ActiveX Interface for ObjectStore User Guide

ObjectStore Release 6.3 for all platforms

© 2005 Progress Software Corporation. All rights reserved.

Progress® software products are copyrighted and all rights are reserved by Progress Software Corporation. This manual is also copyrighted and all rights are reserved. This manual may not, in whole or in part, be copied, photocopied, translated, or reduced to any electronic medium or machine-readable form without prior consent, in writing, from Progress Software Corporation.

The information in this manual is subject to change without notice, and Progress Software Corporation assumes no responsibility for any errors that may appear in this document.

The references in this manual to specific platforms supported are subject to change.

A (and design), Allegrix, Allegrix (and design), Apama, Business Empowerment, DataDirect (and design), DataDirect Connect, DataDirect Connect OLE DB, DirectAlert, EasyAsk, EdgeXtend, Empowerment Center, eXcelon, Fathom, IntelliStream, O (and design), ObjectStore, OpenEdge, PeerDirect, P.I.P., POSSENET, Powered by Progress, Progress, Progress Dynamics, Progress Empowerment Center, Progress Empowerment Program, Progress Fast Track, Progress OpenEdge, Partners in Progress, Partners en Progress, Persistence, Persistence (and design), ProCare, Progress en Partners, Progress in Progress, Progress Profiles, Progress Results, Progress Software Developers Network, ProtoSpeed, ProVision, SequelLink, SmartBeans, SpeedScript, Stylus Studio, Technical Empowerment, WebSpeed, and Your Software, Our Technology-Experience the Connection are registered trademarks of Progress Software Corporation or one of its subsidiaries or affiliates in the U.S. and/or other countries. AccelEvent, A Data Center of Your Very Own, AppsAlive, AppServer, ASPen, ASP-in-a-Box, BusinessEdge, Cache-Forward, DataDirect, DataDirect Connect64, DataDirect Technologies, DataDirect XQuery, DataXtend, Future Proof, ObjectCache, ObjectStore Event Engine, ObjectStore Inspector, ObjectStore Performance Expert, POSSE, ProDataSet, Progress Business Empowerment, Progress DataXtend, Progress for Partners, Progress ObjectStore, PSE Pro, PS Select, SectorAlliance, SmartBrowser, SmartComponent, SmartDataBrowser, SmartDataObjects, SmartDataView, SmartDialog, SmartFolder, SmartFrame, SmartObjects, SmartPanel, SmartQuery, SmartViewer, SmartWindow, WebClient, and Who Makes Progress are trademarks or service marks of Progress Software Corporation or one of its subsidiaries or affiliates in the U.S. and other countries. Java and all Java-based marks are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries. Any other trademarks or trade names contained herein are the property of their respective owners.

ObjectStore includes software developed by the Apache Software Foundation (http://www.apache.org/). Copyright 2000-2003 The Apache Software Foundation. All rights reserved. The names "Ant," "Xerces," and "Apache Software Foundation" must not be used to endorse or promote products derived from the Products without prior written permission. Any product derived from the Products may not be called "Apache", nor may "Apache" appear in their name, without prior written permission. For written permission, please contact apache@apache.org.

September 2005
Contents

Preface ................................................................. 5

Chapter 1 Overview of ObjectStore .NET COM Interop ............... 9
Installing ObjectStore .NET COM Interop ............................. 11
  System Requirements ............................................... 11
  Setting Up the OSNCI Environment ................................. 11
Examples ............................................................... 12
  Books Example ....................................................... 12
A Tour of the Books Example .......................................... 12
  The Visual C++ Project ............................................. 13
  The Microsoft .NET C# Project .................................... 13
  The Books Database .................................................. 15

Chapter 2 Building OSNCI Object Servers ............................. 17
Building an Object Server ............................................... 17
Accessing an Object Server ............................................ 26
Basic Operations Supported by an Object Server .................... 26
Basic Types Supported by an Object Server .......................... 27
Object Server Configuration ............................................ 27
Type Description File .................................................... 28
  Library Specification ............................................... 28
  Object Specification ................................................. 30
Accessing OSAX Objects ................................................ 35
Creating Objects ......................................................... 36
  Expose a Method on One Class That Creates Instances of Another Class 36
  Expose a C++ Static Member Function That Creates Instances of Its Class 37
  Expose a C++ Constructor That Creates Instances of Its Class ............. 38
Deleting Temporary C++ Objects ..................................... 38
Accessing and Creating Character Strings ........................... 39
  Creating Persistent Character Strings ............................... 40
Accessing Collections ................................................... 40
  Defining a Collection ............................................... 41
<table>
<thead>
<tr>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>Creating a Collection of Objects</td>
</tr>
<tr>
<td>Iterating over Collections</td>
</tr>
<tr>
<td>Querying Collections</td>
</tr>
<tr>
<td>Deleting a Collection</td>
</tr>
<tr>
<td>Creating and Using Database Roots</td>
</tr>
<tr>
<td>Object Server Exceptions</td>
</tr>
<tr>
<td>Signaling Exceptions from an Object Server</td>
</tr>
<tr>
<td>Handling Exceptions from an Object Server</td>
</tr>
<tr>
<td>The osgentyp Utility</td>
</tr>
<tr>
<td>Command-Line Syntax</td>
</tr>
<tr>
<td>Generating UUIDs for Attributes</td>
</tr>
<tr>
<td>Multisession Support</td>
</tr>
</tbody>
</table>

**Chapter 3**

Distributed OSNCI Object Servers | 49 |
Using Object Servers with Web Servers | 50 |
Using DCOM to Access Networked Object Servers | 51 |
DCOM Client Configuration | 51 |

**Chapter 4**

ActiveX Interface Reference | 53 |
OSNCI Interfaces at a Glance | 53 |
IOSAXCollection | 54 |
IOSAXDatabase | 55 |
IOSAXObjectStore | 56 |
IOSAXSessionPool | 58 |
IOSAXStorage | 58 |
IOSAXString | 59 |
IOSAXStringConstructor | 59 |
IOSAXType | 60 |
IOSAXUnknownObject | 60 |
Preface

Purpose

ObjectStore .NET COM Interop describes how to install and use ObjectStore .NET COM Interop (OSNCI) Release 6.3 on a Windows system.

Audience

This book is primarily intended for developers who will build OSNCI 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 will be accessing the database as ActiveX clients. They should be familiar with either Microsoft .NET or a scripting language that interacts with ActiveX.

Scope

This document provides all the necessary information for installing OSNCI Release 6.3 and for building an OSNCI 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 OSNCI. The later chapters describe how to build OSNCI object servers.

Notation Conventions

This document uses the following conventions

<table>
<thead>
<tr>
<th>Convention</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Courier</td>
<td>Courier font indicates code, syntax, file names, API names, system output, and the like.</td>
</tr>
<tr>
<td>Bold Courier</td>
<td>Bold Courier font is used to emphasize particular code.</td>
</tr>
<tr>
<td>Italic Courier</td>
<td>Italic Courier font indicates the name of an argument or variable for which you must supply a value.</td>
</tr>
<tr>
<td>Sans serif</td>
<td>Sans serif typeface indicates the names of user interface elements such as dialog boxes, buttons, and fields.</td>
</tr>
<tr>
<td>Italic serif</td>
<td>In text, italic serif typeface indicates the first use of an important term.</td>
</tr>
<tr>
<td>[ ]</td>
<td>Brackets enclose optional arguments.</td>
</tr>
</tbody>
</table>
The ObjectStore Web site (www.progress.com/realtime/products/objectstore) provides a variety of useful information about products, news and events, special programs, support, and training opportunities.

