` at the head of the list it maintains, using the `Books` database root.

```vbnet
Sub InsertBook(Book As IBook)
    Dim osElement As IBookElt
    Dim Head As IBookElt
    Set Head = osDatabase.Value("Books", CBookElt)
    Set osElement = CBookElt.Create(osDatabase, Book, Head)
    Set osDatabase.Value("Books", CBookElt) = osElement
End Sub
```
Object Server Exceptions

When you perform certain operations in C++, such as trying to open a database that does not exist, ObjectStore signals an exception. The exception might be a C++ exception or a TIX exception, depending on the storage system and the cause of the exception. (ObjectStore uses its own exception mechanism that predates C++ exceptions.) Your OSAX object server must handle the exception, because the ActiveX client cannot interpret the error otherwise.

Signaling Exceptions from an Object Server

C++ exceptions (derived from the class exception) can be caught and translated into ActiveX exceptions. To do this, use the exception attribute in the .ost file. The exception attribute takes one argument, which is the name of the hresult to be returned if a C++ exception is thrown. The exception attribute can be specified for the library, a class, or a method. When specified, it affects all contained methods.

For example, if an exception is thrown in the following method’s implementation:

```c++
[exception(E_INVALIDARG)]
void SetName(char const * name)
```

then an ActiveX exception is created with the description of the exception, and E_INVALIDARG is returned from the method.

Handling Exceptions from an Object Server

The osgentyp utility inserts macros for general exception handlers in each of your methods. These macros interpret the C++ or TIX exception and convert it to an OLE Automation error. If the error received by an ActiveX client came from ObjectStore, it will have one of the following values, which can be expressed as either a symbolic constant or a hexadecimal value:

**OSAXETxnAbort (&H80041000)**

An exception occurred that requires that the current transaction be aborted and should be retried. For example, if your application and another application were both trying to modify data on the same page in the database and the other
application committed the transaction first, your application should just retry the entire transaction. OSAX has finished aborting the transaction. You do not need to perform any ObjectStore-specific cleanup.

**OSAXETixError (&H80041001)**
An unspecified exception has occurred. Look at Visual Basic’s error object to determine what action to take.

**OSAXETixFatal (&H80041002)**
A fatal exception occurred and ObjectStore is unusable.

In addition to these exceptions, you could receive any of the exceptions that are normally raised by ActiveX.

You can obtain additional information about any exception from the error string.

### Transactions and exceptions
When an exception occurs during a transaction, you need to roll back the transaction. At the same time, you should roll back any relevant transient state associated with the transaction. Then you can retry the transaction.

You might find it easiest to structure the work done during a transaction in a subroutine. This subroutine is called by a function that sets up the transaction and an error handler:

```vba
Private Sub TransactionExample()
    Dim count As Integer
    count = 0
    On Error GoTo Handle
    os.BeginTrans
    Call TransactionBody
    os.CommitTrans
    Exit Sub
Handle:
    If Err.Number = OSAXETxnAbort Then
        count = count + 1
    End If
    If count < 10 Then Resume
    Err.Raise (Err.Number)
End Sub
```
The osgentyp Utility

The osgentyp utility reads a type description file and generates the C++ and Interface Definition Language (IDL) source code required to implement the object server and its associated type library. osgentyp is typically invoked from a custom build rule attached to a type description file in a Visual C++ project. When you use the ObjectStore ActiveX AppWizard to create your projects, the AppWizard creates the custom build rules for you.

Command-Line Syntax

The command-line syntax for invoking osgentyp is:

```
osgentyp option-list
```

where `option-list` is a space-separated list of any of the following:

- `/c files`
  Specifies one or more files, separated by spaces, to be inserted in `#include` directives in the generated file.

- `/dt file.ost`
  Specifies the name of the type description file. If this option is not specified, then the name of the type description file is taken from the argument to the `/l` option.

- `/i file.cpp`
  Specifies the name of the generated C++ file that implements the server. If this option is not specified, then the name of the C++ file is taken from the argument to the `/l` option.

- `/ih file.h`
  Generates class definitions in a separate header file. Use this option if you are implementing a method by hand. If this option is not specified, then the name of the header file is taken from the argument to the `/l` option.

- `/l name`
  Specifies the language-independent component of the type library name. If your type description, DLL, C++, and TLB files all have the same name, you need only specify the `/l`; `name` will be supplied as the default for any missing options. Note, however, that you must specify the `/c` option for any include files.

ActiveX Interface for ObjectStore
Chapter 2: Building OSAX Object Servers

/t file.idl
Specifies the generated .idl file name that is input to the midl compiler. If this option is not specified, then the name of the .idl file is taken from the argument to the /l option.

/u
Rewrites the .ost file with generated UUIDs for any that are missing. All C and C++ style comments (/ / ... and /* ... */) will be removed from the rewritten file.

Generating UUIDs for attributes

The osgentyp utility can automatically generate the UUIDs for the various uuid, classuuid, and factoryuuid attributes used in the type description file. Typically, these UUIDs are assigned when the interface in question is first created, and are rarely changed thereafter. Letting osgentyp manage UUIDs simplifies the administration of the object server by preserving its interface registration entries in the Windows system registry.

To cause osgentyp to generate UUIDs, invoke it with the /u option, as described earlier. The /u option causes osgentyp to read your type description file, generate new UUIDs for every interface, and write the file back out with the new UUIDs in place. Note, however, that any comments you have added to the file and any formatting changes you have made are removed as part of this process. If you want to preserve comments and formatting, rename the file before using the /u option and then manually make changes to the file afterwards. If the name of the type description file is books.ost, the following command line updates the file with UUIDs:

osgentyp /dt books.ost /u books.ost

Once you have inserted the UUIDs in this way, you can make future changes to the file and its interfaces and preserve their registration identity through the development cycle. If you add classes or substantially modify existing interfaces, you can specify the /u option again to supply new UUIDs.
The osgentyp Utility
Chapter 3
Using OSAX with Microsoft ATL

The Microsoft Active Template Library (ATL) is a set of template-based C++ classes that you can use to create small, fast Component Object Model (COM) objects. OSAX supports ATL as a way to create COM interfaces for persistent objects stored in ObjectStore. The ATL approach is an alternative to the approach described in Chapter 2, Building OSAX Object Servers, on page 13, which is based on a type description file. Using ATL requires more C++ expertise, but ATL provides more flexibility for customizing COM interfaces.

When using OSAX with the ATL interfaces, you define COM interfaces using normal ATL techniques, classes, and interfaces. You then connect them to ObjectStore objects using template classes provided by the ActiveX interface. The resulting control is linked with the ObjectStore and OSAX run-time libraries.

The following facilities are described in this chapter:

• Handlers for ObjectStore storage system faults on persistent data
• Handlers for exceptions signaled by the ObjectStore storage system
• Template classes for defining OSAX instances and instance classes

An ATL-defined control that uses ObjectStore must be linked with osax.lib. The type description file and osgentyp are not needed.
However, `osgentyp` produces code that uses the same interfaces as described in this chapter and can be used to generate initial code for an ATL-based project. See Type Description File on page 19 for information about type description files and The osgentyp Utility on page 44 for information about `osgentyp`. 
OSAX and ATL Concepts

The following sections define the terms and concepts used in this chapter.

**Terminology**

The following terms are used to describe OSAX and ATL:

- **COM object**—An object exposed through COM interfaces.
- **COM class**—A class (COM) that a COM client uses to create a COM object.
- **ATL objects**—COM objects implemented with ATL.
- **ATL class**—A C++ class that implements a COM object using ATL.
- **OSAX class**—A C++ class that implements a COM object using ATL and OSAX.
- **OSAX objects**—COM objects implemented with OSAX.
- **C++ class**—A C++ class that implements a C++ object referenced by OSAX.

**OSAX Object Servers**

A typical OSAX object server consists of a top-level object with methods that are used to access the server’s other objects. The top-level object has a COM class. The OSAX client creates the top-level object with the class factory for the COM class. The top-level object establishes the context for the other objects, so that all of the client’s objects execute in a single context.

An OSAX object server is the same as a standard ATL object server, except that a different module class is used, additional header files are needed, and there is an additional library. An OSAX object server may be a DLL file, an .exe file, or a Windows NT service.

**OSAX Classes**

OSAX classes use a threading model of Both. They are ATL classes with the additional public base template classes CExceptionTranslator and IOSAXContextInformationImpl. In
addition, the interface IOSAXContextInformation must be added to the interfaces listed in the COM_MAP section of the class.