Technical Support

To obtain information about purchasing technical support, contact your local sales office listed at www.progress.com/realtime/techsupport/contact, or in North America call 1-781-280-4833. When you purchase technical support, the following services are available to you:

- You can send questions to support@objectstore.com. Remember to include your serial number in the subject of the electronic mail message.
- You can call the Technical Support organization to get help resolving problems. If you are in North America, call 1-781-280-4005. If you are outside North America, refer to the Technical Support Web site at www.progress.com/realtime/techsupport/contact.
- You can file a report or question with Technical Support by going to www.progress.com/realtime/techsupport/techsupport_direct.
- You can access the Technical Support Web site, which includes
  - A template for submitting a support request. This helps you provide the necessary details, which speeds response time.
  - Solution Knowledge Base that you can browse and query.
  - Online documentation for all ObjectStore products.
  - White papers and short articles about using ObjectStore products.
  - Sample code and examples.
  - The latest versions of ObjectStore products, service packs, and publicly available patches that you can download.
  - Access to an ObjectStore support matrix that lists platform configurations supported by this release.
  - Support policies.
  - Local phone numbers and hours when support personnel can be reached.

Education Services

Use the ObjectStore education services site (www.progress.com/realtime/services) to learn about the standard course offerings and custom workshops.
If you are in North America, you can call 1-800-477-6473 x4452 to register for classes. If you are outside North America, refer to the Technical Support Web site. For information on current course offerings or pricing, send e-mail to classes@progress.com.

In addition to the online documentation that is included with your software distribution, the full set of product documentation is available on the Technical Support Web site at www.progress.com/realtime/techsupport/documentation. The site supports the most recent release and the previous supported release of ObjectStore documentation. Service Pack README files are also included to provide historical context for specific issues. Be sure to check this site for new information or documentation clarifications posted between releases.

Your Comments

ObjectStore product development welcomes your comments about its documentation. Send any product feedback to support@objectstore.com. To expedite your documentation feedback, begin the subject with Doc:. For example:

Subject: Doc: Incorrect message on page 76 of reference manual
Chapter 1
Overview of ObjectStore .NET COM Interop

ObjectStore .NET COM Interop (OSNCI) exposes ObjectStore C++ databases and their contents to Microsoft .NET 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. ObjectStore .NET COM Interop provides a mapping between the ObjectStore schema and the object model underlying COM. It also provides a mechanism for communicating object references and values between an ObjectStore database and an Automation controller.

OSNCI servers are COM servers that can be accessed from .NET Common Language Runtime (CLR) applications via the COM interoperability support in Microsoft .NET. This support is called .NET/COM Interop. The support provided by .NET/COM Interop means that .NET CLR applications can access ObjectStore databases via OSNCI servers that are wrapped as .NET components.

OSNCI features

ObjectStore .NET COM Interop 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, by using generated ActiveX type libraries.
- Access to the ObjectStore collections facility. You can use the collections facility transparently in place of .NET 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 OSNCI ActiveX interfaces, see Chapter 4, ActiveX Interface Reference, on page 53.
Three features in the ActiveX architecture 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 COM type libraries generated from an ObjectStore schema.

The following diagram shows the role the object server plays between an Automation controller and an ObjectStore database.

The ActiveX interface supports several different configurations for the OSNCI object server and several means of making type information available to COM. The ActiveX interface enables you to expose objects in an ObjectStore database, making them accessible to .NET applications with little or no direct programming of COM interfaces.

ObjectStore .NET COM Interop has been developed from the ObjectStore OSAX technology and the term “OSAX” is used in many places in this document. The term “OSAX” is also used for many of the class, object, and directory names used by OSNCI.
Installing ObjectStore .NET COM Interop

ObjectStore .NET COM Interop is installed automatically when you install ObjectStore.

System Requirements

Minimum system requirements for ObjectStore .NET COM Interop are

- One of the following operating systems:
  - Microsoft Windows XP Service Pack 2
  - Microsoft Windows 2003 Server
- Microsoft Visual Studio .NET 2003
- ObjectStore 6.3 for Windows

Setting Up the OSNCI Environment

Before you use OSNCI you need to set the environment to specify where the library and include files are located. You can do this either by specifying the environment in Visual Studio or by setting the INCLUDE and LIB environment variables you (or you can do both).

Setting the Environment in Visual Studio

To specify your environment from Visual Studio .NET 2003:

1. From the Visual Studio .NET menu select Tools->Options->Projects->VC++ Directories.
2. At the Options dialog add the following:
   - c:\odi\osax\bin to Executable Files.
   - c:\odi\osax\lib to Library Files.
   - c:\odi\osax\include to Include Files.
3. Click OK.

Setting the INCLUDE and LIB Environment Variables

To specify your environment by setting the INCLUDE and LIB environment variables:

1. From the Windows menu select Start->Settings->Control Panel->System->Advanced->Environment Variables.
2. Edit the LIB environment variable by adding c:\odi\osax\lib.
3. Edit the INCLUDE environment variable by adding c:\odi\osax\include.
4. Click OK twice.
Examples

ObjectStore .NET COM Interop includes example programs and databases. Each of the examples contains a Visual C++ 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 OSNCI, and a Microsoft .NET application written in C# that uses that object server to display the contents of the database.

Build this object server in the release project configuration. This object server can then be used directly by the Microsoft .NET C# application in the Books.NET.Client subdirectory. The .NET 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 OSNCI object server; see A Tour of the Books Example on page 12.

A Tour of the Books Example

This section introduces you to the elements of an OSNCI project and to the process of creating a project, using the Books example that comes with OSNCI. The Books example uses ActiveX with C# to access objects in a prebuilt ObjectStore database of book titles and authors. After building the OSNCI object server for this example, you can access the database from C# and display the following form:
Chapter 1: Overview of ObjectStore .NET COM Interop

The 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 OSNCI object server for the Books example is built using a Visual C++ project. The Books example project files are in the directory osax\examples\books\bookServer. The solution file bookServer.sln contains all the C++ source code for the project, as well as the type description file and database schema definition. You can examine, modify, and build that project by using Microsoft Visual C++. See Building an Object Server on page 17 for in-depth instructions on building OSNCI servers.

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 COM 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 that are generated by OSNCI 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 OSNCI 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 OSNCI Object Servers, on page 17.

Note

OSNCI also supports the Microsoft Active Template Library (ATL 3.0) as a way to create COM interfaces for persistent objects stored in an ObjectStore database. Using ATL 3.0 instead of type description files requires experience with ATL 3.0 and C++, but it provides more flexibility and control when you are customizing COM interfaces.

The Microsoft .NET C# Project

The solution file Books.NET_2003.sln is located in the osax\examples\books\Books.NET.Client directory. This file contains the C# project that enables you to build a .NET application to access the database. The C# project for the Books example does (among other things) the following:

- Defines the design of a WinForms application used to display objects from the database.
- References OSAXBooksLib which is a .NET/COM Interop assembly that uses the OSAX.Books object server.
- Provides the code for the C# methods that access the object server.
Global properties

The OSAX.Books object server provides two global properties to the C# project:

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

- **CBookElt**, an COM object representing a class of objects stored in the database.