**OSAX Methods**

OSAX methods are ATL methods with their bodies wrapped by macros that establish exception translation, fault-handling, and context management. The method bodies are preceded by `OSAX_BEGIN_METHOD` and followed by `OSAX_TRANSLATE_EXCEPTIONS` and `OSAX_END_METHOD`. The macro `OSAX_END_METHOD` includes a `return`, so it requires no explicit `return`. A method looks like this:

```cpp
STDMETHODIMP CAuthor::get_Name(IOSAXString * * pVal)
{
    OSAX_BEGIN_METHOD(IID_IAuthor)
    // method body
    OSAX_TRANSLATE_EXCEPTIONS
    OSAX_END_METHOD
}
```

**OSAX Instances**

OSAX instances and their related classes are the most important OSAX objects. OSAX instances are fixed references to C++ objects that are valid across transactions. OSAX instances are created only by OSAX. When the C++ objects are persistent, OSAX instances hold the transient state associated with the persistent object. For example, an OSAX instance holds the reference count, because the reference count is only used transiently and must be available even when a transaction is not in progress.

For a client, an OSAX instance container is like an OSAX instance, except that the reference to the C++ object can be changed.

OSAX instances and instance containers are both defined by a single OSAX class. If the OSAX class supports a class factory, then the class factory will create instance containers. In addition, the normal ATL methods, such as `CreateObject`, can be used to allocate instance containers. OSAX methods call the internal `SetDataImpl` method to change the C++ object being referenced. `SetDataImpl` will fail if called on an instance instead of an instance container.
The following illustrates OSAX instances:

![OSAX Instance Classes Diagram]

**OSAX Instance Classes**

Every OSAX instance must have an associated OSAX instance class. The same OSAX class may be used for both, or one OSAX class may be used for each. The `osgentyp` utility produces separate OSAX classes for the instance and instance class; see The `osgentyp` Utility on page 44.

An OSAX instance class is an object that supports constructors and methods applicable to the entire C++ class, rather than to specific objects of the class. The OSAX class for the OSAX instance includes the public template base `CInstanceImpl` and adds the interface `IOSAXInstanceInformation`, in addition to the standard additional OSAX base classes and interfaces. Instance implementations access the referenced C++ object through a member called `data`. `data` behaves like a pointer to the referenced C++ object. The public nested class `CDataPtr` implements `data`. The `CDataPtr` has a number of useful constructors that let it be used to access the referenced C++ object from the methods of other OSAX objects, and to obtain OSAX instances for C++ pointers.

The OSAX class for the instance class includes the public base template class `CInstanceClassImpl` and exposes the additional interface `IOSAXType`. The top-level server object normally includes methods that return an object for each of the instance classes.
Implementing an Object Server with ATL

The following procedure implements an OSAX object server directly in ATL:

1. Use the ATL COM AppWizard to make a skeleton ATL project.
2. Make the project-level changes to enable OSAX.
3. Add ATL classes.
4. Add base classes, interfaces, and so on to the class definitions to convert them to OSAX classes, OSAX instances, and OSAX instance classes.
5. Add methods using the ATL class wizard.

The following sections explain each step in more detail, using a variation of the Books example; see A Tour of the Books Example on page 8. The project for the Books example (see the Visual C++ project file c:\odi\osax\examples\steps\stepn\books.dsw) shows the result at the completion of each of the following steps. In the pathnames in the following descriptions, replace Books with the name of your server.

**Step 1. Use the ATL COM AppWizard.**

Use the ATL COM AppWizard to create the skeleton ATL project. In the example for this procedure, the server is implemented as a dynamic link library (DLL).

**Step 2. Make Project-Level Changes.**

For each configuration you build, make the following changes:

- Under the C/C++ tab in Project Options, add /EHs to enable exception asynchronous handling. This change is needed to make ObjectStore’s fault handler work.
- In the Preprocessor category, change _DEBUG to _OSAXDEBUG in all configurations that define it. Remove _ATL_MIN_CRT from any configurations that have it. For all configurations, add _ODI_FORCE_OSTORE_LIB.
- Under the Code Generation category, specify the run-time library Multithreaded DLL for the release version. You can use the Debug Multithreaded DLL run-time library for debugging, but the Multithreaded DLL is more reliable.
You can use the debug run-time library during debugging. To do so, change `_OSAXDEBUG` to `_DEBUG` and select the Debug Multithreaded DLL run-time library in the Code Generation category. You will continue using the Multithreaded DLL run-time library for the C++ operators `new` and `delete`.

Under the Link tab in category General, add `ostore.lib` to the front (left side) of the Object/library modules.

In the `stdafx.h` file, make these changes:

- Change:
  ```
  #include <atlbase.h>
  ```
  to:
  ```
  #include <osax/osaxbase.h>
  ```

- Change:
  ```
  extern CComModule _Module;
  ```
  to:
  ```
  extern OSAX::COSAXModule _Module;
  ```

- Following the line:
  ```
  #include <atlcom.h>
  ```
  add the lines:
  ```
  #include <osax/osax.h>
  using namespace OSAX;
  ```

In `books.cpp`, change:
```
CComModule _Module;
```

to:
```
OSAX::COSAXModule _Module;
```

Save the changes to your project.

**Step 3. Add ATL Classes.**

You will need a top-level object that clients create to access your server objects and objects that represent your C++ objects. For the sake of illustration, the example in this procedure parallels the Books example that uses the type description file, except that for
each C++ class we combine the ATL classes for the OSAX instance and the OSAX instance class. We also add one class to simplify the *Books* collection, resulting in five ATL classes.

For the top-level object, we use the name *OSAXBooks* and a ProgID of *OSAX.ATLBooks*. We use slightly different interfaces from those used in the type description version of the Books example. Under attributes, we select a threading model of *Both* and support *ISupportErrorInfo*. Aggregation won’t be supported, and the dual interface is optional. We use the same settings for the other objects.

**Step 4. Add base Classes, Interfaces, and So On.**

Every OSAX class needs to include the template base classes *CExceptionTranslator* and *IOSAXContextInformationImpl*. *CExceptionTranslator* implements the translation of TIX and C++ exceptions into COM exceptions. Its template argument is the ATL class. *IOSAXContextInformationImpl* supplies the context information that manages sessions with the storage system. Its template argument is the ATL class. Each of these classes must add the following to the `COM_MAP` section of the class definition:

```cpp
COM_INTERFACE_ENTRY(IOSAXContextInformation)
```

Every ATL class that implements an OSAX instance class needs to include the template base class *CInstanceClassImpl*. Since this example uses one ATL class to implement each instance and instance class, each of these classes requires *CInstanceClassImpl*. It takes two template arguments, the ATL class that implements the instance class, and the ATL class that implements the instance. In the Books example, these are the same.

The template base class implements the interface *IOSAXType*, so you must add the following to the `COM_MAP` section of your classes:

```cpp
COM_INTERFACE_ENTRY(IOSAXType)
```

Also, because *IOSAXType* is a dispatch interface, you must modify the entry for *IDispatch* to tell it which *IDispatch* interface to use for your class, by using:

```cpp
COM_INTERFACE_ENTRY2(IDispatch,IMyClass)
```
where \texttt{IMyClass} is the dispatch interface you want to serve as the default \texttt{IDispatch} interface.

Each instance class should add the following to its class definition:

\begin{verbatim}
OSAX_CLASS_INTERFACE(IInstanceClass)
\end{verbatim}

The area above the \texttt{COM_MAP} is a good place to add macros like this. The macro defines an accessor for the instance class.

Add the following to the \texttt{COM_MAP} section of your class:

\begin{verbatim}
COM_INTERFACE_ENTRY(IOSAXInstanceInformation)
\end{verbatim}

Each instance class should also add

\begin{verbatim}
OSAX_TYPE_NAME("C++ type")
\end{verbatim}

This returns the name of the type, which can be used to obtain an \texttt{os_typespec} for persistent allocation. Use of \texttt{OSAX_TYPE_NAME} is optional.

Every ATL class that implements an OSAX instance needs to include the template base class \texttt{CInstanceImpl}, which takes three arguments:

- The ATL class that implements the instance class
- The ATL class that implements the instance
- The type of the C++ data

In this procedure, the first two arguments are the same. You also need to include the header files for your C++ classes. In this example, include directives for the header files were inserted in \texttt{stdafx.h}.

Each instance class should add the following to its definition:

\begin{verbatim}
OSAX_INTERFACE_CONVERSION(IInstance)
\end{verbatim}

The part of the class definition near the \texttt{COM_MAP} is a good place for this. \texttt{OSAX_INTERFACE_CONVERSION} defines a conversion from the ATL class to the specified interface. This conversion is used when writing methods.

The ATL class definitions for your instance classes should add:

\begin{verbatim}
DECLARE_GET_CONTROLLING_UNKNOWN()
\end{verbatim}

You can add it just before the \texttt{COM_MAP} section.
Finally, you must include the header files for your C++ classes since the ATL instance classes need to see the types. In this example, the file `persistent.h` was added to the project.

Add the following line to the `.idl` file:

```
import "osaxint.idl";
```

**Step 5. Add Methods.**

You can use Class View to add a default property called `Name` to the `IAuthor` interface. The class wizard adds the method and the class definition to the `.idl` file, and generates a stub method body. In this example, the property’s value is an `IOSAXString*`.