A method from the C# application

The following method from the C# application uses both of the global properties, ObjectStore and CBookElt:

```csharp
private void displayBookList()
{
    OSAXBooksLib.IBookElt bookElt = null;
    OSAXBooksLib.IBookElt nextBookElt = null;
    OSAXBooksLib.IBook book = null;
    string strAuthor;
    string strTitle;
    ostore.BeginTrans(true);
    OSAXBooksLib.IOSAXType t = (OSAXBooksLib.IOSAXType)booksServer.CBookElt;
    bookElt = (OSAXBooksLib.IBookElt)db.get_Value("Books", t);
    do
    {
        strAuthor = book.Author.Name.Value;
        strTitle = book.Name.Value;
        string nItem = strAuthor + " - " + strTitle;
        listBox1.Items.Add(nItem);
        nextBookElt = bookElt.Next;
        releaseComObject(bookElt);
        bookElt = nextBookElt;
        releaseComObject(book);
    } while(bookElt != null);
    ostore.CommitTrans();
    releaseComObject(bookElt);
    releaseComObject(t);
}
```

The following statement in the method:

```csharp
ostore.BeginTrans(true);
```

uses ObjectStore to start a transaction in the database. Similarly, the following statement:

```csharp
ostore.CommitTrans();
```

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

The following statements

```csharp
OSAXBooksLib.IOSAXType t = (OSAXBooksLib.IOSAXType)booksServer.CBookElt;
bookElt = (OSAXBooksLib.IBookElt)db.get_Value("Books", t);
```

use CBookElt to specify 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 .NET 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 C# properties and syntax used to access them:

<table>
<thead>
<tr>
<th>C# Code</th>
<th>Code in books.ost</th>
<th>C++ Class Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>IAuthor Author; string AuthorName; IOSAXType t = (IOSAXType) bookServer.CAuthor; Author = (IAuthor) osDB.get_Value(&quot;SampleAuthor&quot;, t); AuthorName = Author.Name;</td>
<td>class Author { [propget] char* Name() data name; ... };</td>
<td>class Author { public: char <em>name; Author</em> author; };</td>
</tr>
<tr>
<td>IBook Book; string BookName; IAuthor BookAuthor; IOSAXType t = (IOSAXType) bookServer.CBook; Book = (IBook) osDB.get_Value(&quot;SampleBook&quot;, t); BookName = Book.Name; BookAuthor = Book.Author;</td>
<td>class Book { [propget] char* Name() data name; [propget] Author* Author() data author; ... };</td>
<td>class Book { public: char* name; Author* author; };</td>
</tr>
<tr>
<td>IBookElt Books; IBook Book; IOSAXType t = (IOSAXType) bookServer.CBookElt; Books = (IBookElt) osDB.get_Value(&quot;SampleBooks&quot;, t); Do { Book = Books.Book; Books = Books.Next; } While (Books != null);</td>
<td>class BookElt { [propget] Book* Book() data book; [propget] BookElt* Next() data next; ... };</td>
<td>class BookElt { public: Book book; BookElt* next = NULL; };</td>
</tr>
</tbody>
</table>

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 by using its defined property names.

Note: Constants for the loop differ in C++ and C#. The C++ NULL pointer value is translated as Null in COM and as null in C#.
Chapter 2
Building OSNCl Object Servers

An OSNCl 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.

As a COM server, an OSNCl server provides .NET client applications with the ability to access ObjectStore databases via .NET/COM Interop. With .NET/COM Interop, .NET common language runtime (CLR) components can access COM components that are wrapped as .NET Components. The .NET framework’s interoperability layer handles all of the plumbing issues between the managed and unmanaged worlds.

You can convert the type definitions in your COM type library into the equivalent metadata definitions in a common language runtime assembly from within a Visual Studio .NET project. This method is described in Building an Object Server on page 17.

You can also use the Microsoft .NET Type Library Importer (tlbimp.exe) to convert type definitions found within your COM type library to the metadata in a .NET assembly. The output of tlbimp.exe is a .NET assembly that contains the runtime metadata for the types defined within the original type library.

For more information about importing Type Libraries as an .NET assembly, please refer to the following document available from Microsoft:


The following sections provide detailed information for building an OSNCl 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 OSNCl object server. The procedure described here builds a version of the books example introduced in A Tour of the Books Example on page 12. You should create a new
directory for this project, such as C:\odi\osax\examples\mybooks in order to not overwrite the books example.

The instructions in this section are specific to Visual Studio .NET 2003.

To build an object server, perform the following steps:

1. Use Microsoft Visual Studio .NET to create a C++ skeleton project.
   a. From the Visual Studio .NET menu, select File->New Project.
   b. For Project Type, select Visual C++ Projects.
   c. For Template, select Win32 Project.
   d. For Name, enter bookServer.
   e. For Location, enter C:\odi\osax\examples\mybooks.
   f. Click OK. The Win32 Application Wizard starts.
   g. In the Wizard, for Application Settings, select DLL.
   h. Check the Empty Project check box.
   i. Click Finish.

2. Define the interfaces to your C++ objects in a type description (.ost) file. This file defines the COM interfaces and object implementations in terms of C++ objects. For the book example, the type description file should contain the following code:

```plaintext
[ helpstring("ObjectStore OSAX books example 3.0"),
  lcid(0x0000),
  version(3.0),
  objectstore
 ]
library OSAXBooksLib
{
  [ helpstring("ObjectStore OSAX books example"),
    version(3.0),
    appobject,
    progid("OSAX.Books.3","OSAX.Books"),
    interface(IOSAXBooksServer)
  ]
  object OSAXBooksServer
  {
    [ helpstring("Interface to ObjectStore C++ class OSAXObjectStore"),
      propget
    ]
    IOSAXObjectStore* ObjectStore() class OSAXObjectStore;
  [ helpstring("Interface to C++ class BookElt"),
      propget
    ]
    IBookEltClass* CBookElt() class BookElt;
  [ helpstring("Interface to C++ class Book"),
      propget
    ]
```
Chapter 2: Building OSNCI Object Servers

IBookClass* CBook()class Book;
[  helpstring("Interface to C++ class Author"),
    propget
]
IAuthorClass* CAuthor()class Author;
[propget]
IOSAXStringConstructor* CString()class char;
[propget]
BSTR ObjectServerDirectory()function get_server_directory;
};

class Book
{
    [propget]
    char* Name()data name;
    [propget]
    Author* Author()data author;
    [class]
    Book* Create(IOSAXStorage* location, Author* a,
                 char* t) new(location) Book(a, t);
};
class Author
{
    [propget]
    char* Name()data name;
    [class]
    Author* Create(IOSAXStorage* location, char* name)
                 new(location) Author(name);
};
class BookEl{t
{
    [propget]
    Book* Book()data book;
    [propget]
    BookEl* Next()data next;
    [class]
    BookEl* Create(IOSAXStorage* location, Book* b,
                   BookEl* next) new(location) BookEl(b, next);
};

For more information about the .ost file, see Type Description File on page 28.

For an example of a type description file, see
C:\odi\osax\examples\books\books_no_uuid.ost.

3 In a command window, add the necessary unique IDs to the type description file by running the osgentyp utility on the .ost file using the following syntax:
   osgentyp /dt books_no_uuid.ost /u books.ost
   The /u option to osgentype creates a new type description file books.ost with the necessary IDs.

4 In a command window, generate the interface, COM, and resource files by running the osgentyp utility on the new type description file using the following syntax:
Building an Object Server

```
oosgentypl /dt books.ost /c Author.h /c Book.h /c BookElt.h /l bookServer.tlb /t books.idl /i books_imp.cpp
```

The `osgentypl` 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; this C++ code is unmanaged code.

The .idl file is processed by Microsoft’s `idl` compiler, `midl`. 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

For more information about the `osgentypl` utility, see The `osgentypl` Utility on page 45.

5 In Visual Studio .NET, set the Solution Configuration to Release.
6 In the Solution Explorer pane, right click the bookServer project and select Add->Add New Folder and name it ObjectStore ActiveX Files.
7 Repeat step 6 to add a new folder named Generated Files.
8 Repeat step 6 to add a new folder named ObjectStore Schema Files.
9 In the Solution Explorer add the type description file as follows:
   a Right click the ObjectStore ActiveX Files folder and select Add->Existing Item from the drop down list.
   b From the Add Existing Item dialog, select the type description file books.ost (for the books example) and click Open.
10 Add a Custom Build Rule for the books.ost file by right clicking the file and selecting Properties from the drop down menu. Add the following for the Custom Build Step:
   a For Command Line add
      ```
      osgentypl /DT $(InputDir)$(InputName).ost /C $(InputDir)stdafx.h /L $(InputName) /O $(InputName) /T $(InputDir)$(InputName).idl /I $(InputDir)$(InputName)_imp.cpp
      ```
   b For Description add
      Generating IDL & CPP files from OST file
   c For Outputs add
      ```
      $(InputDir)$(InputName).idl;$(InputDir)$(InputName)_imp.cpp
      ```
   d Click OK.
11 Create a new module definition file as follows:
Chapter 2: Building OSNCI Object Servers

1. In the Solution Explorer right click the ObjectStore ActiveX Files folder and select Add->Add New Item from the drop down menu.

2. In the Add New Item dialog, for Categories, select the Visual C++ folder and for Template, select Module-Definition File (.def).

3. Name the file books.def and set the location to C:\ODI\osax\examples\mybooks\bookServer and click Open.

12 Double click the books.def file to edit it and add the following code:

LIBRARY bookServer

EXPORTS
DllCanUnloadNow PRIVATE
DllGetClassObject PRIVATE
DllRegisterServer PRIVATE
DllUnregisterServer PRIVATE

13 Create the necessary header files for the server application. For the books example, you need Author.h, Book.h, and BookElt.h.

a The Author.h header file should contain the following code:

#pragma once
class Author
{
public:
    char *name;
    Author() {};
    Author (char *n) { name = n; }
    virtual ~Author() {};
    static os_typespec *get_os_typespec();
};

b The Book.h header file should contain the following code:

#pragma once
#include "Author.h"
class Book
{
public:
    Author *author;
    char *name;
    Book() {};
    Book (Author *a, char *t) { author = a; name = t; }
    virtual ~Book() {};
    static os_typespec *get_os_typespec();
};
Building an Object Server

The `BookElt.h` header file should contain the following code:

```c
#pragma once
#include "Book.h"
class BookElt
{
public:
  BookElt *next;
  Book *book;
  BookElt() { };
  BookElt (Book *b, BookElt *e) { book = b; next = e; }
  virtual ~BookElt() { };
  static os_typespec *get_os_typespec();
};
```

14 In the Solution Explorer, add your application header files from step 13. For the books example, add the header files as follows:

a Right click the Header Files folder and select Add->Add Existing Item from the drop down menu.

b From the Add Existing Item dialog, select the `Author.h`, `Book.h`, and `BookElt.h` files and click Open.

15 Create the source code files for your application; for the books example the code is contained in a file named `servutil.cpp`. The `servutil.cpp` file should contain the following code:

```c
#define STRICT
#define _WIN32_WINNT 0x0400
#define _ATL_APARTMENT_THREADED

#include <osax/osaxbase.h>
extern OSAX::COSAXModule _Module;

// Utility function to return pathname of server DLL, exposed and
// used by controllers to locate companion databases
BSTR get_server_directory ()
{
  HMODULE hmodule = _Module.m_hInstResource;
  TCHAR strModName[_MAX_PATH];
  DWORD modLen = GetModuleFileName(hmodule, strModName,
    sizeof(strModName)/sizeof(TCHAR));
  TCHAR drive[_MAX_DRIVE];
  TCHAR dir[_MAX_DIR];
  TCHAR fname[_MAX_FNAME];
  TCHAR ext[_MAX_EXT];
  _tsplitpath(strModName,drive,dir,fname,ext);
  // Move to the parent directory
  int dirLen = (int) strlen(dir);
  if (dirLen > 2)
  {
    int count = 2;
    int i;
    for(i=dirLen-2;i>=0;i--)
    {
      if (dir[i] == '\')
      {
        count--;
      }
```
if (count == 0)
{
    dir[i+1] = '\0';
    break;
}

_tmakepath(strModName, drive, dir, 0, 0);
unsigned long size = (unsigned long) strlen(strModName);
BSTR result = SysAllocStringLen(0, size);
mbstowcs(result, strModName, size+1);
return result;

16 In the Solution Explorer, add the ObjectStore application source file(s) as follows:
   a Right click the Source Files folder and select Add->Existing Item from the drop down list.
   b From the Add Existing Item dialog, select the appropriate application source files. For the books example select servutil.cpp and click Open.

17 In the Solution Explorer, add the new stdafx.h header file as follows:
   a Right click the Header Files folder and select Add->New Item from the drop down menu.
   b In the Add New Item dialog, for Categories, select the Visual C++ folder and for Template, select C++ File (.h).
   c Name the file stdafx.h, set the location to C:\ODI\osax\examples\mybooks\bookServer, and click Open.

18 In the Solution Explorer, double click the stdafx.h file to edit it and add the following code:
#include "Author.h"
#include "Book.h"
#include "BookElt.h"

BSTR get_server_directory();

19 In the Solution Explorer, create the ObjectStore schema file as follows:
   a Right click the ObjectStore Schema Files folder and select Add->New Item from the drop down menu.
   b In the Add New Item dialog, for Categories, select the Visual C++ folder and for Template, select C++ File (.cpp).
   c Name the file books_ostore.scm, set the location to C:\ODI\osax\examples\mybooks\bookServer, and click Open.

20 In the Solution Explorer, double click the books_ostore.scm file to edit it and add the following code:
#include <ostore/ostore.hh>
#include <ostore/manschem.hh>
#include "Author.h"
#include "Book.h"
#include "BookElt.h"
OS_SCHEMA_DLL_ID("OSAX:OSAX.Books.3");
OS_MARK_SCHEMA_TYPE(Book);
OS_MARK_SCHEMA_TYPE(Author);
OS_MARK_SCHEMA_TYPE(BookElt);

The OS_MARK_SCHEMA_TYPE macros are the standard way of specifying persistent objects for ObjectStore applications. The values are the names of the persistent objects.

The OS_SCHEMA_DLL_ID macro is the standard way of specifying a component schema for the named DLL. The value should agree with the progid attribute specified in the application's type description file (books.ost, in the books example). For more information on the progid attribute, see Type Description File on page 28.

For more information on creating ObjectStore schema files, see “Creating Schema Source Files” in Chapter 2 of Building ObjectStore Applications.

21 Add a Custom Build Rule for the schema source file by right clicking the books_ostore.scm file and selecting Properties from the drop down menu. Add the following for the Custom Build Step:

a For Command Line add
$(OS_ROOTDIR)\bin\ossg -asdb $(OutDir)\schema_ostore.adb
-asof $(IntDir)\schema_ostore.obj $(InputPath) /I $(InputDir)
/I $(OS_ROOTDIR)/include /MD /D_DLL /DWIN32

b For Description add
Generating ObjectStore schema

c For Outputs add
$(IntDir)\schema_ostore.obj;$(OutDir)\schema_ostore.adb

d Click OK.