The `Name` method accesses the `char*` value of the data associated with the `IAuthor` interface pointer. The method’s implementation must first access the C++ `Author*`, then get the `char*` name, get an `IOSAXString*` for the `char*`, and return an appropriate `HRESULT`. At the same time, any exceptions from ObjectStore must be translated to COM exceptions, and the ObjectStore fault handler must be enabled.

**Fault-handling macros**

OSAX provides three macros to perform fault-handling and exception translation. At the beginning of your method you use `OSAX_BEGIN_METHOD`, which needs the interface IID as an argument. Your method body follows, and then you must provide the macros `OSAX_TRANSLATE_EXCEPTIONS` and `OSAX_END_METHOD`. `OSAX_END_METHOD` returns an appropriate `HRESULT`, so you do not need to do anything special to return a value from the method. The method looks like this:

```
STDMETHODIMP CAuthor::get_Name(IOSAXString ** pVal)
{
    OSAX_BEGIN_METHOD(IID_IAuthor)
    OSAX_TRANSLATE_EXCEPTIONS
    OSAX_END_METHOD
}
```

**Smart pointer**

OSAX provides a member variable called `data` that is a smart pointer that acts like a pointer to the data referenced by the OSAX object. To access the `m_strName` member of the object, all you need to write is

```
data->m_strName
```
Obtaining OSAX instances

The OSAX instance class implementation provides useful methods for obtaining OSAX instances for C++ values. The OSAX_INTERFACE_CONVERSION macro defines a conversion function from the ATL class to an interface pointer. C++ constructors and conversion operators permit the conversion function to work directly on the C++ object to be referenced by the OSAX instance. The class OSAX::COSAXString implements IOSAXString, and its GetInterfacePtr member takes a char* and an IOSAXString** as arguments and returns an HRESULT. If it succeeds, the contents of the IOSAXString** argument are set to the OSAX instance for the char* pointer. The complete get_Name method looks like

STDMETHODIMP CAuthor::get_Name(IOSAXString ** pVal)
{
    OSAX_BEGIN_METHOD(IID_IAuthor)
    return COSAXString::GetInterfacePtr(data->m_strName, pVal);
    OSAX_TRANSLATE_EXCEPTIONS
    OSAX_END_METHOD
}

The methods for CAuthor, CBook, and CBookElt are all similar.

The OSAX_CLASS_INTERFACE macro defines the accessor GetClassInterfacePtr, so the method just needs to call it (the object server in the Books example exposes only the BookElt instance class):

STDMETHODIMP COSAXBooks::get_BookEltClass(IBookElt ** pVal)
{
    OSAX_BEGIN_METHOD(IID_IOSAXATLBooks)
    return CBookElt::GetClassInterfacePtr(pVal);
    OSAX_TRANSLATE_EXCEPTIONS
    OSAX_END_METHOD
}

Top-level server object

The top-level server object needs a method to return an IOSAXObjectStore interface pointer. OSAX provides a function for this, so the method is defined as

STDMETHODIMP COSAXBooks::get_ObjectStore(IOSAXObjectStore ** pVal)
{
    OSAX_BEGIN_METHOD(IID_IOSAXATLBooks)
    return GetOSAXObjectStore(pVal);
    OSAX_TRANSLATE_EXCEPTIONS
    OSAX_END_METHOD
}
Implementing an Object Server with ATL

In the file *stdafx.h*, add the following just before the include directive for *osax.h* to enable ObjectStore-specific definitions in *osax.h*:

```
#include <ostore/ostore.hh>
```

In the header file *persistent.h*, add the following to each of the C++ classes:

```
static os_typespec* get_os_typespec()
```

Finally, add a custom build rule for *ossg*. To add the custom build rule, add a text file to the project. In this example, the name of the text file is *schema.scm*. Under Project Settings, set the settings for All Configurations, and select the file *schema.scm*. Add the following *ossg* line:

```
ossg -asdb $(OutDir)\schema.adb -asof $(IntDir)\schema.obj   $(InputDir)\$(InputPath) /I $(InputDir) /MD /D_DLL /DWIN32
```

For the targets, use the following:

```
$(IntDir)\schema.obj   $(OutDir)\schema.adb
```

### Allowing more than one OSAX object server

The schema file lists the C++ types in your database. If you want to allow more than one OSAX object server to be used at a time, mark the schema file as a DLL schema by including

```
OS_SCHEMA_DLL_ID(id)
```

where *id* can be in either of the following:

- **DLL:** *dllname*
  - *dllname* is the name of your DLL file. It should be in your path at the time an application needs to use the schema information.

- **OSAX:** *componentname*
  - *componentname* is the name of your component. The component does not need to be in your path but it must have been registered.

ObjectStore uses *id* to load the DLL when an object defined by *dllname* or *componentname* is referenced.

### Creating an empty book list

The server has a top-level object method that creates an empty book list, represented by a *BookElt* that has a **NULL Book** and a **NULL** next element in the list.

### Defining a nested base class

The *CinstanceImpl* template base class used for OSAX instances defines a nested base class called *CDataPtr*. The *CDataPtr* class is the class of the smart pointer called *data*. It has constructors that
initialize it from variants, C++ pointers, and interface pointers. In this example, you have an $\text{IOSAXDatabase}^*$ interface pointer, and need an $\text{os\_database}^*$. $\text{COSDatabase}$ is the class that implements $\text{IOSAXDatabase}$ for ObjectStore databases, so you can use the $\text{CDataptr}$ to get a smart pointer and the smart pointer can be converted to an $\text{os\_database}^*$, as follows:

$$\text{os\_database}^* \ pDatabase = \text{COSDatabase}\::\text{CDataptr}(pDb);$$

The complete method is

```cpp
STDMETHODIMP COSAXBooks::CreateBookList(IOSAXDatabase * pDb, BSTR bstrName, IBookElt * * ppBookElt)
{
    USES_CONVERSION;
    OSAX_BEGIN_METHOD(IID_IOSAXATLBooks)
    os_database* pDatabase = COSDatabase::CDataptr(pDb);
    os_database_root* pRoot = pDatabase->create_root(OLE2A(bstrName));
    BookElt* pBookElt = new(pDatabase,BookElt::get_os_typespec()) BookElt;
    pRoot->set_value(pBookElt,BookElt::get_os_typespec());
    return CBookElt::GetInterfacePtr(pBookElt,ppBookElt);
    OSAX_TRANSLATE_EXCEPTIONS   OSAX_END_METHOD
}
```

Overloadings of new operator

OSAX provides special overloadings of the new operator that can be used with $\text{IUnknown}$ and $\text{VARIANT}$ placement arguments. The object server in the Books example exposes a method for creating a new Author in a segment or database, passed in an optional $\text{VARIANT}$ pointer.

The method definition is

```cpp
STDMETHODIMP COSAXBooks::CreateAuthor(BSTR bstrName, VARIANT where, IAuthor * * ppAuthor)
{
    USES_CONVERSION;
    OSAX_BEGIN_METHOD(IID_IOSAXATLBooks)
    // Copy the ANSI version of the name to its permanent location
    char* strNameIn = OLE2A(bstrName);
    ULONG nLengthName = strlen(strNameIn)+1;
    char* strNameAuthor = new(where,COSAXString::InstanceClass(), nLengthName) char[nLengthName];
    strcpy(strNameAuthor,strNameIn);
    // Allocate a new Author
    Author* pAuthor = new(where,CAuthor::InstanceClass()) Author(strNameAuthor);
    return CAuthor::GetInterfacePtr(pAuthor,ppAuthor);
    OSAX_TRANSLATE_EXCEPTIONS OSAX_END_METHOD
}
```
Implementing an Object Server with ATL

}
Class factories can be used by ActiveX controllers to create OSAX instances. When a class factory creates an OSAX instance, the instance is not initialized. OSAX instance initialization methods can be used to set the instance to a new or existing C++ object.

In the following example, Aut is an uninitialized Author OSAX instance. The Storage property specifies where the persistent state is allocated.

```vba
Dim Aut As New Author
Set Aut.Storage = MyDatabase
Aut.Name = "Samuel Clemens"
```

You can also use other approaches. In the following example, an OSAX instance is initialized to another OSAX instance:

```vba
Dim Aut1 As New Author
Aut1 = Aut
```

Notice that it is not

```vba
Set Aut1 = Aut
```

which would change the value of the variable Aut1. Here, Author has a default value setter that is defined to set the reference.