22 Add the resource file generated by the osgentyp utility in step 4 above as follows:

a Right click the Resource Files folder and select Add Existing Item from the drop down list.

b From the Add Existing Item dialog, select (for the books example) booksosax.rc and click Open.

23 Add a preprocessor definition to the application resource file as follows:

a Right click the resource file (booksosax.rc in the books example) and select Properties from the drop down menu.

b In the Property Pages dialog, for Configuration Properties->Resources->General->Preprocessor Definitions, add APSTUDIO_INVOKED and click OK.

24 Add the other files generated by the osgentyp utility in step 4 above as follows:

a Right click Generated Files folder and select Add Existing Item from the drop down menu.
b Select (for the books example) books.idl, OSAXBooksServer.rgs, and books_imp.cpp and click Open.

25 For the generated .idl file, specify the properties to be used by the Microsoft midl compiler by right clicking the books.idl file (for the books example) and selecting Properties from the drop down menu. Add the following:
   a For MIDL->General->Generate Stubless Proxies select Yes(/Oicf; Same as /Oif.
   b For MIDL->Output->Header File enter $(InputName).h.
   c For MIDL->Output->IID File enter $(InputName)_i.c.
   d Click OK.

26 Add a preprocessor definition to the generated _imp.cpp file as follows:
   a Right click the _imp.cpp file (books_imp.cpp for the books example) and select Properties from the drop down menu.
   b In the Property Pages dialog, for Configuration Properties->C/C++->Preprocessor->Preprocessor Definitions, add _ATL_DLL and click OK.

27 Specify project level properties by right clicking the bookServer project (for the books example) and selecting Properties from the drop down menu. Add the following:
   a For Configuration Properties->C/C++->Preprocessor->Preprocessor Definitions, add _ODI_FORCE_OSTORE_LIB.
   b For Configuration Properties->C/C++->Code Generation, select Multi-threaded DLL (/MD).
   c For Configuration Properties->Linker->Input->Module Definition File, specify books.def (for the books example).
   d For Configuration Properties->Custom Build Step->General->Command Line, add the following:
      echo regsvr32 /s /c "$(TargetPath)"
      regsvr32 /s /c "$(TargetPath)"
      echo regsvr32 exec. time > $(OutDir)\regsvr32.trg
   e For Configuration Properties->Custom Build Step->General->Description, add the following:
      Registering ActiveX Control
   f For Configuration Properties->Custom Build Step->General->Outputs, add the following:
      $(OutDir)\regsvr32.trg

28 From the Visual Studio .NET menu, select Build->Rebuild Solution.

   In addition to building the OSNCI object server, this process also registers the OSNCI server.
Accessing an Object Server

In order for .NET client applications to make use of an OSNCI object server, the .NET client needs to reference the server. You can specify this reference from Visual Studio .NET. For example, to access the bookServer object server from the .NET client application in the books example:

1. In Visual Studio .NET 2003, open the c:\odi\example\books\Books.NET.Client\Books.NET_2003.sln solution file.
2. In the Solution Explorer right click References and select Add Reference from the drop down menu.
3. In the Add Reference dialog click Browse, navigate to the C:\ODI\osax\examples\mybooks\bookServer\Release directory.
4. In the Add Component dialog select bookServer.tlb and click Open.
5. In the Add Reference dialog click OK.
6. From the Visual Studio menu select Build->Rebuild Solution.

Basic Operations Supported by an Object Server

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

<table>
<thead>
<tr>
<th>Object Type</th>
<th>C# Syntax</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>IOSAXObjectStore</td>
<td>ObjectStore.OpenDatabase(pathname,</td>
<td>Open the specified database.</td>
</tr>
<tr>
<td></td>
<td>ReadOnly,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CreateMode,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>schemadatabase)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ObjectStore.BeginTrans(ReadOnly)</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.get_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 OSNCI 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 OSNCI 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>.NET Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>char, int, long</td>
<td>I4</td>
<td>System.Int32</td>
</tr>
<tr>
<td>float, double</td>
<td>R4 or R8</td>
<td>System.Double</td>
</tr>
<tr>
<td>char*, signed char*, unsigned char*, char[]</td>
<td>BSTR (Literal values)</td>
<td>System.String</td>
</tr>
<tr>
<td></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 pointers</td>
<td>Generated interface from typelib, if available; else IDispatch</td>
<td>Generated object type, or Object</td>
</tr>
<tr>
<td>NULL (void* == 0)</td>
<td>Null</td>
<td>Null</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 subclasses</td>
<td></td>
<td></td>
</tr>
<tr>
<td>os_dictionary*</td>
<td>IOSAXDatabase*</td>
<td>IOSAXDatabase</td>
</tr>
<tr>
<td>os_set*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>os_database*</td>
<td>IOSAXDatabase*</td>
<td></td>
</tr>
</tbody>
</table>

Note: C++ character arrays require special treatment. See Accessing and Creating Character Strings on page 39.

Object Server Configuration

An OSNCI 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 OSNCI servers, or you can use the ATL interface to OSNCI to create an OSNCI server as an executable or as a Windows XP service.

OSNCI object servers use a COM threading model of both, which means that they are suitable for single-threaded and multithreaded clients. Object servers expose each C++ class with a single dual automation interface, making them accessible to scripting languages. The ATL interface to OSNCI can be used to expose multiple custom interfaces.

For information about support for multisession applications, see Multisession Support on page 47.

Type Description File

You define an OSNCI object server by writing a type description file for processing by the osgentyp utility (see The osgentyp Utility on page 45). 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 OSNCI and ATL Concepts on page 47.

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}

attributes

Is a comma-separated list of the following:

helpstring(string)
Associates string with the generated type library. string specifies textual information about the OSNCI 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 45); or you can supply a GUID generated by 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 30).

Example
The following is the library specification from books.ost in the Books example:

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

Object Specification

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

[attributes] class typename {method-specifications };
[attributes] object coclass {method-specifications };

class typename

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

object coclass

Defines 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

Specifies that the object or instance class should be exposed as an appobject. The appobjects are automatically created when a .NET application starts. Their methods appear at global scope. This attribute is optional and is meaningful only when specified with the object keyword.

appinstance

Specifies that the instance container should be exposed as an appobject. The appobjects are automatically created when a .NET application 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 45.

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.

```plaintext
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.

```plaintext
factoryuuid(uid)
```
Associates `uid` with the class factory being defined. This attribute is optional and is meaningful only in defining a class factory; see the `progid` attribute. 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 45`. For information about setting the UUID for the instance, see the `instancefactoryuuid` attribute.

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

```plaintext
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 in defining an instance container. See the `progid` attribute for the meaning of `DepName` and `IndepName`.

```plaintext
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 45`.

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

```plaintext
isomorphic(boolean)
```
Overrides the default value set in the Library Specification on page 28. If `boolean` is true, the C++ object identity implies object identity for instance objects.

```plaintext
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.” 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 45.

version(major[.minor])
Specifies the version number for the interface. For more information, see the
version attribute for the Library Specification on page 28.

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 COM interfaces define method IDs explicitly.
See the COM 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
.NET, 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 COM documentation for
additional information.

propputref
Specifies that the method is a property setter. See the COM documentation for
additional information.
restricted
Specifies that the method is restricted. See the COM 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 45.

MethodName
Is the name of the method as it appears to the COM 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 45. The variables in the argument list are available as C++ types for use in the method implementation.

implementation
Tells osgentyp how to implement the COM 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.

  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.

  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.

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.

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:

```
[  
  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 IOSAXObjectStore;
  [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 COM interface IOSAXObjectStore, implemented by the C++ class named IOSAXObjectStore. This interface and this class are part of the OSNCI 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:

```cpp
[uuid(326D9EC6-4012-11D1-B9C3-0800091AAA11),
classuuid(326D9EC9-4012-11D1-B9C3-0800091AAA11)]
class Book
{
    [propget]
    char* Name() data name;
    [propget]
    Author* Author() data author;
    [class]
    Book* Create(IOSAXStorage* location, Author* a, char* t) new(location) Book(a, t);
};
```

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 COM interface that
osgentyp automatically generates for the C++ type char* to represent the C++ data
member named name. Author uses an COM 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 OSNCl, 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 ASP .NET 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 .NET you can initialize a session variable to
Creating Objects

Creating Objects

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:

```cpp
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 OSNCI. 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:

```cpp
void SetValue(IUnknown* pUnk) function SetDataImpl(pUnk);
```

Creating Objects

An OSNCI 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. OSNCI simply exposes this method on the corresponding types. Following is the C++ and .ost code for this approach:

**C++ Definition**

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

**Definition in .ost File**

```ost
class Bookshelf {
    Book *Add(char *title) method
    AddBook(title);
}
```

Given these definitions, a C# program could create a new book by using syntax like the following:

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

**C++ Definition**

```cpp
class Book {
    public:
    static Book *create(char*title);
}
```

**Definition in .ost File**

```ost
class Book {
    [class]
    Book *Create(char *title) method create(title);
}
```

Every type exposed by OSNCI 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 COM 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:

```ost
object OSAXBookExample
{
    IOSAXObjectStore* ObjectStore() class OSAXObjectStore;
    [propget] IBookClass* CBook() class Book;
```
Creating Objects

[propget] IAuthorClass* CAuthor() class Author;
};

Given these definitions, a C# program could create and access a new book by using syntax like the following:

IBook = CBook.Create(newTitle);
System.Console.WriteLine("The new book title is {0}", 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 OSNCI 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:

C++ Definition

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

Definition in .ost File

```cpp
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 ObjectStore’s overloaded new operator, as described in the ObjectStore C++ API Reference. 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, indicating 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, C#), 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:

```
myObject X;
X = GetOneOfMyObjects;
```
SetCleanup(X);

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

```c
void DeleteOnRelease() function DeleteDataOnFinalRelease(TRUE);
```

## Accessing and Creating Character Strings

OSNCI 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, strings are considered literal values without identity. The OSNCI 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.

### C++ Definition

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

### Definition in .ost File

```c
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);
};
```

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

```c
IAuthor Author;
IAuthor Copy;
WriteLine("The author’s name is " + Author.Name);
Copy.Name = Author.Name;
```

The `WriteLine` statement requires a .NET 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 by using the `IOSAXString` interface. This is so the property can be copied to another object (for example, `Copy.Name`) without losing its identity. The result is that the persistent object representing `Copy` contains exactly the same `char*` character string as the persistent object representing `Author`. 

```c
References: Release 6.3
```
Creating Persistent Character Strings

OSNCI provides a mechanism for creating and initializing persistent and transient strings. OSNCI 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 char*. The strings are exposed using the IOSAXString interface, and they are created using the IOSAXStringConstructor interface.

An IOSAXStringConstructor interface represents the class object for IOSAXString. It is typically exposed at the top level of an object server, as in the following example. This example .ost object definition provides class objects named CAuthor and CString for the Author objects described in Accessing and Creating Character Strings on page 39.

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

The CString class used in this example implements three methods:

- **IOSAXStringInterface::Ansi** creates a new string in the specified location and initializes it by using the supplied ActiveX Unicode string translated to 8-bit representation via the ANSI code page.

- **IOSAXStringInterface::OEM** creates a new string in the specified location and initializes it by using the supplied ActiveX Unicode string translated to 8-bit representation via the OEM code page. If you do not know whether to use OEM or Ansi, use Ansi.

- **IOSAXStringInterface::Copy** creates a new string in the specified location and initializes it by using the supplied IOSAXString, which is already in 8-bit form.

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

```csharp
Private IAuthor CreateAuthor(string name) {
    IOSAXString osName = CString.Ansi(name, osDatabase);
    IAuthor CreateAuthor = CAuthor.Create(osDatabase, osName);
}
```

Accessing Collections

To expose ObjectStore collections defined using the template classes os_Set, os_Bag, os_List, and os_Dictionary through OSNCI, 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;
}
```

At run time, all ObjectStore collections are represented by 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 67.

### Defining a Collection

To define and expose a collection class in the OSNCI `.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 OSNCI `.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*>;
};
```

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.

```cpp
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. OSNCI provides the ActiveX interface from the

---

**Release 6.3**
interface for the os_Collection base class. The class interface() 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 C# 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 Book objects are then created and inserted
into the collection.

Private CreateLibrary() {
  Root = osDatabase.CreateRoot("Books");
  Root.Value = CBookList.Create(osDatabase);
  Root.Type = CBookList;
  Library = Root.Value;
  Library.Add CreateBook("Struggling Upward", "Alger, Horatio Jr.");
  Library.Add CreateBook("Peter Pan", "Barrie, James Matthew");
  Library.Add CreateBook("The Wonderful Wizard of Oz", "Baum, L Frank");
}

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 C#
into the database. The standard ANSI code page is used for character conversion.

IBook CreateBook(string Title, string Author) {
  IOSAXString osTitle = CString.Ansi(Title, osDatabase);
  IOSAXString osAuthor = CString.Ansi(Author, osDatabase);
  IBook newBook = CBook.Create(osDatabase, osTitle, osAuthor);
  Return newBook;
}

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 C#, the iteration syntax is the Foreach construct, as in
the following example:

foreach(IBook book in MyBookShelf.Books)
{
  WriteLine("{0}", book.Title);
}
Querying Collections

The IOSAXCollection interface allows a collection to be queried. Use the ObjectStore collection query language documented in the C++ Collections Guide and Reference.

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

"type:query-expression"

For example:

```java
ResultSet = ASetInstance.MemberSet.Evaluate("A*:Num == 111");
```

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.

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++ method `os_database::create_root()`, or use an object server by means of 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 (null in C#, NULL in C++)

```java
osDatabase = ObjectStore.CreateDatabase(    cdFilename.filename, null, true, null);
ObjectStore.BeginTrans(false);
IOSAXDatabaseRoot Root;
    // Create the database root, representing an empty list of books
Root = osDatabase.CreateRoot("Books");
    // Set root type before initializing root value
Root.Type = CBookElt;
Root.Value = null;
CreateBookList();
ObjectStore.CommitTrans();
```

Once a database root has been created, you can access and update its value by 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.

```c++
void InsertBook(IBook book) {
    IBookElt osElement;
    IBookElt Head;
    Head = osDatabase.Value("Books", CBookElt);
    osElement = CBookElt.Create(osDatabase, Book, Head);
    osDatabase.Value("Books", CBookElt) = osElement;
}
```

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. Your OSNCI 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 an 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) {
    function set_my_name (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 exception and convert it to a COM 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 the current transaction to be aborted and then 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. OSNCI has finished aborting the transaction. You do not need to perform any ObjectStore-specific cleanup.

- `OSAXETixError (&H80041001)`
  - An unspecified exception has occurred. Look at the C# exception 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.