A constructor style can be used as follows:

```vba
Dim Aut As New Author
Aut.Create(MyDatabase,"Samuel Clemens")
```

An alternative approach is to have a separate OSAX class object:

```vba
Dim Aut As Author
Set Aut = AuthorClass.Create(MyDatabase,"Samuel Clemens")
```

**OSAX Instances for Existing C++ Objects**

OSAX methods return OSAX instances that refer to C++ objects. You can set an object variable to the result of the method. If the class supports assignment, you can also set the C++ reference of an existing object to the result:

```vba
Dim Aut As Author
Dim Aut1 As New Author
'Set Aut variable to the returned object
Set Aut = Writers.GetAuthor("Elbert Hubbard")
'Set C++ reference of Aut1 to the returned object
```
Creating OSAX Instances

\textbf{Aut1 = Writers.GetAuthor("Nicolas Bourbaki")}

Why would you want to use one approach instead of another? If you are using the Microsoft Transaction Server (MTS), your objects may be under its control. You can initialize them by setting their C++ references to another OSAX instance, similar to the way you initialize other objects by loading them from a file.
Object Deletion

A class can support object deletion. When an OSAX instance is marked for deletion, the referenced C++ object is deleted on final release. See `DeleteDataOnFinalRelease` in Methods of the `CInstance::CDataPtr` Class on page 66.
OSAX Methods

The general format for an OSAX method is

```
OSAX_BEGIN_METHOD(IID)
    // method body
[optional exception translators]
OSAX_TRANSLATE_EXCEPTIONS
OSAX_END_METHOD
```

The argument `IID` should be the interface ID (IID) of the interface implemented by the method. These macros enable the ObjectStore fault handler, perform synchronization, establish appropriate transaction context, and translate C++ and TIX exceptions into ActiveX exceptions.

Exceptions

Exceptions are translated via translate methods defined by the base class `CExceptionTranslator`. The translate methods extract appropriate information from the exception and call `AtlReportError`. You can define your own translate methods in your class if the default behavior is not appropriate.

Optional Exception Translators

The optional exception translators can be used to translate exceptions that are not normally translated by OSAX.

```
OSAX_HANDLE_EXCEPTION(H)
```

If a C++ exception derived from the class exception is thrown, an ActiveX exception is formed from the error description and the method returns an `HRESULT` of `H`.

```
OSAX_HANDLE_EXCEPTION(E, e) exception handling code
```

If a C++ exception derived from `E` is thrown, `e` is bound to a reference to the exception. You can handle the exception in an arbitrary way.

Raising Exceptions

OSAX provides `Error` methods corresponding to `AtlError`. These can be called anywhere within the scope of an OSAX method body. They will throw to the OSAX exception handler and then call the corresponding `AtlError` function. The functions that take an IID or CLSID use the value for the method if `GUID_NULL` is used as an argument to `Error`. 
**Template Classes**

All OSAX classes must use the following as a base class:

```cpp
template<class T> class IOSAXContextInformationImpl
```

The argument `T` should be the ATL class that implements your object. The class is used internally by OSAX to associate OSAX objects with appropriate transactions.

All OSAX instance class classes must use the following class as a base class:

```cpp
template <class CInstanceClass, class CInstance> class CInstanceClassImpl
```

The template arguments are the class of the instance class and the class of the instance. The class implements the `IOSAXType` interface.

```cpp
template <class CInstanceClass, class CInstance, class C> class CInstanceImpl
```

All classes that implement OSAX instances must include this base class. It defines the `CInstance::CDataPtr` class and many methods used internally by OSAX.

**OSAX Instance Data Members**

Every OSAX instance and instance container has a data member called `data`, which is implemented by the class `CInstance::CDataPtr`. `data` is a reference to the C++ object. In the descriptions that follow, `CInstance` is the class that implements your instance.

```cpp
static CInstance::GetInterfacePointer(const CInstance::CDataPtr&, I** ppI)
```

The `GetInterfacePointer` methods are used to obtain an interface pointer for an OSAX instance or instance container. They are defined by the `OSAX_INTERFACE_CONVERSION` macro.

```cpp
static CInstance::GetClassInterfacePtr(I** ppI)
```

The `GetClassInterfacePtr` method is used to obtain an interface pointer for the OSAX instance class associated with `CInstance`. The method is defined by `OSAX_CLASS_INTERFACE`.

```cpp
HRESULT CInstance::SetDataImpl(IUnknown* pUnk)
```

Use `SetDataImpl` to change which C++ object is associated with an instance container. The context associated with `pUnk` becomes the
context for the instance container. If SetDataImpl is called by an OSAX instance, E_INVALIDARG is returned.

static void CInstance::OnDeleteData(C* pC)

This method is called on FinalRelease when CInstance::DeleteDataOnFinalRelease has been called to mark the object for deletion on FinalRelease. It deletes pC. You can override this definition in the CInstance class to do something different.

HRESULT CInstance::DeleteDataOnFinalRelease(VARIANT_BOOL bDelete)

Use this method to specify whether or not the C++ data should be deleted upon final release of the OSAX instance.

static CComPtr<IOSAXInstanceClass> CInstance::InstanceClass()

Returns the InstanceClass (an internal OSAX object) for use with the OSAX operator new.

Methods of the CInstance::CDataPtr Class

The CDataPtr class is like a transient reference to the C++ object referenced by an OSAX instance. CDataPtr constructors are used to obtain OSAX instances from C++ pointers, as well as to obtain C++ pointers from OSAX instance interface pointers.

In the descriptions that follow, C is the class of the C++ type associated with CInstance.

CInstance::CDataPtr::CDataPtr()

Creates a reference to a NULL pointer.

CInstance::CDataPtr::CDataPtr(C* pC)

Obtains the OSAX instance for pC and wraps it in CDataPtr.

CInstance::CDataPtr::CDataPtr(IUnknown* pUnk)

If pUnk is an OSAX instance, CDataPtr refers to its C++ object.

If pUnk supports IATKRefString (an ATK object), then CDataPtr will reference the C++ pointer referenced by the ATK object.

CInstance::CDataPtr::CDataPtr(VARIANT& rvar, const CDataPtr& refDefault)

If rvar is an optional argument to a method, and the argument was supplied by the caller, then it should be an interface pointer, and the CDataPtr(IUnknown* pUnk) procedure is used.
If `rvar` was not supplied by the caller, then the `CDataPtr` is initialized to the C++ pointer referenced by `refDefault`.

```c
C* CInstance::CDataPtr::operator -> () const
CInstance::CDataPtr::operator C* () const
C* CInstance::CDataPtr::Resolve() const
```

All these return the referenced C++ pointer. You may need to explicitly use `Resolve` in some cases where a conversion operator cannot be applied.

```c
CDataPtr& CInstance::CDataPtr::operator = (const CDataPtr& rDataPtr)
```

Changes the referenced C++ pointer to that of `rDataPtr`. Do not assign to the data member of an OSAX instance. Instead, use the method `CInstance::SetDataImpl`.

```c
bool CInstance::CDataPtr::operator !() const
```

Returns `true` if `CDataPtr` references a `NULL` pointer.

### Using OSAX Instances to Implement Other Object Servers

You can use an OSAX instance that is implemented in another OSAX object server, as long as the object server is an `Inproc` server. You do this by using the class `CConversionImpl`.

```c
template <class CInstance, class C, const GUID* G>
class CConversionImpl
```

`C` is the C++ type referenced by the OSAX instance, and `G` is the CLSID for the instance class. `CInstance` is the class name you use as the class name in the server you are using. It does not need to be the same as the class name where the server is actually implemented.

You must add `OSAX_INTERFACE_CONVERSION` and `OSAX_CLASS_INTERFACE` macros for the relevant interfaces, and a constructor.

Examples can be found in `osax.h`. For example, `IOSAXString` is implemented in an OSAX object server and exposed to other object servers in `osax.h` as

```c
class COSAXString : public CConversionImpl<COSAXString,char,&CLSID_OSAXString> {
  public:
    COSAXString (IUnknown* pUnk) : ConversionImpl(pUnk){}
```
API Reference

OSAX_INTERFACE_CONVERSION(IOSAXString)
OSAX_CLASS_INTERFACE(IOSAXStringConstructor)

Collections

OSAX provides template base classes to expose ObjectStore collections as automation collections that can be used with Visual Basic’s For Each...Next statement.

template <class IDispatchImplCollection, class CInstance, class TElement,
class IElement>
class IOSAXCollectionImpl

Use this as a base class instead of IDispatchImpl if you want to expose a class derived from os_Collection as a collection.

IDispatchImplCollection is an IDispatchImpl template instantiation for your collection interface. CInstance is your ATL class. TElement is the class that implements the OSAX instances for the elements of the collection. IElement is the interface to use for methods that accept elements as arguments.

template <class IDispatchImplCollection, class CInstance, class TKey, class IKey,
class TElement, class IElement>
class IOSAXDictionaryImpl

Use this as a base class instead of IDispatchImpl if you want to expose a class derived from os_dictionary as a collection.

IDispatchImplCollection is an IDispatchImpl template instantiation for your collection interface. CInstance is your ATL class. TKey is the ATL class that implements the keys. IKey is the interface for the keys. TElement is the ATL class that implements the elements. IElement is the interface for the elements.

Macros for Class Definitions of OSAX Instances and Conversions

OSAX_INTERFACE_CONVERSION(I)

Add this macro to the class definition of your instance to define the method

\[\text{CInstance}::\text{GetInterfacePointer(const CDataPtr\&, I** ppi)}\]

OSAX_CLASS_INTERFACE(I)

Add this macro to the class definition of your instance to define the method
Chapter 3: Using OSAX with Microsoft ATL

CInstance::GetClassInterfacePtr(I** ppI)

OSAX_TYPE_NAME(name)

Add this macro to the class definition of your instance to specify the name of the C++ class. This name can be used to obtain an os_typespec.