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 function. This function is called by a function that sets up the transaction and an error handler:

```csharp
Private void TransactionExample() {
    int count;
    bool tryTxn = true;
    while (tryTxn) {
        try {
            os.BeginTrans(false);
            Transaction Body();
            os.CommitTrans();
            tryTxn = false;
        } catch (COMException e) {
            if (e.ErrorCode == (int) OSAXExceptionCode.OSAXETixError) {
                tryCount++;
                ostore.Rollback();
                if (tryCount >= 10) throw e;
            } else {
                throw e;
            }
        }
    }
}
```

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.

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.
The osgentyp Utility

/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.

/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.

Multisession Support

OSNCI supports multiple concurrent sessions for increased performance and scalability. Multisession support allows client threads to use more than one concurrent session without being blocked from entering OSNCI. Multisession support is accomplished by an internally maintained pool (the session pool) of ObjectStore sessions. Multisession clients can use the session pool by means of the IOSAXSessionPool interface, as described in IOSAXSessionPool on page 58.
Multisession Support
Chapter 3
Distributed OSNCI Object Servers

An OSNCI 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 50.
- Use DCOM to directly access the remote object server; see Using DCOM to Access Networked Object Servers on page 51.
Using Object Servers with Web Servers

The advantages and disadvantages of these configurations are

**Web server accesses local OSNCI object server:**

<table>
<thead>
<tr>
<th>Protocols</th>
<th>HTTP to access Web pages InProc COM within server</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Advantages</strong></td>
<td>Supports all Web browsers.</td>
</tr>
<tr>
<td></td>
<td>Access to very large centralized databases.</td>
</tr>
<tr>
<td></td>
<td>Centralized database can be easier to update and administer.</td>
</tr>
<tr>
<td></td>
<td>High-performance data access on server side.</td>
</tr>
<tr>
<td><strong>Disadvantages</strong></td>
<td>User interface is restricted to HTML constructs.</td>
</tr>
<tr>
<td></td>
<td>Applications with extensive computation requirements must perform them on the server (where the data is), instead of distributing burden to clients.</td>
</tr>
</tbody>
</table>

**Client accesses remote object server:**

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

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 OSNCI object server and database can be accessed directly from an Active Server Page by 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.

```
<OBJECT RUNAT=Server ID="osBooks"
CLASSID="CLSID:84858E05-533D-11D0-8771-000000000001">
```
Using DCOM to Access Networked Object Servers

An OSNCI 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 OSNCI 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).

```plaintext
[HKEY_ROOT_CLASSES\CLSID\{clsid}]
"AppID" = "{clsid}"
```
Chapter 4
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.

OSNCCI Interfaces at a Glance

The following table provides a quick reference of equivalent interfaces:

<table>
<thead>
<tr>
<th>COM Interface</th>
<th>Methods and Properties</th>
<th>ObjectStore C++ API or Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>IOSAXCollection</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Item (default)</td>
<td>os_collection::pick()</td>
<td></td>
</tr>
<tr>
<td>Add</td>
<td>os_collection::insert()</td>
<td></td>
</tr>
<tr>
<td>Remove</td>
<td>os_collection::remove()</td>
<td></td>
</tr>
<tr>
<td>ElementType</td>
<td>C.ElementType = XClass</td>
<td></td>
</tr>
<tr>
<td>Cardinality() As Long</td>
<td>Returns number of elements</td>
<td></td>
</tr>
<tr>
<td>Count() As Long</td>
<td>Returns number of elements</td>
<td></td>
</tr>
<tr>
<td>IsEmpty() As Boolean</td>
<td>os_collection::empty()</td>
<td></td>
</tr>
<tr>
<td>IsOrdered() As Boolean</td>
<td>false</td>
<td></td>
</tr>
<tr>
<td>Evaluate</td>
<td>os_collection::query()</td>
<td></td>
</tr>
<tr>
<td>IOSAXDatabase</td>
<td>Open</td>
<td>os_database::open()</td>
</tr>
<tr>
<td>OpenMVCC</td>
<td>os_database::open_mvcc()</td>
<td></td>
</tr>
<tr>
<td>Close</td>
<td>os_database::close()</td>
<td></td>
</tr>
<tr>
<td>Value</td>
<td>Name As String, type As IOSAXType</td>
<td></td>
</tr>
</tbody>
</table>
The following sections describe the interfaces in more detail. The interfaces are listed in alphabetical order.

## IOSAXCollection

### Purpose

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

When `osgentyp` defines a COM 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

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

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` argument. Otherwise, `type` is assumed to be correct.

## IOSAXObjectStore

### Purpose

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

### 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 by using `os_transaction::begin()`.

**CommitTrans**

Commits a transaction by 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.

The `SchemaDatabase` specifies a database to use for the schema.

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

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

**Initialize([Reserved As Variant])**

Initializes 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.
IOSAXSessionPool

Purpose

The IOSAXSessionPool interface provides multisession applications with control over the session pool; see Multisession Support on page 47.

Properties and Methods

PoolSize As long

Puts or gets the pool size property. The default value is 1. PoolSize should be set to a value near the anticipated maximum number of concurrent OSAX sessions.

Initialize()

Initializes the session pool. This method is optional; pool initialization occurs automatically when ObjectStore is initialized.

IsInitialized As boolean (read-only)

Returns true if the session pool has been initialized.

IOSAXStorage

Purpose

The IOSAXStorage interface is a storage allocator for OSNCI. It provides a generic interface to the OSNCI storage management for use when creating objects.

The IOSAXStorage interface has no methods. To use it, you pass it as an argument to a top-level server object’s instance creation method, specifying the storage location (IOSAXDatabase or IOSAXSegment) to use when the object is allocated. Objects of the
types IOSAXDatabase and IOSAXSegment can be queried for their IOSAXStorage interface.

For information about the IOSAXDatabase and IOSAXSegment ActiveX data types, see Basic Types Supported by an Object Server on page 27 in Chapter 2, Building OSNCI Object Servers, on page 17.

**IOSAXString**

**Purpose**

Visual Basic and COM use BSTRs to represent strings, while C++ uses char*. If OSNCI 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.