OSAX_OBJECTSTORE_TYPESPEC

Add this macro to the class definition of your instance to allow it to get a typespec from the C++ class’s get_os_typespec method.

OSAX_NO_OBJECT_TABLE

Add this macro to the class definition of your instance to make it not use an object table. This means that there may be multiple OSAX instances for the same C++ object.

Global Functions

HRESULT OSAX::DeleteDataOnFinalRelease(IUnknown* pUnk, VARIANT_BOOL bFree)

Use this method if you want the C++ object referenced by pUnk to be deleted upon final release of pUnk.

bool OSAX::Equal(IUnknown* pUnk1, IUnknown* pUnk2)

Returns true if pUnk1 and pUnk2 refer to the same C++ object.

HRESULT OSAX::OnEndPage()

Call this in servers used by ASP if you want them to abort the transaction when a page is ended.

HRESULT OSAX::GetOSAXObjectStore(IOSAXObjectStore**)

Call this to obtain an IOSAXObjectStore interface pointer.

Overloadings of Operator new

void* operator new(size_t nSize, IUnknown* pUnkWhere, IOSAXInstanceClass* plInstanceClass)
void* operator new(size_t nSize, IUnknown* pUnkWhere, IOSAXInstanceClass* plInstanceClass, size_t nElts)
void* operator new(size_t nSize, VARIANT& rvar, IOSAXInstanceClass* plInstanceClass)
void* operator new(size_t nSize, VARIANT& rvar, IOSAXInstanceClass* plInstanceClass, size_t nElts)

OSAX provides these overloadings of operator new to simplify the allocation of C++ objects. You do not need these news. The pUnkWhere or rvar arguments specify where the allocation should occur. If they are a segment or database, allocation takes place in
the segment or database. If they are an OSAX instance, allocation occurs in the same segment as the referenced data. Otherwise, transient allocation is used.

Use the \textit{nElts} versions for vector allocation. The \textit{nElts} argument specifies the number of elements.
Chapter 4
Distributed OSAX Object Servers

An OSAX object server can be used in many different configurations, including local and distributed configurations. In local configurations, the database, object server, and application all execute on the same computer. In distributed configurations, these components execute on two or more computers cooperatively on behalf of a single user.

In distributed configurations, the object server initially resides on a remote computer, the server host. There are two mechanisms that a client can use to access the object server:

- Load dynamic Web pages from an ActiveX-enabled Web server, which generates the pages by accessing the object server; see Using Object Servers with Web Servers on page 73.
- Use DCOM to directly access the remote object server; see Accessing Networked Object Servers Using DCOM on page 74.

The advantages and disadvantages of these configurations are:

**Web server accesses local OSAX object server:**

- **Protocols:** HTTP to access Web pages, InProc COM within server.
### Advantages
- Supports all Web browsers.
- Access to very large centralized databases.
- Centralized database can be easier to update and administer.
- High-performance data access on server side.

### Disadvantages
- User interface is restricted to HTML constructs.
- Applications with extensive computation requirements must perform them on the server (where the data is), instead of distributing burden to clients.

#### Client accesses remote object server:

<table>
<thead>
<tr>
<th>Protocols</th>
<th>DCOM</th>
</tr>
</thead>
</table>
| Advantages | Access to very large centralized databases.  
            | Centralized database can be easier to update and administer.  
            | Easy to blend client-side and server-side computation. |
| Disadvantages | Performance 100 to 10000 times slower than InProc DCOM; requires careful design of object server interface.  
                | Network must pass DCOM protocols. |

The appropriate configuration for a given application depends on the application requirements for interactive performance, database size and administration, network scale, and platform independence.
Using Object Servers with Web Servers

An OSAX object server and database can be accessed directly from an Active Server Page using the Microsoft Active Server. Simply place a named reference to the object server class on the page. Then use the server’s scripting capability to access the database contents and format them into HTML content.

The following example shows the body of a Microsoft Active Server page that formats the contents of the Books example database as an HTML table.

```html
<object runat=server id="osBooks" classid="CLSID:84858E05-533D-11D0-8771-000000000001"></object>
<h2>ObjectStore OSAX Books Example - Microsoft Active Server</h2>
<% Set osDatabase = osBooks.ObjectStore.OpenDatabase("C:\ODI\OSAX\Examples\Books\Books.db") %>
<table border=2>
<tr><th>Title</th><th>Author</th></tr>
<% Set B = osDatabase.Value("Books", osBooks.CBookElt) %>
<% Do %>
<% Author = B.Book.Author.Name %>
<% Title = B.Book.Name %>
<tr><td><%=Title%></td><td><%=Author%></td></tr>
<% Set B = B.Next %>
<% Loop While Not B Is Nothing %>
</table>
<% osDatabase.Close %>
```
Accessing Networked Object Servers Using DCOM

An OSAX object server can be accessed remotely from a DCOM client with no changes to its implementation. Only the registration of the object changes, on both the client and server.

It is important to note that DCOM performance is dramatically slower than InProc COM performance. This significantly affects the design of an object server and its clients. An InProc COM client might reasonably perform computations directly on ObjectStore objects and properties, because of the high performance of object access in that configuration (millions of accesses per second). A DCOM client should minimize interaction with the object server by performing more computation on the server side, because of the poor performance of object/method access in that configuration (hundreds of accesses per second).

**DCOM Client Configuration**

You can access an OSAX object server remotely from a DCOM client. Register the class and indicate its location on a named remote server. There are different ways of accomplishing this, depending on the client environment. Here are two approaches:

- Install the object server (DLL, TLB, and supporting run-time libraries) on the client computer. Self-register the object server. Remove the entries for InProc and OutOfProc servers. Add the entry for RemoteServerName. DCOM configuration utilities like dcomcnfg ease this process somewhat, but it still requires user interaction with the client computer.

- Alternatively, prepare and distribute a customized .reg file to each client computer. See the Microsoft registry documentation for details.

In either case, the desired registry information for the class must not include the LocalServer32 or InProcServer32 keys, and must include the AppID key and the associated RemoteServerName.

**NOTE:** If the application name does not show up in the dcomcnfg control panel, make sure there is an AppID key associated with the object server’s CLSID in the registry (reregister with the osaxout utility).

```
[HKEY_ROOT_CLASSES\CLSID\{clsid}\]
```
"AppID" = "{clsid}"
Chapter 5
The Portfolio Sample Application

The Portfolio sample application illustrates a simple ActiveX Interface for ObjectStore application implemented in C++ and Visual Basic using an OSAX object server. The application provides a user interface for

- Selecting among a number of defined financial securities and portfolios
- Computing the valuation of a selected instrument over a period of time
- Displaying the result using a standard graphing component
Available portfolios (groups of securities) range from simple industry-sector portfolios (which maintain constant positions in a given list of securities, and thus contain few trades) to complex, large-scale funds containing many trades per day. (The Benchmark portfolio supplied in the sample database contains 20 trades per day over the course of 1995.)

Computing the value of a portfolio over a period of time is typical of financial data management problems involving time series data. The valuation of the portfolio on a given date is a function of the positions held in the portfolio. The number of shares held in each security is a function of the trades that have occurred in that security prior to that date. Essential to optimizing portfolio valuation computations is the ability to efficiently store and rapidly retrieve a value indexed by date.

The sample application includes a small database containing the 1995 trading history of about 600 publicly registered stocks, and a
few predefined portfolios. This would not be considered a very large database for a financial institution, yet it is large enough to illustrate the performance demands of this domain. The objective of the sample application is to provide interactive performance, meaning apparently instantaneous computation and display of portfolio valuations. You must take reasonable care in the design of the data representations to attain this. The OSAX server architecture implements critical computations in C++ and presents the results to Visual Basic for graphing and display. This is a good example of partitioning an application to use the respective strengths of each language.

With OSAX installed in the default location on your computer, the Portfolio sample application is in the following directory:

```
c:\odi\osax\examples\portfolio
```

To run the Portfolio sample application, load the Visual Basic project into the Visual Basic development environment and run it. When the form appears, select OpenDatabase from the Database menu to open the supplied `osaxport.db` database. The Securities and Portfolios list boxes are initialized with the available instruments from the database.

Once the list boxes are populated, selecting any security displays a graph of that security’s price history over 1995. Selecting any portfolio computes and graphs the Net Asset Value (the y-axis) of the portfolio over 1995 (the x-axis). If you select the name of a security in the Securities list, the graph reflects the trade volume just in that security rather than the entire portfolio.

In the lower left corner of the screen is a data display showing the trades associated with the selected portfolio. The first column lists the security ID, the second lists the date (in the form `yymmdd` for year, month, day), and the third lists the number of shares traded.
Portfolio Data Model

The principal data types in the data model for the Portfolio sample application are shown in the following table.

<table>
<thead>
<tr>
<th>Type</th>
<th>Properties</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Security</td>
<td>Symbol</td>
<td>String name of trading symbol</td>
</tr>
<tr>
<td></td>
<td>Price history</td>
<td>Price table indexed by date</td>
</tr>
<tr>
<td>Portfolio</td>
<td>Name</td>
<td>String name of portfolio</td>
</tr>
<tr>
<td></td>
<td>Trades</td>
<td>List of all trade objects in this portfolio</td>
</tr>
<tr>
<td>Trade</td>
<td>Security</td>
<td>Security object participating in trade</td>
</tr>
<tr>
<td></td>
<td>Date</td>
<td>Date of trade</td>
</tr>
<tr>
<td></td>
<td>Delta</td>
<td>Number of shares sold or acquired</td>
</tr>
</tbody>
</table>

You implement these data types in C++ and they are exposed to Visual Basic through an OLE type library. The type library is generated from the Portfolio database schema. You can load the OLE type library corresponding to the database schema into Visual Basic and inspect it using its Object Browser.

Implementation of the Portfolio Sample Application

The Portfolio sample application contains an object server implemented by a Visual C++ project. It also contains several different client applications written in different dialects and configurations of Visual Basic. The source code for all these projects is provided in the `examples\portfolio` directory.

Visual Basic Implementation of the Application

The Portfolio sample application uses the following Visual Basic components:

<table>
<thead>
<tr>
<th>Component</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>frmViewer</td>
<td>Form</td>
</tr>
<tr>
<td>grdTrades</td>
<td>Trade grid</td>
</tr>
<tr>
<td>grfNAVHistory</td>
<td>Value history graph</td>
</tr>
<tr>
<td>lblNAVHistory</td>
<td>Value history graph label</td>
</tr>
<tr>
<td>lblPortfolios</td>
<td>Portfolios list box label</td>
</tr>
</tbody>
</table>
### Component Purpose

<table>
<thead>
<tr>
<th>Component</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>lblSecurities</td>
<td>Securities list box label</td>
</tr>
<tr>
<td>lblTrades</td>
<td>Trades list box label</td>
</tr>
<tr>
<td>lstPortfolios</td>
<td>Portfolios list box</td>
</tr>
<tr>
<td>lstSecurities</td>
<td>Securities list box</td>
</tr>
<tr>
<td>menuCloseDatabase</td>
<td><strong>Close Database</strong> menu choice</td>
</tr>
<tr>
<td>menuExit</td>
<td><strong>Exit</strong> menu choice</td>
</tr>
<tr>
<td>menuOpenDatabase</td>
<td><strong>Open Database</strong> menu choice</td>
</tr>
<tr>
<td>menuSeparator</td>
<td>Menu separator bar</td>
</tr>
<tr>
<td>sbPortfolioViewer</td>
<td>Status bar</td>
</tr>
</tbody>
</table>
Implementation of the Object Server

The following table describes the object model for the Portfolio database.

<table>
<thead>
<tr>
<th>Object Type</th>
<th>Exposed Method</th>
<th>Method Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Security</td>
<td>Symbol</td>
<td>Property</td>
<td>Return string name.</td>
</tr>
<tr>
<td></td>
<td>LookupPrice</td>
<td>Method</td>
<td>Look up security price on a given date.</td>
</tr>
<tr>
<td>Portfolio</td>
<td>Name</td>
<td>Property</td>
<td>Return string name.</td>
</tr>
<tr>
<td>Trades</td>
<td></td>
<td>Property</td>
<td>List all trade objects in this portfolio.</td>
</tr>
<tr>
<td></td>
<td>ComputeNAV</td>
<td>Method</td>
<td>Compute valuation over specified range of dates.</td>
</tr>
<tr>
<td>Trade</td>
<td>Security</td>
<td>Property</td>
<td>Return Security object.</td>
</tr>
<tr>
<td></td>
<td>Date</td>
<td>Property</td>
<td>Return Date of trade.</td>
</tr>
<tr>
<td></td>
<td>Delta</td>
<td>Property</td>
<td>Return number of shares sold or acquired.</td>
</tr>
</tbody>
</table>

The Portfolio sample application is provided in `\examples\portfolio`. It includes the files listed in the Portfolio ReadMe.txt file.
Chapter 6
ActiveX Interface Reference

This chapter describes the ActiveX interfaces, which are listed alphabetically by interface name. Within the entry for each interface, the methods and properties are listed alphabetically. For more information about the equivalent ObjectStore C++ API or behavior, as presented in the table in the next section, see the ObjectStore C++ API Reference.
OSAX Interfaces at a Glance

The following table provides a quick reference of equivalent interfaces.

<table>
<thead>
<tr>
<th>OLE Interface</th>
<th>Methods and Properties</th>
<th>ObjectStore C++ API or Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>IOSAXCollection</td>
<td>Item (default)</td>
<td>os_collection::pick()</td>
</tr>
<tr>
<td></td>
<td>Add</td>
<td>os_collection::insert()</td>
</tr>
<tr>
<td></td>
<td>Remove</td>
<td>os_collection::remove()</td>
</tr>
<tr>
<td></td>
<td>ElementType</td>
<td>C.ElementType = XClass</td>
</tr>
<tr>
<td></td>
<td>Cardinality() As Long</td>
<td>Returns number of elements</td>
</tr>
<tr>
<td></td>
<td>Count() As Long</td>
<td>Returns number of elements</td>
</tr>
<tr>
<td></td>
<td>IsEmpty() As Boolean</td>
<td>os_collection::empty()</td>
</tr>
<tr>
<td></td>
<td>IsOrdered() As Boolean</td>
<td>false</td>
</tr>
<tr>
<td></td>
<td>Evaluate</td>
<td>os_collection::query()</td>
</tr>
<tr>
<td>IOSAXDatabase</td>
<td>Open</td>
<td>os_database::open()</td>
</tr>
<tr>
<td></td>
<td>OpenMVCC</td>
<td>os_database::open_mvcc()</td>
</tr>
<tr>
<td></td>
<td>Close</td>
<td>os_database::close()</td>
</tr>
<tr>
<td></td>
<td>Value</td>
<td>Name As String, type As IOSAXType</td>
</tr>
<tr>
<td>IOSAXObjectStore</td>
<td>CreateDatabase</td>
<td>os_database::create()</td>
</tr>
<tr>
<td></td>
<td>OpenDatabase</td>
<td>os_database::open()</td>
</tr>
<tr>
<td></td>
<td>OpenDatabaseMVCC</td>
<td>os_database::open_mvcc()</td>
</tr>
<tr>
<td></td>
<td>BeginTrans</td>
<td>os_transaction::begin()</td>
</tr>
<tr>
<td></td>
<td>LookupDatabase</td>
<td>Returns unopened IOSAXDatabase</td>
</tr>
<tr>
<td></td>
<td>CommitTrans</td>
<td>os_transaction::commit()</td>
</tr>
<tr>
<td></td>
<td>Rollback</td>
<td>os_transaction::abort()</td>
</tr>
<tr>
<td></td>
<td>Equal</td>
<td>==</td>
</tr>
<tr>
<td></td>
<td>IsPersistent</td>
<td>objectstore::is_persistent()</td>
</tr>
<tr>
<td>IOSAXString</td>
<td>Value</td>
<td>Translates char* value to string</td>
</tr>
</tbody>
</table>
The following sections describe the interfaces in more detail. They are presented in alphabetical order.

<table>
<thead>
<tr>
<th>OLE Interface</th>
<th>Methods and Properties</th>
<th>ObjectStore C++ API or Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>IOSAXStringConstructor</td>
<td>Ans1</td>
<td>Allocates char*; uses ANSI code page for translation</td>
</tr>
<tr>
<td></td>
<td>Copy</td>
<td>Copies char* referenced by IOSAX</td>
</tr>
<tr>
<td></td>
<td>OEM</td>
<td>Allocates char*; uses OEM code page for translation</td>
</tr>
<tr>
<td>IOSAXType</td>
<td>Name</td>
<td>String name of type</td>
</tr>
<tr>
<td></td>
<td>Cast</td>
<td>Returns its argument’s referenced C++ object as an instance object of this type</td>
</tr>
<tr>
<td>IOSAXUnknownObject</td>
<td></td>
<td>For C++ objects that do not have an OSAX class; like void* in C++</td>
</tr>
</tbody>
</table>
**IOSAXCollection**

**Purpose**

Represents an `os_collection` or a class derived from it.

When `osgentyp` defines an OLE class for a parameterized collection, the instance objects have two interfaces. One is `IOSAXCollection`. The other interface is like `IOSAXCollection`, except that the element arguments and return values of the element-specific methods use the element type rather than `Object`.

**Methods**

**Add(element As Object)**

Inserts the element into the collection, using `os_collection::insert`.

**Cardinality() As Long**

Returns the number of elements in the collection. This is the same as `Count()`.