OSNCI 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 by 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**
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++.
A
ActiveX
   architecture 10
   Automation 9
   data types 27
   exceptions 32
   overview of ActiveX Interface 9
Ansi code page 40
   ANSI method
target description 59
appinstance attribute 30
appobject attribute 30
ATL
   interface 28
attributes
      library specification 29
      method specification 32
      object specification 30
   automatic type-checking 15
   automation
      interfaces 27
Automation server, configurations 10
B
basic types 27
   BeginInitialization method 56
   BeginTrans method
      Books example 14
      C# syntax 26
target description 56
Books example 12
      database objects, defining 15
      directory 13
      object server 13
      type description file 13
      Visual C++ project 13
      Visual Studio .NET project 13
Both threading model 28
BSTR ActiveX data type 27
BSTR, Unicode 39
building object servers 18
C
/c option 45
C++
   constructors, exposing 38
   exceptions 32
   implementation file, specifying name 46
   static member functions, exposing 37
Cast method 60
character arrays 27
character strings
accessing and creating 39
creating persistent 40
persistent 40
transtent 40
in type description files 39
class attribute 32
class factory
    and progid attribute 31
class ID 28
class method implementation 34
classinterface attribute 30
classuuid attribute 30
clients, single-threaded and multi-threaded 28
Close method
    C# syntax 26
description 55
CloseDatabase method
    Books example 14
CLSID 28
collections
    accessing 40
    creating 42
    defining in type definition file 41
deleting 43
    exposing, example 41
    iterating over 42
    querying 43
COM
    Automation error 44
    objects 14
COM class
    registration 20
COM interfaces
    creating with ATL 13
    IUnknown and IDispatch 10
CommitTrans method
    Books example 14
    C# syntax 26
description 56
configuration options 9
configurations, Automation server 10
constants, C++ and Visual Basic differences 15
constructors, exposing 38
conversion attribute 30
converting types 27
Copy method 59
CreateDatabase method 56
CreateLibrary routine 42
creating objects 36
D
/d option 46
data types
  IOSAXDatabase 58
  IOSAXSegment 58
database objects, defining 15
database roots
  Books example 15
  using and creating 43
DCOM
  client configuration 51
DCOM configuration option 9
DeleteOnRelease function 39
deleting
collections 43
temporary objects 38
distributed configurations 49
DLL
  specifying filename 46
domains, object type 10
E
E_INVALIDARG 44
early binding 9
Equal method 56
definitions 12
  Books 12
exception attribute
  ActiveX exceptions 44
  method specification in .ost file 32
  object specification in .ost file 31
exceptions
  ActiveX 32
  C++, catching and signaling 44
  exception attribute 32
  handlers 44
  and transactions 45
exposing
  C++ constructor 38
  C++ static member function 37
  collections 41
  methods 36
factoryuuid attribute 31
features of ActiveX Interface for ObjectStore 9
filenames, specifying 46
files
  .idl 46
  .ost 18
  .rc 20
  .rgs 20
  .tlb 20
generated by osgentyp utility 46
header 46
header for interface definition 20
implementing COM object 20
interface definitions 20
resource 20
type library binary (TLB) 20
For Each...Next statement
  IOSAXCollection 42
function implementation 33, 34
functions, static member 37
G
generating UUIDs 46
generic interface 27
get_Value, IOSAXDatabase method
  C# syntax 26
global
  properties 14
H
header files 20
helpstring attribute
  library specification in .ost file 29
  method specification in .ost file 32
  object specification in .ost file 31
I
/i option 46
I4 ActiveX data type 27
id attribute 32
IDispatch
  interface and contents of database 27
interface, and ActiveX 10
objects 39

.idl file
  generated by osgentyp 20
syntax for specifying 46

/ih option 46
#include files 45

Initialize method
  IOSAXObjectSore interface 56
  IOSAXSessionPool interface 58
InitializeTransactions method 56
InProc configuration option 9
in-process servers 28
installing 11
instance class interface 32
instance classes 30
  described in .ost file 28
instance containers
  described in .ost file 28
    progid attribute 31
instance interface 32
instancefactoryuuid attribute 31
instanceprogid attribute 31
instances
  described in .ost file 28
interface attribute 31
interface definitions 20
invoking osgentyp 45
IOSAXCollection ActiveX data type 27
IOSAXCollection interface
  described 54
  ObjectStore collections 41
IOSAXDatabase ActiveX data type
  description 27
  specifying 58
IOSAXDatabase interface 55
IOSAXDatabaseRoot interface. 43
IOSAXObjectStore interface 56
IOSAXSegment ActiveX data type
  specifying 58
IOSAXSessionPool interface 58
IOSAXStorage interface 58
IOSAXString ActiveX data type 27
IOSAXString interface
  described 59
  exposing strings 39
IOSAXStringConstructor interface
  creating strings 40
  described 59
IOSAXStringInterface
Ansi method 40
Copy method 40
OEM method 40
IOSAXType interface
described 60
IOSAXUnknownObject interface 60
IsInitialized property 58
isomorphic attribute
library specification in .ost file 29
object specification in .ost file 31
IsPersistent method 56
iterating over collections 42
IUnknown interface 10
L
/l option 46
LanguageInterfaceMaintenanceRelease method 57
LanguageInterfaceMajorRelease method 57
LanguageInterfaceMinorRelease method 57
LanguageInterfaceName method 57
lcid attribute 29
library specification
attributes 29
example 29
syntax 28
LoadATKReference method 57
local configuration 49
Local configuration option 9
LookupDatabase method 57
M
MaintenanceRelease method 57
MajorRelease method 57
marshaling 20
method implementation 33
method specifications
attributes 32
syntax 32
methods
exposing 36
implementing 33
midl compiler 20
MinorRelease method 57
multiple sessions
  IOSAXSessionPool interface 58
    support for 47
multi-threaded clients 28
N
Name method 60
named object 14
null (.NET)
  converting 27
null (C#)
  and database root 43
NULL (C++)
  and database root 43
  converting 27
  in Books example 15
O
object servers 17
  basic operations supported 26
  basic types supported 27
  Books example 13
  building 18
  configuration 27
  creating objects 36
  exceptions 44
  in-process 28
  out-of-process 27
  using with Web servers 50
object specifications
  attributes 30
  example 34
  syntax 30
Object Visual Basic type 27
objects
  creating 36
  not deleted when unreferenced 38
  type domains 10
objectstore attribute 29
OEM code page 40
OEM method 59
Open method
  C# syntax 26
  description 55
OpenDatabase method
  Books example 14
  C# syntax 26
  description 57
OpenDatabaseMVCC method 58
OpenMVCC method 55
options for osgentyp 45
os_array 27
os_bag 27
os_Bag base class 40
os_collection 27
os_Collection base class
  and ActiveX interface 42
os_Collection::query 43
os_database 27
os_dictionary 27
os_Dictionary base class 40
os_List 40
os_set 27
os_Set base class 40
osgentytyp utility
  description 45
  generating exception handlers 44
  generating UUIDs 46
  options 45
  syntax 45
OSNCI
  overview 9
.ost file
  osgentytyp utility 46
    type description file 18
out-of-process servers 27
overview of ActiveX Interface for ObjectStore 9
P
persistent character strings, creating 40
persistent objects 13
PoolSize property 58
ProductName method 58
progid attribute 31
property accessor 32
property setter 32
propget attribute 32
propput attribute 32
propputref attribute 32
Q
querying collections 43
R
R4 ActiveX data type 27
R8 ActiveX data type 27
.rc resource file 20
resource file 20
restricted attribute 33
.rgs file 20
Rollback method
  Books example 14
  C# syntax 26
described 58
root object 14
RPC 9
S
sessions 47
SetCleanup method 38
SetContext method
  and instance classes 36
description 58
SetValue function 36
signaling exceptions 44
single-threaded clients 28
specifying filenames 46
static member functions, exposing 37
storage management
  deleting transient objects 38
  IOSAXStorage interface 58
StorageSystemName method 58
system requirements 11
T
/t option 46
template classes 40
threading model
  single-threaded and multi-threaded clients 28
.tlb file, See type library binary 20
top-level objects
  described in .ost file 28
  object attribute 30
  progid attribute 31
transactions and exceptions 45
transport mechanism 10
type conversions 27
type description file 18
  Books example 13
  contents 28
  library specification 28
  method specifications 32
  object specifications 30
  read by osgentyp 45
  specifying name 46
syntax 28
type library binary 20
  in type description file 28
  specifying name 30
types, support for 27
U
/u option 46
Unicode BSTR 39
uuid attribute
  library specification in .ost file 29
  object specification in .ost file 32
UUIDs
  generating 46
  in library specification 29
  in object specification 30
V
Value, IOSAXDatabase method
description 56
Value, IOSAXString method 59
version attribute
  library specification in .ost file 29
  object specification in .ost file 32
Visual Basic
  project 13
  syntax for database operations 26
Visual C++ project 13
W
WriteLine and strings 39