**Count() As Long**

Returns the number of elements in the collection. This is the same as `Cardinality()`.

**ElementType()**

A settable property. You must set this to the element-appropriate type when you have a nonparameterized collection, such as an `os_set`. For example, if you have a collection of `X*` objects, and `XClass` is the class object for class `X`, and `C` is your collection, you need to say

```plaintext
C.ElementType = XClass
```

before you can access elements from the collection.

**Evaluate(query As String) As Variant**

Returns a collection of the same element type that is the result of the specified query.

**IsEmpty() As Boolean**

Returns `true` if the collection is empty. Uses `os_collection::empty()`.

**IsOrdered() As Boolean**
Returns true for collections that are ordered lists.

**Item() As Object**

Returns an element from the collection using `os_collection::pick()`.

**Remove(element As Object)**

Removes the element from the collection using `os_collection::remove()`.
IOSAXDatabase

Purpose

Represents a database.

Methods

Close

Closes the database with `os_database::close()`.

Open

Opens a database with `os_database::open()`.

OpenMVCC

Opens a database for MVCC; equivalent to `os_database::open_mvcc()`.

Value(Name As String, type As IOSAXType) As Object

Returns the value of the named root, which must have the specified type. Use a class object for the type. Class objects support the IOSAXType interface.

If the root has a typespec, the typespec is verified against the type. Otherwise, the type is assumed to be correct.
Chapter 6: ActiveX Interface Reference

IOSAXObjectStore

Purpose

The following are general ObjectStore control methods for opening databases, transactions, and so on.

Methods

BeginInitialization() As Boolean

Returns false if ObjectStore has already been initialized. This method can be called before any other ObjectStore methods that must be called before ObjectStore is initialized.

BeginTrans([ReadOnly As Variant])

Starts a transaction using os_transaction::begin().

CommitTrans

Commits a transaction using os_transaction::commit().

CreateDatabase(Name As String, [CreateMode As Variant], [OverWrite As Variant], [SchemaDatabase As Variant]) As IOSAXDatabase

Creates a database.

The CreateMode is an octal UNIX-style file create mode that defaults to 0664. OverWrite is an optional Boolean that defaults to false. If true, an existing database by the same name is overwritten.

SchemaDatabase specifies a database to use for schema.

Equal(First As Object, Second As Object) As Boolean

Tells if two objects refer to the same C++ pointer.

Initialize([Reserved As Variant])

Initialize ObjectStore. Should be set to 0.

InitializeTransactions()

Does nothing on ObjectStore.

IsPersistent(Obj As Object) As Boolean

Returns true if the object is stored in an ObjectStore database.

LanguageInterfaceMaintenanceRelease() As Long
Returns the maintenance release of the language interface.

LanguageInterfaceMajorRelease() As Long
    Returns the major release of the language interface.

LanguageInterfaceMinorRelease() As Long
    Returns the minor release of the language interface.

LanguageInterfaceName() As Long
    Returns the name of the language interface.

LoadATKReference(Type As IOSAXType, RefString As Object) As Object
    Returns the object corresponding to the reference from ATK.

LookupDatabase(Name As String) As IOSAXDatabase
    Returns an unopened IOSAXDatabase. Name is the name of a database.

MaintenanceRelease() As Long
    Returns the maintenance release number.

MajorRelease() As Long
    Returns the major release number.

MinorRelease() As Long
    Returns the minor release number.

OpenDatabase(Name As String, [ReadOnly As Variant], [CreateMode As Variant], [SchemaDatabase As Variant]) As IOSAXDatabase
    Opens an existing database with the given Name. ReadOnly is an optional Boolean defaulting to false. CreateMode is a UNIX-style protection mode defaulting to 0. If it has a nonzero value and the database does not exist, the database is created with the specified protection. SchemaDatabase is an optional IOSAXDatabase to be used for the schema.

OpenDatabaseMVCC(Name As String)
    Opens an existing database with the given Name for MVCC; equivalent to os_database::open_mvcc(Name).

ProductName() As String
Returns the name of the product.

Rollback

Rolls back a transaction using \texttt{os\_transaction::abort()}.

SetContext(\texttt{Obj As Object})

Changes the context associated with this ObjectStore Server to that of \texttt{Obj}.

StorageSystemName() As String

Returns the name of the storage system.
**Purpose**

Visual Basic and OLE use BSTRs to represent strings, while C++ uses `char*`. If OSAX converted `char*` to BSTR, it would need to copy all strings. Therefore, if you passed such a string from one method to another, you would be passing a twice-copied version of it, rather than the intended original.

OSAX avoids this problem with **IOSAXString**, which is a reference to a `char*`. When Visual Basic asks for the value of an **IOSAXString**, a copy is made and returned at that time. If an **IOSAXString** object is passed to something that wants an **IOSAXString** argument, the original `char*` is available.

**Methods**

**Value() As String**

Translates the `char*` value to a string using the **ANSI** code page.
IOSAXStringConstructor

**Purpose**

IOSAXStringConstructor is the class interface for IOSAXString. It contains methods for allocating and copying IOSAXStrings.

**Methods**

- **ANSI(init As String, [where As Variant]) As IOSAXString**
  
  Allocates a char* and initializes it with init using the ANSI code page for translation. If where is specified, it should be the database in which to allocate the string.

- **Copy(Value As IOSAXString, [where As Variant]) As IOSAXString**
  
  Makes a copy of the string. If where is specified, it should be the database in which to allocate the string.

- **OEM(init As String, [where As Variant]) As IOSAXString**
  
  Allocates a char* and initializes it with init using the OEM code page for translation. If where is specified, it should be the database in which to allocate the string.
IOSAXType

Purpose

This interface is available on all instance class objects and is used for methods that need a type, such as IOSAXDatabase.Value.

For example, the C++ class X has an IX interface for its instances and an IXClass interface for its class methods. The IXClass object also has an IOSAXType interface.

Methods

Cast(Obj As Object) As Object

Returns an OSAX object for the same C++ pointer as its argument, only using the class as its instance type. You can use this to access root values that have no associated type.

Name() As String

Returns the class name associated with the type.
IOSAXUnknownObject

**Purpose**

Used for instances when nothing is known about the C++ type. It is like a `void*` pointer in C++. 

Index

A
aborting transactions 69
Active Template Library, See ATL
ActiveX
   AppWizard 14
   architecture 2
   Automation 2
   exceptions 24
   ObjectStore AppWizard and type
description file 9
   ObjectStore AppWizard installed by
setup 4
   overview of ActiveX Interface 1
Ansi code page 35
ANSI method
description 93
appinstance attribute 22
appobject attribute 22
AppWizard
   ATL COM 52
   creating custom build rules 44
   ObjectStore ActiveX 4
   ObjectStore and type description file 9
   ObjectStore AppWizard installed by
setup 4
As New modifier 23
ASP 69
ATK objects 66
ATL 47
   classes 49
   classes, adding to code 53
   COM AppWizard 52
   concepts 49
   implementing an object server 52
   interface 18
   methods 50
   objects 49
   vs. type description file 9, 47
AtlError function 64
AtlReportError 64
attributes
   library specification 20
   method specification 24
   object specification 21
   automatic type-checking 11
   automation
      collections 68
      interfaces 17
Automation server, configurations 3
B
basic types 17
BeginInitialization method 89
BeginTrans method
   Books example 10
description 89
98 ActiveX Interface for ObjectStore

C

Visual Basic syntax 16
Books example 6
database objects, defining 11
directory 8
object server 9
type description file 9
Visual Basic project 10
Visual C++ project 8
Both threading model 18
BSTR ActiveX data type 17
BSTR, Unicode 34
building object servers 14

C

/c option 44
C++
class, and OSAX 49
constructors, exposing 32
exceptions 24
implementation file, specifying name 44
static member functions, exposing 31
Cast method 94
CDataPtr class
    CInstance method 66
data member 51
nested base class 58
CExceptionTranslator class
    and OSAX classes 49
CExceptionTranslator class 64
character arrays 17
character strings
    accessing and creating 34
    creating persistent 35
    persistent 35
    transient 35
    in type description files 34
class attribute 24
class factory
    and progid attribute 23
    creating OSAX instance 61
class ID 19

class method implementation 26
classes, ATL and OSAX 49
classinterface attribute 22
classuuid attribute 22
clients, single-threaded and multi-threaded 18
Close method
description 88
    Visual Basic syntax 16
CloseDatabase method
    Books example 10
CLSID 19
collections
    accessing 37
    creating 38
    defining in type definition file 37
    deleting 40
    exposing as automation collection 68
    exposing, example 37
    iterating over 39
    querying 39
COM class 49
    registration 14
COM interfaces
    creating with ATL 9
    defining 47
    IUnknown and IDispatch 2
COM object 49
CommitTrans method
    Books example 10
description 89
    Visual Basic syntax 16
configuration options 2
configurations, Automation server 3
constants, C++ and Visual Basic
differences 12
constructors, exposing 32
conversion attribute 22
converting types 17
Copy method 93
.cpp file 15
CreateDatabase method 89
CreateLibrary routine 38
CreateObject function 23
creating objects 31
custom build rules, created by
AppWizard 44

D
/d option 44
data member
  implemented by CDataPtr 51
  OSAX instance data member 65
  smart pointer 56
database objects, defining 11
database roots
  Books example 12
  using and creating 41
DCOM 4
  client configuration 74
DCOM configuration option 2
debug run-time library 53
.def file 14
DeleteOnRelease function 33
deleting
  collections 40
  objects 63
  temporary objects 33
distributed configurations 71
DLL
  API 15
  entry points 14
  specifying filename 44
domains, object type 2

E
E_INVALIDARG 42
early binding 2
Equal method 89
Error methods 64
examples 6

Books 6
Portfolio, described 77
Portfolio, overview 6
Travel 7
exception attribute
  ActiveX exceptions 42
  method specification in .ost file 24
  object specification in .ost file 22
exceptions 64
  ActiveX 24
  C++, catching and signaling 42
  exception attribute 24
  handlers 42
  and transactions 43
  translate methods 64
exposing
  C++ constructor 32
  C++ static member function 31
  collections 37
  methods 31

F
factoryuuid attribute 22
fault-handling macros 56
features of ActiveX Interface for
  ObjectStore 1
filenames, specifying 44
files
  .idl 45
  .ost 14
  .rc 14
  .rgs 14
  .tlb 15
generated by osgentyp utility 44
header 44
header for interface definition 15
implementing COM object 14
interface definitions 15
resource 14
schema 15
stdafx.h 15
type library binary (TLB) 15
For Each...Next statement
IOSAXCollection 39
template base classes 68
function implementation 26
functions, static member 31

G
generating UUIDs 45
generic interface 17
global
functions 69
properties 10

H
header files 15
helpstring attribute
library specification in .ost file 20
method specification in .ost file 24
object specification in .ost file 23

I
/i option 44
/l ActiveX data type 17
id attribute 24
IDispatch
interface and contents of database 17
interface, and ActiveX 2
objects 34
.idl file
generated by osgentyp 14
syntax for specifying 45
/ih option 44
*_imp.cpp file 15
#include files 44
Initialize method 89
InitializeTransactions method 89
InProc configuration option 2
Inproc server 67
in-process servers 18
installing 4
instance class interface 24
instance classes 21
described in .ost file 19
OSAX 51
produced by osgentyp 51
instance containers
described in .ost file 19
OSAX 50
progid attribute 23
instance interface 24
InstanceClass object 66
instancelfactoryuuid attribute 23
instanceprogid attribute 23
instances
creating 61
described in .ost file 19
for existing C++ objects 61
OSAX 50
interface attribute 23
interface definitions 15
invoking osgentyp 44
IOSAXCollection ActiveX data type 17
IOSAXCollection interface
described 86
ObjectStore collections 37
IOSAXContextInformationImpl class
and OSAX classes 49
IOSAXDatabase ActiveX data type 17
IOSAXDatabase interface 88
IOSAXDatabaseRoot interface 41
IOSAXObjectStore interface 89
IOSAXObjectStore interface pointer 69
IOSAXString ActiveX data type 17
IOSAXString interface
described 92
exposing strings 34
IOSAXStringConstructor interface
creating strings 35
described 93
Index

IOSAXStringInterface::Ansi method 35
IOSAXStringInterface::Copy method 35
IOSAXStringInterface::OEM method 35
IOSAXType interface
  described 94
  implementing 54
IOSAXUnknownObject interface 95
isomorphic attribute
  library specification in .ost file 20
  object specification in .ost file 23
IsPersistent method 89
iterating over collections 39
IUnknown argument 59
IUnknown interface 2

L

/l option 44
LanguageInterfaceMaintenanceRelease
  method 89
LanguageInterfaceMajorRelease method 90
LanguageInterfaceMinorRelease method 90
LanguageInterfaceName method 90
lcid attribute 20
library specification
  attributes 20
  example 21
  syntax 19
LoadATKReference method 90
local configuration 71
Local configuration option 2
LookupDatabase method 90

M

macros
  defining methods 68
  for exception-handling 50
MaintenanceRelease method 90
MajorRelease method 90
marshaling 15
method implementation 26
method specifications
  attributes 24
  syntax 24
methods
  ATL and OSAX 50
  exposing 31
  implementing 25
  OSAX 64
Microsoft Active Template Library
  See ATL
midl compiler 15
MinorRelease method 90
MsgBox and strings 34
multi-threaded clients 18

N

Name method 94
named object 11
new operator, overloaded
  described 69
  used with IUnknown and VARIANT 59
Nothing (Visual Basic)
  and database root 41
  converting 17
  in Books example 12
NULL (C++)
  and database root 41
  converting 17
  in Books example 12

O

object servers 13
  basic operations supported 16
  basic types supported 17
  Books example 9
  building 14
  configuration 18
  creating objects 31
  exceptions 42
  implementing with ATL 52
in-process 18
out-of-process 18
registered by setup 4
typical 49
using with Web servers 73

object specifications
  attributes 21
  example 26
  syntax 21

Object  Visual Basic type 17
  objects
    creating 31
    deleting 63
    not deleted when unreferenced 33
    type domains 2

objectstore attribute 20

OEM code page 35

OEM method 93

OLE
  Automation error 42
  objects 10

Open method
  description 88
  Visual Basic syntax 16

OpenDatabase method
  Books example 10
  Visual Basic syntax 16

OpenDatabase method 90

OpenDatabaseMVCC method 90

OpenMVCC method 88

options for osgentyp 44

os_array 17

os_bag 17

os_Bag base class 37

os_collection 17

os_Collection base class
  and ActiveX interface 38
  described 68

os_Collection::query 39

os_database 17

os_dictionary 17

os_Dictionary base class 37

os_List 37

os_segment 17

os_set 17

os_Set base class 37

OSAX
  class 49
  client 49
  installing 4
  instance, most important OSAX
    object 50
  instance, obtaining 57
  instances in other object servers 67
  methods 64
  objects 49
  overview 1
  uninstalling 5

osax.h 58

osgentyp utility
  generating exception handlers 42
  generating UUIDs 45
  options 44
  syntax 44

osggentyp utility 44

ossg utility
  adding custom build rule 58
  application wizard 15

.ost file
  osgentyp utility 44
  type description file 14

out-of-process servers 18

overloaded new
  described 69
  used with IUnknown and VARIANT 59

overview of ActiveX Interface for
  ObjectStore 1

P

persistent character strings, creating 35

persistent objects 9

persistent.h 58
Portfolio sample application
  data model 80
  described 77
  directory location 79
  object model 82
  object server implementation 82
  overview 6
  Visual Basic implementation 80

Product Name method 90
  progid attribute 23
  property accessor 24
  property setter 25
  proppget attribute 24
  proppput attribute 25
  proppputref attribute 25

Q
  querying collections 39

R
  R4 ActiveX data type 17
  R8 ActiveX data type 17
  .rc resource file 14
  registering object servers
    osgentyp utility 15
    setup 4
  regsvr32 program 15
  removing the product 5
  resource file 14
  restricted attribute 25
  .rgs file 14
  Rollback method
    Books example 10
    described 91
    Visual Basic syntax 16
  root object 11
  RPC 2

S
  schema file 15
  schema files, marked as DLL 58
  self-registering DLL 15
  Service Packs and installation 4
  SetCleanup method 33
  SetContext function 29
  SetContext method 91
  SetDataImpl method 50
  SETUP program 4
  SetValue function 30
  signaling exceptions 42
  single-threaded clients 18
  smart pointers 59
  specifying filenames 44
  static member functions, exposing 31
  stdafx.h
    changing 58
    used with DLL 15
  storage management 33
  StorageSystemName method 91
  system registry 4
  system requirements 4

T
  /t option 45
  template classes 37
    and OSAX classes 65
    exposing collections 68
  threading model
    OSAX classes 49
    single-threaded and multi-threaded clients 18
  TIX exception 42
  .tlb file, See type library binary 15
  top-level objects
    described in .ost file 19
    object attribute 21
    progid attribute 23
  transactions and exceptions 43
translate methods and exceptions 64
transport mechanism 2
Travel example 7
type conversions 17
type description file 14
and ATL 47
Books example 9
contents 19
library specification 19
method specifications 24
object specifications 21
read by osgentyp 44
specifying name 44
syntax 19
type library binary 15
in type description file 19
specifying name 21
types, support for 17

U

/u option 45
Unicode BSTR 34
uninstalling OSAX 5
uuid attribute
  library specification in .ost file 20
  object specification in .ost file 24
UUIDs
  generating 45
  in library specification 20
  in object specification 22

V

Value, IOSAXDatabase method
description 88
  Visual Basic syntax 16
Value, IOSAXString method 92
VARIANT argument 59
version attribute
  library specification in .ost file 20
  object specification in .ost file 24